Presto: Edge-based Load Balancing for Fast Datacenter Networks

Keqiang He[†] Eric Rozner[*] Kanak Agarwal[*]
Wes Felter[*] John Carter[*] Aditya Akella[†]

[†]University of Wisconsin–Madison [*]IBM Research

ABSTRACT

Datacenter networks deal with a variety of workloads, ranging from latency-sensitive small flows to bandwidth-hungry large flows. Load balancing schemes based on flow hashing, *e.g.*, ECMP, cause congestion when hash collisions occur and can perform poorly in asymmetric topologies. Recent proposals to load balance the network require centralized traffic engineering, multipath-aware transport, or expensive specialized hardware. We propose a mechanism that avoids these limitations by (i) pushing load-balancing functionality into the soft network edge (*e.g.*, virtual switches) such that no changes are required in the transport layer, customer VMs, or networking hardware, and (ii) load balancing on fine-grained, near-uniform units of data (flowcells) that fit within end-host segment offload optimizations used to support fast networking speeds. We design and implement such a soft-edge load balancing scheme, called Presto, and evaluate it on a 10 Gbps physical testbed. We demonstrate the computational impact of packet reordering on receivers and propose a mechanism to handle reordering in the TCP receive offload functionality. Presto's performance closely tracks that of a single, non-blocking switch over many workloads and is adaptive to failures and topology asymmetry.

CCS Concepts

•**Networks** → **Network architectures; End nodes; Programmable networks; Data center networks;** *Data path algorithms; Network experimentation;*

Keywords

Load Balancing; Software-Defined Networking;

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SIGCOMM'15, August 17–21, 2015, London, United Kingdom

© 2015 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-3542-3/15/08…$15.00

DOI: http://dx.doi.org/10.1145/2785956.2787507

1. INTRODUCTION

Datacenter networks must support an increasingly diverse set of workloads. Small latency-sensitive flows to support real-time applications such as search, RPCs, or gaming share the network with large throughput-sensitive flows for video, big data analytics, or VM migration. Load balancing the network is crucial to ensure operational efficiency and suitable application performance. Unfortunately, popular load balancing schemes based on flow hashing, *e.g.*, ECMP, cause congestion when hash collisions occur [3, 17, 19, 22, 53, 54, 63] and perform poorly in asymmetric topologies [4, 65].

A variety of load balancing schemes aim to address the problems of ECMP. Centralized schemes, such as Hedera [3] and Planck [54], collect network state and reroute elephant flows when collisions occur. These approaches are fundamentally reactive to congestion and are very coarse-grained due to the large time constraints of their control loops [3] or require extra network infrastructure [54]. Transport layer solutions such as MPTCP [61] can react faster but require widespread adoption and are difficult to enforce in multi-tenant datacenters where customers often deploy customized VMs. In-network reactive distributed load balancing schemes, *e.g.*, CONGA [4] and Juniper VCF [28], can be effective but require specialized networking hardware.

The shortcomings of the above approaches cause us to re-examine the design space for load balancing in datacenter networks. ECMP, despite its limitations, is a highly practical solution due to its proactive nature and stateless behavior. Conceptually, ECMP's flaws are not internal to its operation but are caused by asymmetry in network topology (or capacities) and variation in flow sizes. *In a symmetric network topology where all flows are "mice", ECMP should provide near optimal load balancing*; indeed, prior work [4, 58] has shown the traffic imbalance ECMP imposes across links goes down with an increase in the number of flows and a reduction in the variance of the flow size distribution.

Can we leverage this insight to design a good proactive load balancing scheme without requiring special purpose hardware or modifications to end-point transport? The system we propose answers this in the affirmative. It relies on the datacenter network's *software edge* to transform arbitrary sized flows into a large number of near uniformly sized small sub-

flows and proactively spreads those uniform data units over the network in a balanced fashion. Our scheme is fast (works at 10+ Gbps) and doesn't require network stack configurations that may not be widely supported outside the datacenter (such as increasing MTU sizes). We piggyback on recent trends where several network functions, *e.g.*, firewalls and application-level load balancers, are moving into hypervisors and software virtual switches on end-hosts [10, 36, 51]. Our paper makes a strong case for moving network load balancing functionality out of the datacenter network hardware and into the software-based edge.

Several challenges arise when employing the edge to load balance the network on a sub-flow level. Software is slower than hardware, so operating at 10+ Gbps speeds means algorithms must be simple, light-weight, and take advantage of optimizations in the networking stack and offload features in the NIC. Any sub-flow level load balancing should also be robust against reordering because packets from the same flow can be routed over different network paths which can cause out-of-order delivery. As shown in Section 2, reordering not only impacts TCP's congestion control mechanism, but also imposes significant computational strain on hosts, effectively limiting TCP's achievable bandwidth if not properly controlled. Last, the approach must be resilient to hardware or link failures and be adaptive to network asymmetry.

To this end, we build a proactive load balancing system called Presto. Presto utilizes edge vSwitches to break each flow into discrete units of packets, called *flowcells*, and distributes them evenly to near-optimally load balance the network. Presto uses the maximum TCP Segment Offload (TSO) size (64 KB) as flowcell granularity, allowing for fine-grained load balancing at network speeds of 10+ Gbps. To combat reordering, we modify the Generic Receive Offload (GRO) handler in the hypervisor OS to mitigate the computational burden imposed by reordering and prevent reordered packets from being pushed up the networking stack. Finally, we show Presto can load balance the network in the face of asymmetry and failures.

Our paper makes the following contributions:

1. We design and implement a system, called Presto, that near-optimally load balances links in the network. We show such a system can be built with no changes to the transport layer or network hardware and scales to 10+ Gbps networking speeds. Our approach makes judicious use of middleware already implemented in most hypervisors today: Open vSwitch and the TCP receive offload engine in the OS (Generic Receive Offload, GRO, in the Linux kernel).

2. We uncover the importance of GRO on performance when packets are reordered. At network speeds of 10+ Gbps, current GRO algorithms are unable to sustain line rate under severe reordering due to extreme computational overhead, and hence per-packet load balancing approaches [17, 22] need to be reconsidered. We improve GRO to prevent reordering while ensuring computational overhead is limited. We argue GRO is the most natural place to handle reordering because it can mask reordering in a light-weight manner while simultaneously limiting CPU overhead by having a direct impact on the segment sizes pushed up the networking stack. In addition, our scheme distinguishes loss from reordering and adapts to prevailing network conditions to minimize the time to recover lost packets.

3. Presto achieves near-optimal load balancing in a proactive manner. For that, it leverages symmetry in the network topology to ensure that all paths between a pair of hosts are equally congested. However, asymmetries can arise due to failures. We demonstrate Presto can recover from network failures and adapt to asymmetric network topologies using a combination of fast failover and weighted multipathing at the network edge.

4. Finally, we evaluate Presto on a real 10 Gbps testbed. Our experiments show Presto outperforms existing load balancing schemes (including flowlet switching, ECMP, MPTCP) and is able to track the performance of a single, non-blocking switch (an optimal case) within a few percentage points over a variety of workloads, including trace-driven. Presto improves throughput, latency and fairness in the network and also reduces the flow completion time tail for mice flows.

2. DESIGN DECISIONS AND CHALLENGES

In Presto, we make several design choices to build a highly robust and scalable system that provides near optimal load balancing without requiring changes to the transport layer or switch hardware. We now discuss our design decisions.

2.1 Design Decisions

Load Balancing in the Soft Edge A key design decision in Presto is to implement the functionality in the soft edge (*i.e.*, the vSwitch and hypervisor) of the network. The vSwitch occupies a unique position in the networking stack in that it can easily modify packets without requiring any changes to customer VMs or transport layers. Functionality built into the vSwitch can be made aware of the underlying hardware offload features presented by the NIC and OS, meaning it can be fast. Furthermore, an open, software-based approach prevents extra hardware cost and vendor lock-in, and allows for simplified network management. These criteria are important for providers today [41]. Thanks to projects like Open vSwitch, soft-switching platforms are now fast, mature, open source, adopted widely, remotely configurable, SDN-enabled, and feature-rich [36, 50, 52]. Presto is built on these platforms.

Reactive vs Proactive Load Balancing The second major design decision in Presto is to use a proactive approach to congestion management. Bursty behavior can create transient congestion that must be reacted to before switch buffers overflow to prevent loss (timescales range from 100s of µs to ~4 ms [54]). This requirement renders most of the centralized reactive schemes ineffective as they are often too

slow to react to any but the largest network events, *e.g.*, link failures. Furthermore, centralized schemes can hurt performance when rerouting flows using stale information. Distributed reactive schemes like MPTCP [61] and CONGA [4] can respond to congestion at faster timescales, but have a high barrier to deployment. Furthermore, distributed reactive schemes must take great care to avoid oscillations. Presto takes a proactive, correct-by-design approach to congestion management. That is, if small, near-uniform portions of traffic are equally balanced over a symmetric network topology, then the load-balancing can remain agnostic to congestion and leave congestion control to the higher layers of the networking stack. Presto is only reactive to network events such as link failures. Fortunately, the larger timescales of reactive feedback loops are sufficient in these scenarios.

Load Balancing Granularity ECMP has been shown to be ineffective at load balancing the network, and thus many schemes advocate load balancing at a finer granularity than a flow [4, 17, 22, 28]. A key factor impacting the choice of granularity is operating at high speed. Operating at 10+ Gbps incurs great computational overhead, and therefore host-based load balancing schemes must be fast, light-weight and take advantage of optimizations provided in the networking stack. For example, per-packet load balancing techniques [17] cannot be employed at the network edge because TSO does not work on a per-packet basis. TSO, commonly supported in OSes and NICs, allows for large TCP segments (typically 64 KB in size) to be passed down the networking stack to the NIC. The NIC breaks the segments into MTU-sized packets and copies and computes header data, such as sequence numbers and checksums. When TSO is disabled, a host incurs 100% utilization of one CPU core and can only achieve around 5.5 Gbps [34]. Therefore, per-packet schemes are unlikely to scale to fast networks without hardware support. Limiting overhead by increasing the MTU is difficult because VMs, switches, and routers must all be configured appropriately, and traffic leaving the datacenter must use normal 1500 byte packets. Furthermore, per-packet schemes [17, 22] are likely to introduce significant reordering into the network.

Another possibility is to load balance on flowlets [4, 28]. A flow is comprised of a series of bursts, and a flowlet is created when the inter-arrival time between two packets in a flow exceeds a threshold inactivity timer. In practice, inactivity timer values are between 100-500 µs [4]. These values intend to strike a good balance between load balancing on a sub-flow level and acting as a buffer to limit reordering between flowlets. Flowlets are derived from traffic patterns at the sender, and in practice this means the distribution of flowlet sizes is not uniform. To analyze flowlet sizes, a simple experiment is shown in Figure 1. We connect a sender and a receiver to a single switch and start an `scp` transfer designed to emulate an elephant flow. Meanwhile, other senders are hooked up to the same switch and send to the same receiver. We vary the number of these competing flows and show a stacked histogram of the top 10 flowlet sizes for a 1 GB `scp` transfer with a 500 µs inactivity timer.

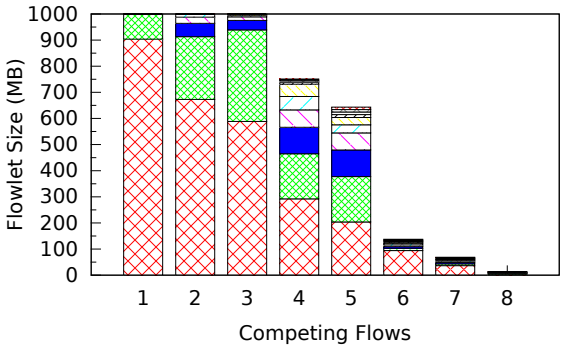


Figure 1: Stacked histogram of flowlet sizes (in MB) for a 1 GB `scp` file transfer. We vary the number of `nuttcp` [44] background flows and denote them as *Competing Flows*. The size of each flowlet is shown within each bar, and flowlets are created whenever there is a 500 µs delay between segments. The top 10 flowlet sizes are shown here. We also analyzed the results of a 1 GB `nuttcp`, `ftp`, and a simple custom client/server transfer and found them to be similar.

The graph shows flowlet sizes can be quite large, with more than half the transfer being attributed to a single flowlet for up to 3 competing flows. Using a smaller inactivity timer, such 100µs, helps (90% of flowlet sizes are 114KB or less), but does not prevent a long tail: 0.1% of flowlets are larger than 1 MB, with the largest ranging from 2.1-20.5 MB. Collisions on large flowlet sizes can lead to congestion. The second problem with flowlets is that small inactivity thresholds, such as 100 µs, can lead to significant reordering. Not only does this impact TCP performance (profiled in Section 5), but it also needlessly breaks small flows into several flowlets. With only one flow in the network, we found a 50 KB mice flow was broken into 4-5 flowlets on average. Small flows typically do not need to be load balanced on a sub-flow level and need not be exposed to reordering.

The shortcomings of the previous approaches lead us to reconsider on what granularity load balancing should occur. Ideally, sub-flow load balancing should be done on near uniform sizes. Also, the unit of load balancing should be small to allow for fine-grained load balancing, but not so small as to break small flows into many pieces or as to be a significant computational burden. As a result, we propose load balancing on 64 KB units of data we call *flowcells*. Flowcells have a number of advantages. First, the maximum segment size supported by TSO is 64 KB, so flowcells provide a natural interface to high speed optimizations provided by the NIC and OS and can scale to fast networking speeds. Second, an overwhelming fraction of mice flows are less than 64 KB in size and thus do not have to worry about reordering [11, 23, 33]. Last, since most bytes in datacenter networks originate from elephant flows [5, 11, 33], this ensures that a significant portion of datacenter traffic is routed on uniform sizes. While promising, this approach must combat reordering to be effective. Essentially we make a tradeoff: the sender avoids congestion by providing fine-grained,

near-uniform load balancing, and the receiver handles reordering to maintain line-rate.

Per-Hop vs End-to-End Multipathing The last design consideration is whether multipathing should be done on a local, per-hop level (*e.g.*, ECMP), or on a global, end-to-end level. In Presto, we choose the latter: pre-configured end-to-end paths are allocated in the network and path selection (and thus multipathing) is realized by having the network edge place flowcells onto these paths. Presto can be used to load-balance in an ECMP style per-hop manner, but the choice of end-to-end multipathing provides additional benefits due to greater control of how flowcells are mapped to paths. Per-hop multipathing can be inefficient under asymmetric topologies [65], and load-balancing on a global end-to-end level can allow for weighted scheduling at the vSwitch to rebalance traffic. This is especially important when failure occurs. The second benefit is flowcells can be assigned over multiple paths very evenly by iterating over paths in a round-robin, rather than randomized, fashion.

2.2 Reordering Challenges

Due to the impact of fine-grained, flowcell-based load balancing, Presto must account for reordering. Here, we highlight reordering challenges. The next section shows how Presto deals with these concerns.

Reordering's Impact on TCP The impact of reordering on TCP is well-studied [37, 47]. Duplicate acknowledgments caused by reordering can cause TCP to move to a more conservative sender state and reduce the sender's congestion window. Relying on parameter tuning, such as adjusting the DUP-ACK threshold, is not ideal because increasing the DUP-ACK threshold increases the time to recover from real loss. Other TCP settings such as Forward Acknowledgement (FACK) assume un-acked bytes in the SACK are lost and degrade performance under reordering. A scheme that introduces reordering should not rely on careful configuration of TCP parameters because (i) it is hard to find a single set of parameters that work effectively over multiple scenarios and (ii) datacenter tenants should not be forced to constantly tune their networking stacks. Finally, many reordering-robust variants of TCP have been proposed [14, 15, 64], but as we will show, GRO becomes ineffective under reordering. Therefore, reordering should be handled below the transport layer.

Computational Bottleneck of Reordering Akin to TSO, Generic Receive Offload (GRO) mitigates the computational burden of receiving 1500 byte packets at 10 Gbps. GRO is implemented in the kernel of the hypervisor, and its handler is called directly by the NIC driver. It is responsible for aggregating packets into larger segments that are pushed up to OVS and the TCP/IP stack. GRO is implemented in the Linux kernel and is used even without virtualization. Similar functionality can be found in Windows (RSC [55]) and hardware (LRO [24]).

Because modern CPUs use aggressive prefetching, the cost of receiving TCP data is now dominated by per-packet, rather than per-byte, operations. As shown by Menon [39], the majority of this overhead comes from buffer management and other routines not related to protocol processing, and therefore significant computational overhead can be avoided by aggregating "raw" packets from the NIC into a single `sk_buff`. Essentially, spending a few cycles to aggregate packets within GRO creates less segments for TCP and prevents having to use substantially more cycles at higher layers in the networking stack. Refer to [31, 39] for detailed study and explanation.

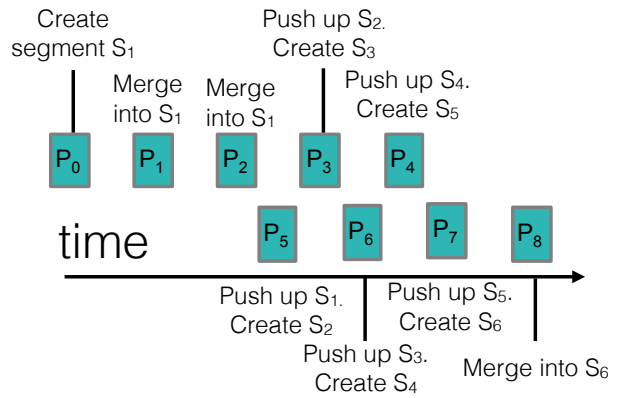


Figure 2: GRO pushes up small segments (S_i) during reordering.

To better understand the problems reordering causes, a brief description of the TCP receive chain in Linux follows. First, interrupt coalescing allows the NIC to create an interrupt for a batch of packets [13, 40], which prompts the driver to poll the packets into an aggregation queue. Next, the driver invokes the GRO handler, located in the kernel, which *merges* the packets into larger segments. The merging continues, possibly across many polling events, until a segment reaches a threshold size, a certain age, or cannot be combined with the incoming packet. Then, the combined, larger segment is *pushed up* to the rest of the TCP/IP networking stack. The GRO process is done on a per-flow level. With GRO disabled, throughput drops to around 5.7-7.1 Gbps and CPU utilization spikes to 100% (Section 5 and [34]). Receive offload algorithms, whether in hardware (LRO) [7, 24] or in software (GRO), are usually *stateless* to make them fast: no state is kept beyond the segment being merged.

We now uncover how GRO breaks down in the face of reordering. Figure 2 shows the impact of reordering on GRO. Reordering does not allow the segment to grow: each reordered packet cannot be merged with the existing segment, and thus the previously created segment must be pushed up. With extreme reordering, GRO is effectively disabled because small MTU-sized segments are constantly pushed up. This causes (i) severe computational overhead and (ii) TCP to be exposed to significant amounts of reordering. We term this the *small segment flooding* problem.

Determining where to combat the reordering problem has not previously taken the small segment flooding problem into account. Using a reordering buffer to deal with reordered packets is a common solution (*e.g.*, works like [17]

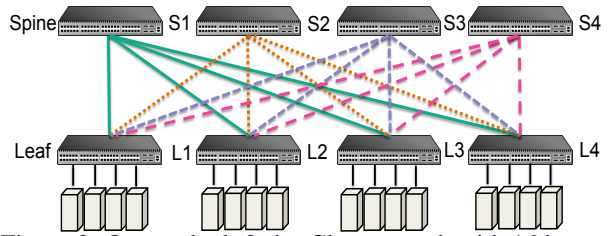


Figure 3: Our testbed: 2-tier Clos network with 16 hosts.

re-sort out-of-order packets in a shim layer below TCP), but a buffer implemented above GRO cannot prevent small segment flooding. Implementing a buffer below GRO means that the NIC must be changed, which is (i) expensive and cumbersome to update and (ii) unlikely to help combat reordering over multiple interrupts.

In our system, the buffer is implemented in the GRO layer itself. We argue this is a natural location because GRO can directly control segment sizes while simultaneously limiting the impact of reordering. Furthermore, GRO can still be applied on packets pushed up from LRO, which means hardware doesn't have to be modified or made complex. Implementing a better GRO algorithm has multiple challenges. The algorithm should be light-weight to scale to fast networking speeds. Furthermore, an ideal scheme should be able to distinguish loss from reordering. When a gap in sequence numbers is detected (*e.g.*, when P_5 is received after P_2 in Figure 2), it is not obvious if this gap is caused from loss or reordering. If the gap is due to reordering, GRO should not push segments up in order to try to wait to receive the missing gap and merge the missing packets into a preestablished segment. If the gap is due to loss, however, then GRO should immediately push up the segments to allow TCP to react to the loss as fast as possible. Ideally, an updated GRO algorithm should ensure TCP does not perform any worse than a scheme with no reordering. Finally, the scheme should adapt to prevailing network conditions, traffic patterns and application demands.

3. DESIGN

This section presents the design of Presto by detailing the sender, the receiver, and how the network adapts in the case of failures and asymmetry.

3.1 Sender

Global Load Balancing at the Network Edge In Presto, a centralized controller is employed to collect the network topology and disseminate corresponding load balancing information to the edge vSwitches. The goal of this design is to ensure the vSwitches, as a whole, can load balance the network in an even fashion, but without requiring an individual vSwitch to have detailed information about the network topology, updated traffic matrices or strict coordination amongst senders. At a high level, the controller partitions the network into a set of multiple spanning trees. Then, the controller assigns each vSwitch a unique forwarding label in each spanning tree. By having the vSwitches partition traffic over these spanning trees in a fine-grained manner, the network can load balance traffic in a near-optimal fashion.

The process of creating spanning trees is made simple by employing multi-stage Clos networks commonly found in datacenters. For example, in a 2-tier Clos network with ν spine switches, the controller can easily allocate ν disjoint spanning trees by having each spanning tree route through a unique spine switch. Figure 3 shows an example with four spine switches and four corresponding disjoint spanning trees. When there are γ links between each spine and leaf switch in a 2-tier Clos network, the controller can allocate γ spanning trees per spine switch. Note that 2-tier Clos networks cover the overwhelming majority of enterprise datacenter deployments and can support tens of thousands of physical servers [4]. The controller ensures links in the network are equally covered by the allocated spanning trees.

Once the spanning trees are created, the controller assigns a unique forwarding label for each vSwitch in every spanning tree and installs the relevant forwarding rules into the network. Forwarding labels can be implemented in a variety of ways using technologies commonly deployed to forward on labels, such as MPLS [18], VXLAN [4, 36], or IP encapsulation [17]. In Presto, label switching is implemented with shadow MACs [1]. Shadow MACs implement label-switching for commodity Ethernet by using the destination MAC address as an opaque forwarding label that can easily be installed in L2 tables. Each vSwitch is assigned one shadow MAC per spanning tree. Note Shadow MACs are extremely scalable on existing chipsets because they utilize the large L2 forwarding table. For example, Trident II-based switches [8, 16, 21] have 288k L2 table entries and thus 8-way multipathing (*i.e.*, each vSwitch has 8 disjoint spanning trees) can scale up to 36,000 physical servers. To increase scalability, shadow MAC tunnels can be implemented from edge switch to edge switch instead of from vSwitch to vSwitch. Switch-to-switch tunneling has been proposed in previous works such as MOOSE [57] and NetLord [43]. Tunneling requires $\mathcal{O}(|\text{switches}| \times |\text{paths}|)$ rules instead of $\mathcal{O}(|\text{vSwitches}| \times |\text{paths}|)$ rules. All shadow MAC labels can route to a destination edge switch that forwards the packet to the correct destination by forwarding on L3 information.

Finally, we note that shadow MACs are also compatible with network virtualization (both L2 and L3 address space virtualization). Tunneling techniques such as VXLAN encapsulate packets in Ethernet frames, which means shadow MACs should still allow path selection in virtualized environments by modifying outer Ethernet headers. VXLAN hardware offload is supported in modern NICs and has little performance overhead [60].

Load Balancing at the Sender After the controller installs the shadow MAC forwarding rules into the network, it creates a mapping from each physical destination MAC address to a list of corresponding shadow MAC addresses. These mappings provide a way to send traffic to a specific destination over different spanning trees. The mappings are pushed from the controller to each vSwitch in the network, either on-demand or preemptively. In Presto, the vSwitch on the

Algorithm 1 Pseudo-code of flowcell creation

```
if bytecount + len(skb) > threshold then
    bytecount ← len(skb)
    current_mac ← (current_mac + 1) % total_macs
    flowcellID ← flowcellID + 1
else
    bytecount ← bytecount + len(skb)
end if
skb ← update(skb, current_mac, flowcellID)
sendToNIC(skb)
```

sender monitors outgoing traffic (*i.e.*, maintains a per-flow counter in the datapath) and rewrites the destination MAC address with one of the corresponding shadow MAC addresses. The vSwitch assigns the same shadow MAC address to all consecutive segments until the 64 KB limit is reached. In order to load balance the network effectively, the vSwitch iterates through destination shadow MAC addresses in a round-robin fashion. This allows the edge vSwitch to load balance over the network in a very fine-grained fashion.

Sending each 64 KB worth of flowcells over a different path in the network can cause reordering and must be carefully addressed. To assist with reordering at the receiver (Presto's mechanisms for combatting reordering are detailed in the next section), the sender also includes a sequentially increasing *flowcell ID* into each segment. In our setup the controller installs forwarding rules solely on the destination MAC address and ARP is handled in a centralized manner. Therefore, the source MAC address can be used to hold the flowcell ID. Other options are possible, *e.g.*, some schemes include load balancing metadata in the reserved bits of the VXLAN header [26] and implementations could also stash flowcell IDs into large IPv6 header fields.[1] Note that since the flowcell ID and the shadow MAC address are modified before a segment is handed to the NIC, the TSO algorithm in the NIC replicates these values to all derived MTU-sized packets. The pseudo-code of flowcell creation is presented in Algorithm 1. Since this code executes in the vSwitch, retransmitted TCP packets run through this code for each retransmission.

3.2 Receiver

The main challenge at the receiver is dealing with reordering that can occur when different flowcells are sent over different paths. The high-level goal of our receiver implementation is to mitigate the effects of the small segment flooding problem by (i) not so aggressively pushing up segments if they cannot be merged with an incoming packet and (ii) ensuring that segments pushed up are delivered in-order.

Mitigating Small Segment Flooding Let's use Figure 2 as a motivating example on how to combat the small segment flooding problem. Say a polling event has occurred, and the driver retrieves 9 packets from the NIC (P_0-P_8). The driver calls the GRO handler, which merges consecutive packets into larger segments. The first three packets (P_0-P_2) are merged into a segment, call it S_1 (note: in practice S_1 already contains in-order packets received before P_0). When P_5 arrives, a new segment S_2, containing P_5, should be created. Instead of pushing up S_1 (as is done currently), both segments should be kept. Then, when P_3 is received, it can be merged into S_1. Similarly, P_6 can be merged into S_2. This process can continue until P_4 is merged into S_1. At this point, the gap between the original out-of-order reception (P_2-P_5) has been filled, and S_1 can be pushed up and S_2 can continue to grow. This means the size of the segments being pushed up is increased, and TCP is not exposed to reordering.

[1]In our implementation, TCP options hold the flowcell ID for simplicity and ease of debugging.

Algorithm 2 Pseudo-code of Presto GRO `flush` function

```
for each flow f do
    for S ∈ f.segment_list do
        if f.lastFlowcell == getFlowcell(S) then
            f.expSeq ← max(f.expSeq, S.endSeq)
            pushUp(S)
        else if getFlowcell(S) > f.lastFlowcell then
            if f.expSeq == S.startSeq then
                f.lastFlowcell ← getFlowcell(S)
                f.expSeq ← S.endSeq
                pushUp(S)
            else if f.expSeq > S.startSeq then
                f.lastFlowcell ← getFlowcell(S)
                pushUp(S)
            else if timeout(S) then
                f.lastFlowcell ← getFlowcell(S)
                f.expSeq ← S.endSeq
                pushUp(S)
            end if
        else
            pushUp(S)
        end if
    end for
end for
```

The current default GRO algorithm works as follows. An interrupt by the NIC causes the driver to poll (multiple) packets from the NIC's ring buffer. The driver calls the GRO handler on the received batch of packets. GRO keeps a simple doubly linked list, called `gro_list`, that contains segments, with a flow having at most one segment in the list. When packets for a flow are received in-order, each packet can be merged into the flow's preexisting segment. When a packet cannot be merged, such as with reordering, the corresponding segment is pushed up (ejected from the linked list and pushed up the networking stack) and a new segment is created from the packet. This process is continued until all packets in the batch are serviced. At the end of the polling event, a `flush` function is called that pushes up all segments in the `gro_list`.

Our GRO algorithm makes the following changes. First, multiple segments can be kept per flow in a doubly linked list (called `segment_list`). To ensure the merging process is fast, each linked list is kept in a hash table (keyed on flow).

When an incoming packet cannot be merged with any existing segment, the existing segments are kept and a new segment is created from the packet. New segments are added to the head of the linked list so that merging subsequent packets is typically $\mathcal{O}(1)$. When the merging is completed over all packets in the batch, the `flush` function is called. The `flush` function decides whether to push segments up or to keep them. Segments may be kept so reordered packets still in flight have enough time to arrive and can then be placed in-order before being pushed up. Reordering can cause the linked lists to become slightly out-of-order, so at the beginning of `flush` an insertion sort is run to help easily decide if segments are in-order.

The pseudo-code of our `flush` function is presented in Algorithm 2. For each flow, our algorithm keeps track of the next expected in-order sequence number (`f.expSeq`) and the corresponding flowcell ID of the most recently received in-order sequence number (`f.lastFlowcell`). When the merging is completed, the `flush` function iterates over the sorted segments (S), from lowest sequence number to highest sequence number in the `segment_list` (line 2). The rest of the code is presented in the subsections that follow.

How to Differentiate Loss from Reordering? In the case of no loss or reordering, our algorithm keeps pushing up segments and updating state. Lines 3-5 deal with segments from the same flowcell ID, so we just need to update `f.expSeq` each time. Lines 6-10 represent the case when the current flowcell ID is fully received and we start to receive the next flowcell ID. The problem, however, is when there is a gap that appears between the sequence numbers of the segments. When a gap is encountered, it isn't clear if it is caused from reordering or from loss. If the gap is due to reordering, our algorithm should be conservative and try to wait to receive the packets that "fill in the gap" before pushing segments up to TCP. If the gap is due to loss, however, then we should push up the segments immediately so that TCP can react to the loss as quickly as possible.

To solve this problem, we leverage the fact that all packets carrying the same flowcell ID traverse the same path and should be in-order. This means incoming sequence numbers can be monitored to check for gaps. A sequence number gap within the same flowcell ID is assumed to be a loss, and not reordering, so those packets are pushed up immediately (lines 3-5). Note that because a flowcell consists of many packets (a 64 KB flowcell consists of roughly 42 1500 byte packets), when there is a loss it is likely that it occurs within flowcell boundaries. The corner case, when a gap occurs on the flowcell boundary, leads us to the next design question.

How to Handle Gaps at Flowcell Boundaries? When a gap is detected in sequence numbers at flowcell boundaries, it is not clear if the gap is due to loss or reordering. Therefore, the segment should be held long enough to handle reasonable amounts of reordering, but not so long that TCP cannot respond to loss promptly. Previous approaches that deal with reordering typically employ a large static timeout (10ms) [17]. Setting the timeout artificially high can handle reordering, but hinders TCP when the gap is due to loss. Setting a low timeout is difficult because many dynamic factors, such as delays between segments at the sender, network congestion, and traffic patterns (multiple flows received at the same NIC affect inter-arrival time), can cause large variations. As a result, we devise an adaptive timeout scheme, which monitors recent reordering events and sets a dynamic timeout value accordingly. Presto tracks cases when there is reordering, but no loss, on flowcell boundaries and keeps an exponentially-weighted moving average (EWMA) over these times. Presto then applies a timeout of $\alpha * \text{EWMA}$ to a segment when a gap is detected on flowcell boundaries. Here α is an empirical parameter that allows for timeouts to grow. As a further optimization, if a segment has timed out, but a packet has been merged into that segment in the last $\frac{1}{\beta} * \text{EWMA}$ time interval, then the segment is still held in hopes of preventing reordering. We find α and β work over a wide range of parameters and set both of them to 2 in our experiments. A timeout firing is dealt with in lines 14-18.

How to Handle Retransmissions? Retransmitted TCP packets are pushed up immediately in order to allow the TCP stack to react without delay. If the flowcell ID of the retransmission is the same as the expected flowcell ID, then the retransmission will be pushed up immediately because its sequence number will be $\leq$ f.expSeq. If the flowcell ID is larger than the expected flowcell ID (when the first packet of a flowcell is a retransmission), then the packet is pushed up (line 13). If a retransmitted packet has a smaller flowcell ID than the next expected flowcell ID (a stale packet), then it will be pushed up immediately (line 20). Note we ensure overflow is handled properly in all cases.

3.3 Failure Handling and Asymmetry

When failures occur, Presto relies on the controller to update the forwarding behavior of the affected vSwitches. The controller can simply prune the spanning trees that are affected by the failure, or more generally enforce a weighted scheduling algorithm over the spanning trees. Weighting allows for Presto to evenly distribute traffic over an asymmetric topology. Path weights can be implemented in a simple fashion by duplicating shadow MACs used in the vSwitch's round robin scheduling algorithm. For example, assume we have three paths in total (p_1, p_2 and p_3) and their updated weights are 0.25, 0.5 and 0.25 respectively. Then the controller can send the sequence of p_1, p_2, p_3, p_2 to the vSwitch, which can then schedule traffic over this sequence in a round robin fashion to realize the new path weights. This way of approximating path weights in the face of network asymmetry is similar to WCMP [65], but instead of having to change switch firmware and use scarce on-chip SRAM/TCAM entries, we can push the weighted load balancing entirely to the network edge.

As an added optimization, Presto can leverage any fast failover features that the network supports, such as BGP fast external failover, MPLS fast reroute, or OpenFlow failover groups. Fast failover detects port failure and can move corresponding traffic to a predetermined backup port. Hardware failover latency ranges from several to tens of millisec-

onds [20, 48]. This ensures traffic is moved away from the failure rapidly and the network remains connected when redundant links are available. Moving to backup links causes imbalance in the network, so Presto relies on the controller learning of the network change, computing weighted multipath schedules, and disseminating the schedules to the edge vSwitches.

4. METHODOLOGY

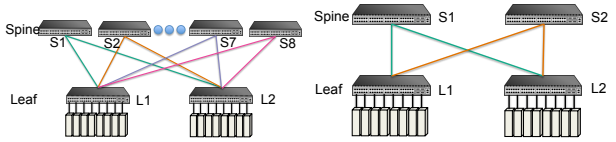
Implementation We implemented Presto in Open vSwitch v2.1.2 [45] and Linux kernel v3.11.0 [38]. In OVS, we modified 5 files and ∼600 lines of code. For GRO, we modified 11 files and ∼900 lines of code.

Testbed We conducted our experiments on a physical testbed consisting of 16 IBM System x3620 M3 servers with 6-core Intel Xeon 2.53GHz CPUs, 60GB memory, and Mellanox ConnectX-2 EN 10GbE NICs. The servers were connected in a 2-tier Clos network topology with 10 Gbps IBM RackSwitch G8264 switches, as shown in Figure 3.

Experiment Settings We ran the default TCP implementation in the Linux kernel (TCP CUBIC [27]) and set parameters `tcp_sack`, `tcp_fack`, `tcp_low_latency` to 1. Further, we tuned the host Receive Side Scaling (RSS) [56] and IRQ affinity settings and kept them the same in all experiments. We send and receive packets from the hypervisor OS instead of VMs. LRO is not enabled on our NICs.

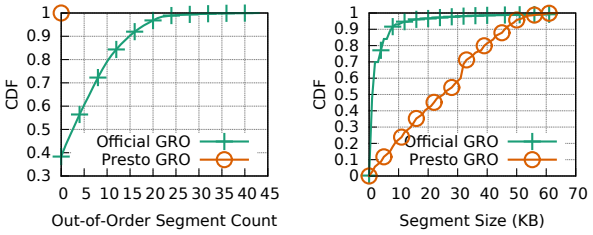
Workloads We evaluate Presto with a set of synthetic and realistic workloads. Similar to previous works [2, 3, 54], our synthetic workloads include: *Shuffle*: Each server in the testbed sends 1GB data to every other server in the testbed in random order. Each host sends two flows at a time. This workload emulates the shuffle behavior of Hadoop workloads. *Stride(8)*: We index the servers in the testbed from left to right. In stride(8) workload, server[i] sends to server[(i+8) mod 16]. *Random*: Each server sends to a random destination not in the same pod as itself. Multiple senders can send to the same receiver. *Random Bijection*: Each server sends to a random destination not in the same pod as itself. Different from random, each server only receives data from one sender. Finally, we also evaluate Presto with trace-driven workloads from real datacenter traffic [33].

Performance Evaluation We compare Presto to ECMP, MPTCP, and a single non-blocking switch used to represent an optimal scenario. ECMP is implemented by enumerating all possible end-to-end paths and randomly selecting a path for each flow. MPTCP uses ECMP to determine the paths of each of its sub-flows. The MPTCP implementation is still under active development, and we spent significant effort in finding the most stable configuration of MPTCP on our testbed. Ultimately, we found that Mellanox `mlx_en4` driver version 2.2, MPTCP version 0.88 [42], subflow count set to 8, OLIA congestion control algorithm [35], and configured buffer sizes as recommended by [35, 46, 53] gave us the best trade-offs in terms of throughput, latency, loss and stability. Unfortunately, despite our efforts, we still occasionally witness some stability issues with MPTCP that we believe are due to implementation bugs.



(a) (b)

Figure 4: (a) Scalability benchmark and (b) Oversubscription benchmark topology.



(a) (b)

Figure 5: (a) Illustration of the modified GRO's effectiveness on masking reordering. (b) In case of massive packet reordering, official GRO cannot merge packets effectively such that lots of small packets are processed by TCP which poses great processing overhead for CPU.

We evaluate Presto on various performance metrics, including: throughput (measured by `nuttcp`), round trip time (a single TCP packet, measured by `sockperf` [59]), mice flow completion time (time to send a 50 KB flow and receive an application-layer acknowledgement), packet loss (measured from switch counters), and fairness (Jain's fairness index [30] over flow throughputs). Mice flows are sent every 100 ms and elephant flows last 10 seconds. Each experiment is run for 10 seconds over 20 runs. Error bars on graphs denote the highest and lowest value over all runs.

5. MICROBENCHMARKS

We first evaluate the effectiveness of Presto over a series of microbenchmarks: (i) Presto's effectiveness in preventing the small segment flooding problem and reordering, (ii) Presto's CPU overhead, (iii) Presto's ability to scale to multiple paths, (iv) Presto's ability to handle congestion, (v) comparison to flowlet switching, and (vi) comparison to local, per-hop load balancing.

Presto's GRO Combats Reordering To examine Presto's ability to handle packet reordering, we perform a simple experiment on the topology shown in Figure 4b. Here two servers attached to leaf switch L1 send traffic to their own receivers attached to leaf switch L2 by spreading flowcells over two network paths. This setup can cause reordering for each flow, so we compare Presto's GRO to an unmodified GRO, denoted "Official GRO". The amount of reordering exposed to TCP is presented in Figure 5a. To quantify packet reordering, we show a CDF of the *out-of-order segment count*: *i.e.*, the number of segments from other flow-

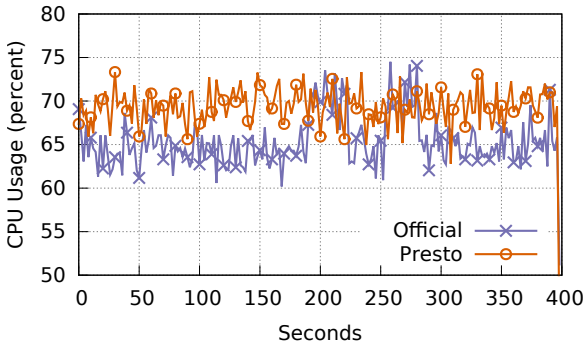


Figure 6: Presto incurs 6% CPU overhead on average.

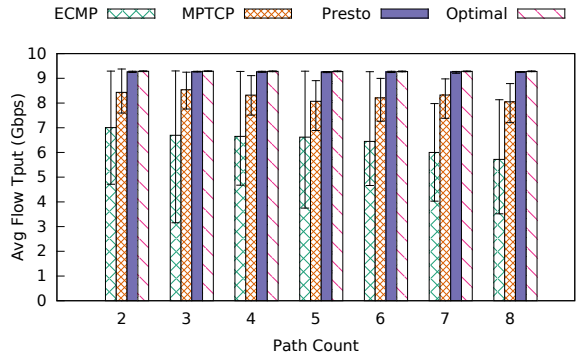


Figure 7: Throughput comparison in scalability benchmark. We denote the non-blocking case as Optimal.

cells between the first packet and last packet of each flowcell. A value of zero means there is no reordering and larger values mean more reordering. The figure shows Presto's GRO can completely mask reordering while official GRO incurs significant reordering. As shown in Section 2, reordering can also cause smaller segments to be pushed up the networking stack, causing significant processing overhead. Figure 5b shows the received TCP segment size distribution. Presto's GRO pushes up large segments, while the official GRO pushes up many small segments. The average TCP throughputs in official GRO and Presto GRO are 4.6 Gbps (with 86% CPU utilization) and 9.3 Gbps (with 69% CPU utilization), respectively. Despite the fact that official GRO only obtains about half the throughput of Presto's GRO, it still incurs more than 24% higher CPU overhead. Therefore, an effective scheme must deal with both reordering and small segment overhead.

Presto Imposes Limited CPU Overhead We investigate Presto's CPU usage by running the stride workload on a 2-tier Clos network as shown in Figure 3. For comparison, official GRO is run with the stride workload using a non-blocking switch (so there is no reordering). Note both official GRO and Presto GRO can achieve 9.3 Gbps. The receiver CPU usage is sampled every 2 seconds over a 400 second interval, and the time-series is shown in Figure 6. On average, Presto GRO only increases CPU usage by 6% compared with the official GRO. The minimal CPU overhead comes from Presto's careful design and implementation. At the sender, Presto needs just two `memcpy` operations (1 for shadow MAC rewriting, 1 for flowcell ID encoding). At the receiver, Presto needs one `memcpy` to rewrite the shadow MAC back to the real MAC and also incurs slight overhead because multiple segments are now kept per flow. The overhead of the latter is reduced because these segments are largely kept in reverse sorted order, which means `merge` on an incoming packet is usually $\mathcal{O}(1)$. The insertion sort is done at the beginning of each `flush` event over a small number of mostly in-order segments, which amortizes overhead because it is called infrequently compared to `merge`.

Presto Scales to Multiple Paths We analyze Presto's ability to scale in the number of paths by setting the number of flows (host pairs) equal to the number of available paths in

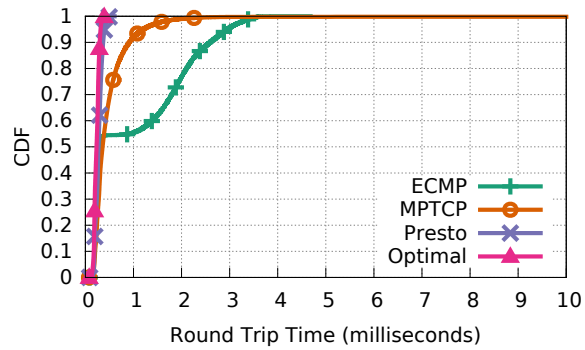
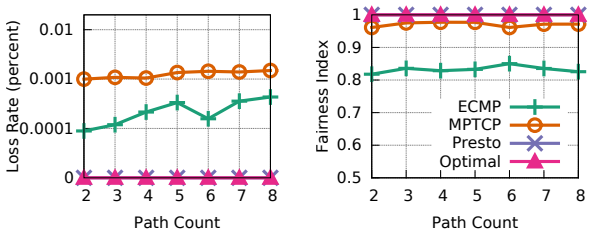


Figure 8: Round trip time comparison in scalability benchmark.

the topology shown in Figure 4a. The number of paths is varied from 2 to 8, and Presto always load-balances over all available paths. Figure 7 shows Presto's throughput closely tracks Optimal. ECMP (and MPTCP) suffer from lower throughput when flows (or subflows) are hashed to the same path. Hashing on the same path leads to congestion and thus increased latency, as shown in Figure 8. Because this topology is non-blocking and Presto load-balances in a near optimal fashion, Presto's latency is near Optimal. Packet drop rates are presented in Figure 9a and show Presto and Optimal have no loss. MPTCP has higher loss because of its bursty nature [4] and its aggression in the face of loss: when a single loss occurs, only one subflow reduces its rate. The other schemes are more conservative because a single loss reduces the rate of the whole flow. Finally, Figure 9b shows Presto, Optimal and MPTCP achieve almost perfect fairness.

Presto Handles Congestion Gracefully Presto's ability to handle congestion is analyzed by fixing the number of spine and leaf switches to 2 and varying the number of flows (host pairs) from 2 to 8, as shown in Figure 4b. Each flow sends as much as possible, which leads to the network being oversubscribed by a ratio of 1 (two flows) to 4 (eight flows). Figure 10 shows all schemes track Optimal in highly oversubscribed environments. ECMP does poorly under moderate congestion because the limited number of flows can be hashed to the same path. Presto does no worse in terms of latency (Figure 11) and loss (Figure 12a). The long tail latency



(a) (b)

Figure 9: (a) Loss rate and (b) Fairness index comparison in scalability benchmark.

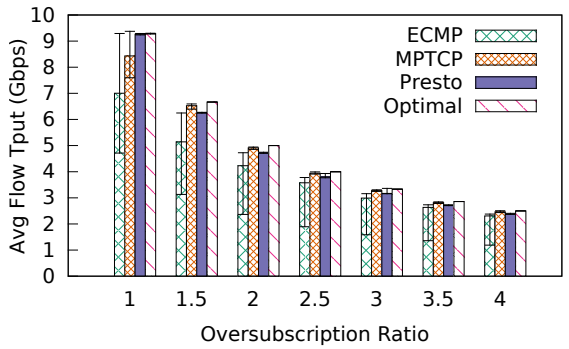


Figure 10: Throughput comparison in oversubscription benchmark.

for MPTCP is caused by its higher loss rates. Both Presto and MPTCP have greatly improved fairness compared with ECMP (Figure 12b).

Comparison to Flowlet Switching We first implemented a flowlet load-balancing scheme in OVS that detects inactivity gaps and then schedules flowlets over disjoint paths in a round robin fashion. The receiver for flowlets uses official GRO. Our flowlet scheme is not a direct reflection of CONGA because (i) it is not congestion-aware and (ii) the flowlets are determined in the software edge instead of the networking hardware. Presto is compared to 500 µs and 100 µs inactivity timers in the stride workload on the 2-tier Clos network (Figure 3). The throughput of the schemes are 9.3 Gbps (Presto), 7.6 Gbps (500 µs), and 4.3 Gbps (100 µs). Analysis of the 100 µs network traces show 13%-29% packets in the connection are reordered, which means 100 µs is not enough time to allow packets to arrive in-order at the destination and thus throughput is severely impacted. Switching flowlets with 500 µs prevents most reordering (only 0.03%-0.5% packets are reordered), but creates very large flowlets (see Figure 1). This means flowlets can still suffer from collisions, which can hurt throughput (note: while not shown here, 500 µs outperforms ECMP by over 40%). Figure 13 shows the latencies. Flowlet 100 µs has low throughput and hence lower latencies. However, since its load balancing isn't perfect, it can still cause increased congestion in the tail. Flowlet 500 µs also has larger tail latencies because of more pronounced flowlet collisions. As compared to the

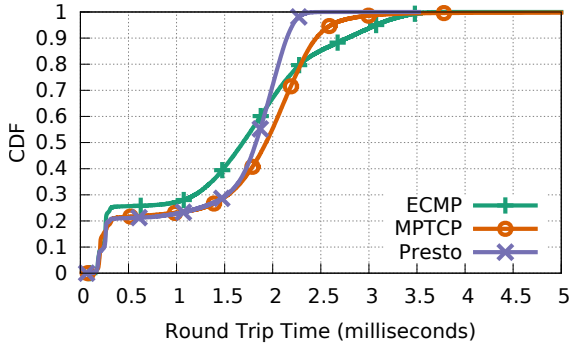
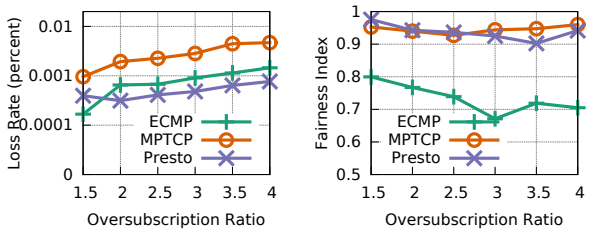


Figure 11: Round trip time comparison in oversubscription benchmark.



(a) (b)

Figure 12: (a) Loss rate and (b) Fairness index comparison in oversubscription benchmark.

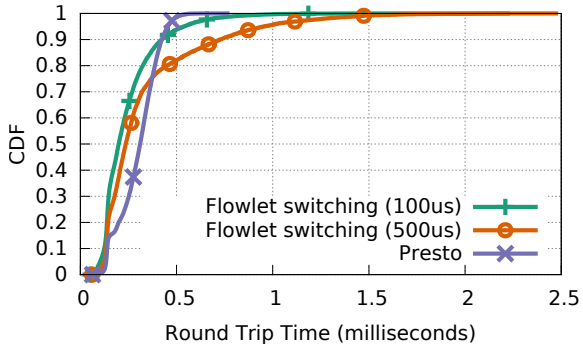


Figure 13: Round trip time comparison of flowlet switching and Presto in Stride workload. The throughputs of Flowlet switching with 100 µs gap, 500 µs gap and Presto are 4.3 Gbps, 7.6 Gbps and 9.3 Gbps respectively.

flowlet schemes, Presto decreases 99.9th percentile latency by 2x-3.6x.

Comparison to Local, Per-Hop Load Balancing Presto sends flowcells in a round robin fashion over pre-configured end-to-end paths. An alternative is to have ECMP hash on flowcell ID and thus provide per-hop load balancing. We compare Presto + shadow MAC with Presto + ECMP using a stride workload on our testbed. Presto + shadow MAC's average throughput is 9.3 Gbps while Presto + ECMP's is 8.9 Gbps. The round trip time CDF is shown in Figure 14. Presto + shadow MAC gives better latency performance compared with Presto + ECMP. The performance difference comes from the fact that Presto + shadow MAC provides better fine-

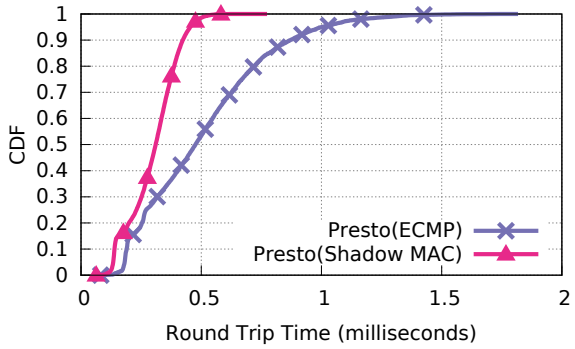


Figure 14: Round trip time comparison between Presto + shadow MAC and Presto + ECMP.

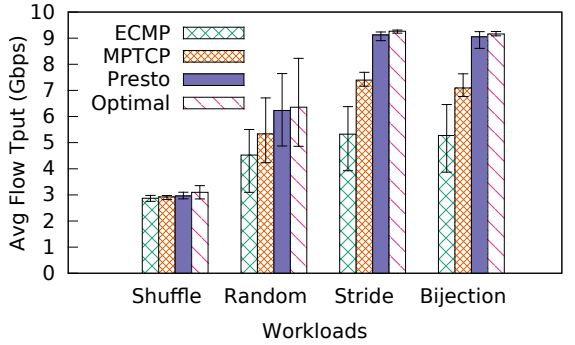


Figure 15: Elephant flow throughput for ECMP, MPTCP, Presto and Optimal in shuffle, random, stride and random bijection workloads.

grained flowcell load balancing because randomization in per-hop multipathing can lead to corner cases where a large fraction of flowcells get sent to the same link over a small timescale by multiple flows. This transient congestion can lead to increased buffer occupancy and higher delays.

6. EVALUATION

In this section, we analyze the performance of Presto for (i) synthetic workloads, (ii) trace-driven workloads, (iii) workloads containing north-south cross traffic, and (iv) failures. All tests are run on the topology in Figure 3.

Synthetic Workloads Figure 15 shows the average throughputs of elephant flows in the shuffle, random, stride and random bijection workloads. Presto's throughput is within 1-4% of Optimal over all workloads. For the shuffle workload, ECMP, MPTCP, Presto and Optimal show similar results because the throughput is mainly bottlenecked at the receiver. In the non-shuffle workloads, Presto improves upon ECMP by 38-72% and improves upon MPTCP by 17-28%.

Figure 16 shows a CDF of the mice flow completion time (FCT) for each workload. The stride and random bijection workloads are non-blocking, and hence the latency of Presto closely tracks Optimal: the 99.9th percentile FCT for Presto is within 350 µs for these workloads. MPTCP and ECMP suffer from congestion, and therefore the tail FCT is much worse than Presto: ECMP's 99.9th percentile FCT is over

Percentile	ECMP	Optimal	Presto
50%	1.0	−12%	−9%
90%	1.0	−34%	−32%
99%	1.0	−63%	−56%
99.9%	1.0	−61%	−60%

Table 1: Mice (<100KB) FCT in trace-driven workload [33]. Negative numbers imply shorter FCT.

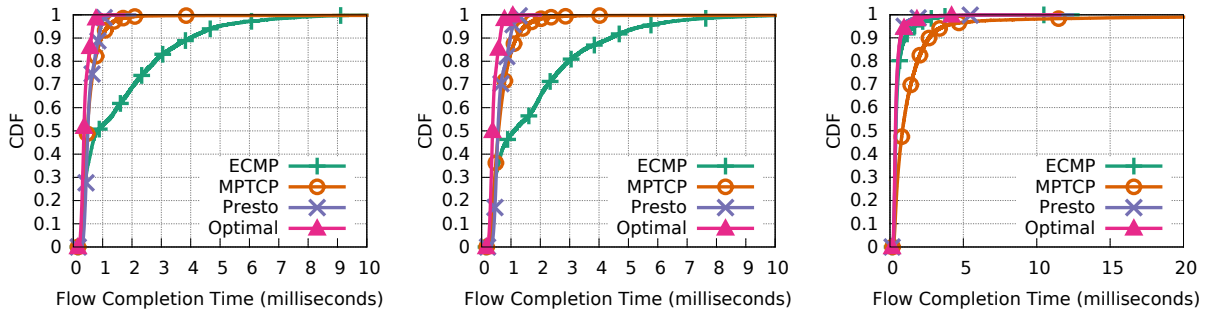
7.5x worse (∼11ms) and MPTCP experiences timeout (because of higher loss rates and the fact that small sub-flow window sizes from small flows can increase the chances of timeout [53]). We used the Linux default timeout (200 ms) and trimmed graphs for clarity. The difference in the random and shuffle workloads is less pronounced (we omit random due to space constraints). In these workloads elephant flows can collide on the last-hop output port, and therefore mice FCT is mainly determined by queuing latency. In shuffle, the 99.9th percentile FCT for ECMP, Presto and Optimal are all within 10% (MPTCP again experiences TCP timeout) and in random, the 99.9th percentile FCT of Presto is within 25% of Optimal while ECMP's is 32% worse than Presto.

Trace-driven Workload We evaluate Presto using a trace-driven workload based on traffic patterns measured in [33]. Each server establishes a long-lived TCP connection with every other server in the testbed. Then each server continuously samples flow sizes and inter-arrival times and each time sends to a random receiver that is not in the same rack. We scale the flow size distribution by a factor of 10 to emulate a heavier workload. Mice flows are defined as flows that are less than 100 KB in size, and elephant flows are defined as flows that are greater than 1 MB. The mice FCT, normalized to ECMP, is shown in Table 1. Compared with ECMP, Presto has similar performance at the 50th percentile but reduces the 99th and 99.9th percentile FCT by 56% and 60%, respectively. Note MPTCP is omitted because its performance was quite unstable in workloads featuring a large number of small flows. The average elephant throughput (not shown) for Presto tracks Optimal (within 2%), and improves upon ECMP by over 10%.

Percentile	ECMP	Optimal	Presto	MPTCP
50%	1.0	−34%	−20%	−12%
90%	1.0	−83%	−79%	−73%
99%	1.0	−89%	−86%	−73%
99.9%	1.0	−91%	−87%	TIMEOUT

Table 2: FCT comparison (normalized to ECMP) with ECMP load balanced north-south traffic. Optimal means all the hosts are attached to a single switch.

Impact of North-South Cross Traffic Presto load balances on "east-west" traffic in the datacenter, *i.e.*, traffic originating and ending at servers in the datacenter. In a real datacenter environment "north-south" traffic (*i.e.*, traffic with an endpoint outside the datacenter) must also be considered. To study the impact of north-south traffic on Presto, we attach



(a) Stride (b) Random Bijection (c) Shuffle

Figure 16: Mice FCT of ECMP, MPTCP, Presto and Optimal in stride, random bijection, and shuffle workloads.

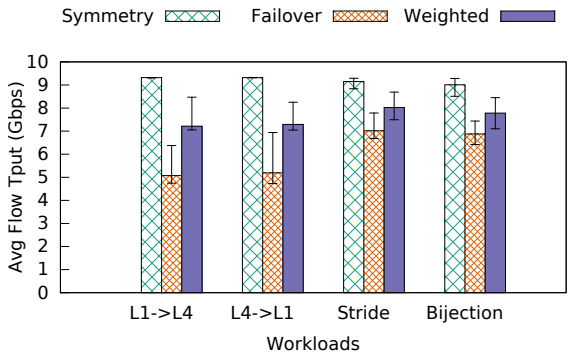


Figure 17: Presto's throughput in symmetry, fast failover and weighted multipathing stages for different workloads.

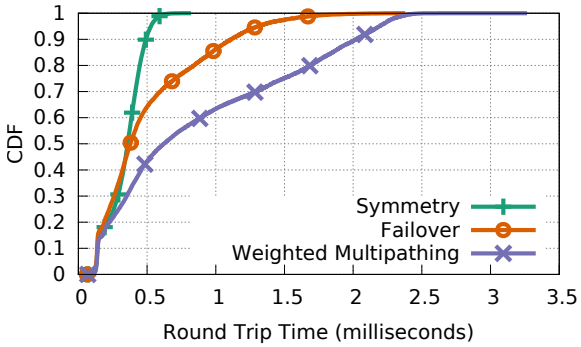


Figure 18: Presto's RTT in symmetry, fast failover and weighted multipathing stages in random bijection workload.

an additional server to each spine switch in our testbed to emulate remote users. The 16 servers establish a long-lived TCP connection with each remote user. Next, each server starts a flow to a random remote user every 1 millisecond. This emulates the behavior of using ECMP to load balance north-south traffic. The flow sizes for north-south traffic are based on the distribution measurement in [29]. The throughput to remote users is limited to 100Mbps to emulate the limitation of an Internet WAN. Along with the north-south flows, a stride workload is started to emulate the east-west traffic. The east-west mice FCT is shown in Table 2 (normalized to ECMP). ECMP, MPTCP, Presto, and Optimal's average throughput is 5.7, 7.4, 8.2, and 8.9Gbps respectively. The experiment shows Presto can gracefully co-exist with north-south cross traffic in the datacenter.

Impact of Link Failure Finally, we study the impact of link failure. Figure 17 compares the throughputs of Presto when the link between spine switch S1 and leaf switch L1 goes down. Three stages are defined: symmetry (the link is up), failover (hardware fast-failover moves traffic from S1 to S2), and weighted (the controller learns of the failure and prunes the tree with the bad link). Workload L1–>L4 is when each node connected to L1 sends to one node in L4 (L4–>L1 is the opposite). Despite the asymmetry in the topology, Presto still achieves reasonable average throughput at each stage. Figure 18 shows the round trip time of each stage in a random bijection workload. Due to the fact that the network is no longer non-blocking after the link failure, failover and weighted multipathing stages have larger round trip time.

7. RELATED WORK

We summarize the related work into three categories: datacenter traffic load balancing, reducing tail latency and handling packet reordering.

Load Balancing in Datacenters MPTCP [53, 61] is a transport protocol that uses subflows to transmit over multiple paths. CONGA [4] and Juniper VCF [28] both employ congestion-aware flowlet switching [58] on specialized switch chipsets to load balance the network. RPS [22] and DRB [17] evaluate per-packet load balancing on symmetric 1 Gbps networks at the switch and end-host, respectively. The CPU load and feasibility of end-host-based per-packet load balancing for 10+ Gbps networks remains open. Hedera [3], MicroTE [12] and Planck [54] use centralized traffic engineering to reroute traffic based on network conditions. FlowBender [32] reroutes flows when congestion is detected by end-hosts and Fastpass [49] employs a centralized arbiter to schedule path selection for each packet. As compared to these schemes, Presto is the only one that proactively load-balances at line rate for fast networks in a near uniform fashion without requiring additional infrastructure or changes to network hardware or transport layers. Furthermore, to the best of our knowledge, Presto is the first work to explore

the interactions of fine-grained load balancing with built-in segment offload capabilities used in fast networks.

Reducing Tail Latency DeTail [63] is a cross-layer network stack designed to reduce the tail of flow completion times. DCTCP [5] is a transport protocol that uses the portion of marked packets by ECN to adaptively adjust sender's TCP's congestion window to reduce switch buffer occupancy. HULL [6] uses Phantom Queues and congestion notifications to cap link utilization and prevent congestion. In contrast, Presto is a load balancing system that naturally improves the tail latencies of mice flows by uniformly spreading traffic in fine-grained units. QJUMP [25] utilizes priority levels to allow latency-sensitive flows to "jump-the-queue" over low priority flows. PIAS [9] uses priority queues to mimic the Shortest Job First principle to reduce FCTs. Last, a blog post by Casado and Pettit [19] summarized four potential ways to deal with elephants and mice, with one advocating to turn elephants into mice at the edge. We share the same motivation and high-level idea and design a complete system that addresses many practical challenges of using such an approach.

Handling Packet Reordering TCP performs poorly in the face of reordering, and thus several studies design a more robust alternative [14, 15, 64]. Presto takes the position that reordering should be handled below TCP in the existing receive offload logic. In the lower portion of the networking stack, SRPIC [62] sorts reordered packets in the driver after each interrupt coalescing event. While this approach can help mitigate the impact of reordering, it does not sort packets across interrupts, have a direct impact on segment sizes, or distinguish between loss and reordering.

8. CONCLUSION

In this paper, we present Presto: a near uniform sub-flow distributed load balancing scheme that can near optimally load balance the network at fast networking speeds. Our scheme makes a few changes to the hypervisor soft-edge (vSwitch and GRO) and does not require any modifications to the transport layer or network hardware, making the bar for deployment lower. Presto is explicitly designed to load balance the network at fine granularities and deal with reordering without imposing much overhead on hosts. Presto is flexible and can also deal with failures and asymmetry. Finally, we show the performance of Presto can closely track that of an optimal non-blocking switch, meaning elephant throughputs remain high while the tail latencies of mice flow completion times do not grow due to congestion.

Acknowledgement

We would like to thank Jon Crowcroft (our shepherd) and the anonymous reviewers for their valuable feedback. This work is supported in part by IBM Corporation, National Science Foundation (grants CNS-1302041, CNS-1330308 and CNS-1345249) and the Wisconsin Institute on Software-Defined Datacenters of Madison.

References

[1] K. Agarwal, C. Dixon, E. Rozner, and J. Carter. Shadow MACs: Scalable Label-switching for Commodity Ethernet. In *HotSDN*, 2014.

[2] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM*, 2008.

[3] M. Al-fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *NSDI*, 2010.

[4] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, F. Matus, R. Pan, N. Yadav, G. Varghese, et al. CONGA: Distributed Congestion-aware Load Balancing for Datacenters. In *SIGCOMM*, 2014.

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

[6] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. Less Is More: Trading a Little Bandwidth for Ultra-Low Latency in the Data Center. In *NSDI*, 2012.

[7] G. Antichi, C. Callegari, and S. Giordano. Implementation of TCP Large Receive Offload on Open Hardware Platform. In *Proceedings of the First Edition Workshop on High Performance and Programmable Networking*, HPPN '13, 2013.

[8] Arista 7250QX Series 10/40G Data Center Switches. http://www.arista.com/assets/data/pdf/Datasheets/7250QX-64_Datasheet.pdf.

[9] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang. Information-Agnostic Flow Scheduling for Commodity Data Centers. In *NSDI*, 2015.

[10] H. Ballani, P. Costa, C. Gkantsidis, M. P. Grosvenor, T. Karagiannis, L. Koromilas, and G. O'Shea. Enabling End Host Network Functions. In *SIGCOMM*, 2015.

[11] T. Benson, A. Akella, and D. A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *IMC*, 2010.

[12] T. Benson, A. Anand, A. Akella, and M. Zhang. MicroTE: Fine Grained Traffic Engineering for Data Centers. In *CoNEXT*, 2011.

[13] C. Benvenuti. *Understanding Linux Network Internals*. " O'Reilly Media, Inc.", 2006.

[14] E. Blanton and M. Allman. On Making TCP More Robust to Packet Reordering. *ACM SIGCOMM Computer Communication Review*, 2002.

[15] S. Bohacek, J. P. Hespanha, J. Lee, C. Lim, and K. Obraczka. TCP-PR: TCP for Persistent Packet Reordering. In *ICDCS*, 2003.

[16] Broadcom Extends Leadership with New StrataXGS Trident II Switch Series Optimized for Cloud-Scale Data Center Networks. http://www.broadcom.com/press/release.php?id=s702418, 2012.

[17] J. Cao, R. Xia, P. Yang, C. Guo, G. Lu, L. Yuan, Y. Zheng, H. Wu, Y. Xiong, and D. Maltz. Per-packet Load-balanced, Low-latency Routing for Clos-based Data Center Networks. In *CoNEXT*, 2013.

[18] M. Casado, T. Koponen, S. Shenker, and A. Tootoonchian. Fabric: A Retrospective on Evolving SDN. In *HotSDN*, 2012.

[19] M. Casado and J. Pettit. Of Mice and Elephants. http://networkheresy.com/2013/11/01/of-mice-and-elephants/, November 2013.

[20] W. Chou and M. Luo. Agile Network, Agile World: the SDN Approach. http://huaweiempresas.com/img_eventos/140314/SDN_Workshop_2.pdf.

[21] S. Das. Smart-Table Technology—Enabling Very Large Server, Storage Nodes and Virtual Machines to Scale Using

Flexible Network Infrastructure Topologies. http://www.broadcom.com/collateral/wp/StrataXGS_SmartSwitch-WP101-R.pdf, July 2012.
[22] A. Dixit, P. Prakash, Y. C. Hu, and R. R. Kompella. On the Impact of Packet Spraying in Data Center Networks. In *INFOCOM*, 2013.
[23] 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 *SIGCOMM*, 2009.
[24] L. Grossman. Large Receive Offload Implementation in Neterion 10GbE Ethernet Driver. In *Linux Symposium*, 2005.
[25] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. Watson, A. W. Moore, S. Hand, and J. Crowcroft. Queues Don't Matter When You Can JUMP Them! In *NSDI*, 2015.
[26] H. Chen and W. Song. Load Balancing without Packet Reordering in NVO3. Internet-Draft. https://tools.ietf.org/html/draft-chen-nvo3-load-banlancing-00, October 2014.
[27] S. Ha, I. Rhee, and L. Xu. CUBIC: A New TCP-friendly High-speed TCP Variant. *SIGOPS Oper. Syst. Rev.*, 2008.
[28] D. R. Hanks. *Juniper QFX5100 Series: A Comprehensive Guide to Building Next-Generation Networks*. " O'Reilly Media, Inc.", 2014.
[29] K. He, A. Fisher, L. Wang, A. Gember, A. Akella, and T. Ristenpart. Next Stop, the Cloud: Understanding Modern Web Service Deployment in EC2 and Azure. In *IMC*, 2013.
[30] R. Jain, D. Chiu, and W. Hawe. A Quantitative Measure Of Fairness And Discrimination For Resource Allocation In Shared Computer Systems. *CoRR*, 1998.
[31] JLS2009: Generic Receive Offload. https://lwn.net/Articles/358910/.
[32] A. Kabbani, B. Vamanan, J. Hasan, and F. Duchene. FlowBender: Flow-level Adaptive Routing for Improved Latency and Throughput in Datacenter Networks. In *CoNEXT*, 2014.
[33] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The Nature of Data Center Traffic: Measurements & Analysis. In *IMC*, 2009.
[34] R. Kapoor, A. C. Snoeren, G. M. Voelker, and G. Porter. Bullet Trains: A Study of NIC Burst Behavior at Microsecond Timescales. In *CoNEXT*, 2013.
[35] R. Khalili, N. Gast, M. Popovic, U. Upadhyay, and J.-Y. Le Boudec. MPTCP is Not Pareto-optimal: Performance Issues and a Possible Solution. In *CoNEXT*, 2012.
[36] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, N. Gude, P. Ingram, et al. Network Virtualization in Multi-tenant Datacenters. In *NSDI*, 2014.
[37] K.-C. Leung, V. O. Li, and D. Yang. An Overview of Packet Reordering in Transmission Control Protocol (TCP): Problems, Solutions, and Challenges. *Parallel and Distributed Systems, IEEE Transactions on*, 2007.
[38] Linux Kernel. https://www.kernel.org.
[39] A. Menon and W. Zwaenepoel. Optimizing TCP Receive Performance. In *USENIX Annual Technical Conference*, 2008.
[40] J. C. Mogul and K. Ramakrishnan. Eliminating Receive Livelock in An Interrupt-driven Kernel. *ACM Transactions on Computer Systems*, 1997.
[41] T. P. Morgan. A Rare Peek Into The Massive Scale of AWS. http://www.enterprisetech.com/2014/11/14/rare-peek-massive-scale-aws/, November 2014.
[42] MPTCP Linux Kernel Implementation. http://www.multipath-tcp.org.
[43] J. Mudigonda, P. Yalagandula, J. Mogul, B. Stiekes, and Y. Pouffary. NetLord: A Scalable Multi-tenant Network Architecture for Virtualized Datacenters. In *SIGCOMM*, 2011.
[44] Nuttcp. http://www.nuttcp.net/Welcome%20Page.html.
[45] Open vSwitch. http://openvswitch.org.
[46] C. Paasch, R. Khalili, and O. Bonaventure. On the Benefits of Applying Experimental Design to Improve Multipath TCP. In *CoNEXT*, 2013.
[47] V. Paxson. End-to-end Internet Packet Dynamics. In *ACM SIGCOMM Computer Communication Review*, 1997.
[48] I. Pepelnjak. NEC+IBM: Enterprise OpenFlow You can Actually Touch. http://blog.ipspace.net/2012/02/necibm-enterprise-openflow-you-can.html.
[49] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fastpass: A Centralized "Zero-Queue" Datacenter Network. In *SIGCOMM*, 2014.
[50] J. Pettit. Open vSwitch and the Intelligent Edge. In *OpenStack Summit*, 2014.
[51] B. Pfaff, J. Pettit, K. Amidon, M. Casado, T. Koponen, and S. Shenker. Extending Networking into the Virtualization Layer. In *HotNets*, 2009.
[52] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, et al. The Design and Implementation of Open vSwitch. In *NSDI*, 2015.
[53] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving Datacenter Performance and Robustness with Multipath TCP. In *SIGCOMM*, 2011.
[54] J. Rasley, B. Stephens, C. Dixon, E. Rozner, W. Felter, K. Agarwal, J. Carter, and R. Fonseca. Planck: Millisecond-scale Monitoring and Control for Commodity Networks. In *SIGCOMM*, 2014.
[55] Receive Segment Coalescing (RSC). http://technet.microsoft.com/en-us/library/hh997024.aspx.
[56] Receive Side Scaling (RSS). https://technet.microsoft.com/en-us/library/hh997036.aspx.
[57] M. Scott, A. Moore, and J. Crowcroft. Addressing the Scalability of Ethernet with MOOSE. In *Proc. DC CAVES Workshop*, 2009.
[58] S. Sinha, S. Kandula, and D. Katabi. Harnessing TCPs Burstiness using Flowlet Switching. In *HotNets*, 2004.
[59] Sockperf: A Network Benchmarking Utility over Socket API. https://code.google.com/p/sockperf/.
[60] VXLAN Performance Evaluation on VMware vSphere 5.1. http://www.vmware.com/files/pdf/techpaper/VMware-vSphere-VXLAN-Perf.pdf.
[61] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley. Design, Implementation and Evaluation of Congestion Control for Multipath TCP. In *NSDI*, 2011.
[62] W. Wu, P. Demar, and M. Crawford. Sorting Reordered Packets with Interrupt Coalescing. *Computer Networks*, 2009.
[63] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz. DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks. In *SIGCOMM*, 2012.
[64] M. Zhang, B. Karp, S. Floyd, and L. Peterson. RR-TCP: A Reordering-robust TCP with DSACK. In *ICNP*, 2003.
[65] J. Zhou, M. Tewari, M. Zhu, A. Kabbani, L. Poutievski, A. Singh, and A. Vahdat. WCMP: Weighted Cost Multipathing for Improved Fairness in Data Centers. In *EuroSys*, 2014.