SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs

Rui Miao
University of Southern California

Hongyi Zeng
Facebook

Changhoon Kim
Barefoot Networks

Jeongkeun Lee
Barefoot Networks

Minlan Yu
Yale University

ABSTRACT

In this paper, we show that up to hundreds of software load balancer (SLB) servers can be replaced by a single modern switching ASIC, potentially reducing the cost of load balancing by over two orders of magnitude. Today, large data centers typically employ hundreds or thousands of servers to load-balance incoming traffic over application servers. These software load balancers (SLBs) map packets destined to a service (with a virtual IP address, or VIP), to a pool of servers tasked with providing the service (with multiple direct IP addresses, or DIPs). An SLB is stateful, it must always map a connection to the same server, even if the pool of servers changes and/or if the load is spread differently across the pool. This property is called *per-connection consistency* or PCC. The challenge is that the load balancer must keep track of millions of connections simultaneously.

Until recently, it was not possible to implement a load balancer with PCC in a merchant switching ASIC, because high-performance switching ASICs typically can not maintain per-connection states with PCC. Newer switching ASICs provide resources and primitives to enable PCC at a large scale. In this paper, we explore how to use switching ASICs to build much faster load balancers than have been built before. Our system, called SilkRoad, is defined in a 400 line P4 program and when compiled to a state-of-the-art switching ASIC, we show it can load-balance ten million connections simultaneously at line rate.

CCS CONCEPTS

- **Networks** → **Programmable networks**; *Network management*; *Data center networks*;

KEYWORDS

Load balancing; Programmable switches

ACM Reference format:
Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *Proceedings of SIGCOMM '17, Los Angeles, CA, USA, August 21–25, 2017,* 14 pages.
https://doi.org/10.1145/3098822.3098824

1 INTRODUCTION

Stateful layer-4 (L4) load balancers scale out services hosted in cloud datacenters by mapping packets destined to a service with a virtual IP address (VIP) to a pool of servers with multiple direct IP addresses (DIPs or DIP pool). L4 load balancing is a critical function for inbound traffic to the cloud and traffic across tenants. A previous study [36] reports that an average of 44% of cloud traffic is VIP traffic and thus needs load balancing function. Building cloud-scale L4 load balancing faces two major challenges:

Support full bisection traffic with low latency: Data centers have rapid growth in traffic: doubling every year in Facebook [11] and growing by 50 times in six years in Google [40]. While the community has made efforts to scale out L2/L3 virtual switching to match full bisection bandwidth for intra-datacenter traffic (or full gateway capacity for inbound traffic) [17, 30], one missing piece is *scaling L4 load balancers to match the full bisection bandwidth of the underlying physical network*. Load balancing is also a critical segment for the end-to-end performance of delay-sensitive applications [23] and for low latency data centers (e.g., 2-5 μs RTT with RDMA [42]).

Ensure per connection consistency (PCC) during frequent DIP pool changes: Data center networks are constantly changing to handle failures, deploy new services, upgrade existing services, and react to the traffic increase [24]. Each operational change can result in many DIP pool changes. For example, when we upgrade a service, we need to bring down DIPs and upgrade them one by one to avoid affecting the service capacity. Such frequent DIP pool updates are observed from a large web service provider with about a hundred of data center clusters (§3.1).

During a DIP pool change, it is critical to ensure *per connection consistency (PCC)*, which means all the packets of a connection should be delivered to the same DIP. Sending packets of an ongoing connection to a different DIP breaks the connection. It often takes subseconds to seconds for applications to recover from a broken connection (e.g., one second in Wget), which significantly affects user experience.

SIGCOMM '17, August 21–25, 2017, Los Angeles, CA, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-4653-5/17/08...$15.00
https://doi.org/10.1145/3098822.3098824

Today, L4 load balancing is often implemented in software servers [20, 36]. The software load balancer (SLB) can easily support DIP pool updates and ensure PCC, but cannot provide full bisection bandwidth with low latency and low cost. This is because processing packets in software incurs high compute overhead (requiring thousands of servers or around 3.75% of the data center size [20]), high latency and jitter (50 μs to 1 ms) [20, 22], and poor performance isolation (§2.2). In contrast, if we run load balancing in switches, we can process the same amount of traffic with about two orders of magnitude saving in power and capital cost [8, 10].

To improve throughput and latency, offloading load balancing to hardware is an appealing option, similar to offloading packet checksumming, segmenting and rate limiting to NICs [29, 33]. One approach to run load balancing at switches is to leverage ECMP hashing to map a VIP to a DIP pool, but do not maintain *the connection state* at switches [22]. However in this approach, during a DIP pool update, switches need to redirect all the related traffic to SLBs to create the connection state there and ensure PCC. The issue here is that there is no clean way to decide when to migrate the VIP traffic back to switches (§3.2). If we migrate too early, many ongoing connections that should match the old DIP pool may match the new DIP pool at switches and violate PCC. If we migrate too late, the SLBs process most of the traffic (the 'slow-path' problem), losing the throughput/latency benefits of using switches. If we migrate VIPs back to switches periodically as used in Duet [22], it leads to either around 1% of connections broken or up to 70% of traffic handles in SLBs (§3.2).

Instead, we propose SilkRoad, which uses simple hardware primitives available in today's switching ASICs for stateful load balancing. SilkRoad aims to provide a direct path between application traffic and application servers by eliminating the need for another software layer (SLBs) in-between. SilkRoad maintains the connection state at the switch, thus ensuring PCC for all the connections. In addition, every packet of a VIP connection is forwarded by ASIC, and hence SilkRoad inherits all the benefits of high-speed commodity ASICs such as high throughput, low latency and jitter, and better performance isolation. The key challenge for SilkRoad is to maintain connection states in the ASIC using hardware primitives while scaling up to millions of connections as well as ensuring PCC upon frequent DIP pool updates. SilkRoad addresses this challenge via the following contributions.

Fitting millions of connections in SRAM: Switching ASICs have continuously increased memory size (growing by five times over the past four years as shown in Table 1 in §4.1) and has just reached a stage where storing all the connection states and running load balancing at switches become possible. However, with a naive approach, storing the states of ten million connections in a match-action table takes a few hundreds of MB of SRAM.[1] This is far more than 50-100 MB SRAM size available in the latest generation of switching ASICs. To reduce the SRAM usage, we looked at what constitutes each connection entry: match field and action data. We propose to store a small hash of a connection rather than the 5-tuple to reduce the match field size, while our design mitigates the impact of false positives introduced by the hash. To reduce the action data bits of a connection entry, we store a DIP pool *version* rather than the actual DIPs and intelligently reuse version numbers across a series of DIP pool updates.

Ensuring PCC during frequent DIP pool updates: Ensuring PCC is challenging because switches often use the slow software (running on a switch management CPU) to insert new connection entries into the table. Hence, a new connection entry may not be ready for the subsequent packets in a timely fashion. We call those connections that arrive before time t but do not have a connection entry installed in the table at time t as *pending connections*. To ensure PCC during DIP pool updates, SilkRoad remembers pending connections in an on-chip bloom filter, built on commonly available transactional memory (as counters/meters) in ASICs. We minimize the size of the bloom filter using a 3-step update process.

Since SilkRoad uses existing features on ASICs today, it can be implemented by either modifying the logic of existing fixed function ASICs or using programmable ASICs (e.g., Barefoot Tofino [1]). In fact, we built the SilkRoad prototype on a programmable switching ASIC and confirmed fitting 10M connections is feasible.

We performed an extensive simulation with a variety of production traffic and update traces from a large web service provider. The results show that SilkRoad achieves 40%-95% SRAM reduction and thus can fit into switch SRAM for all the clusters we studied. Using the available SRAM in switches, one SilkRoad can replace up to hundreds of SLBs. Our design always ensures PCC with a bloom filter of only 256 bytes even under the scenarios with the most frequent DIP pool updates observed from the network.

2 BACKGROUND ON LOAD BALANCING

In this section, we first give background on the layer 4 load balancing function. Next, we discuss two existing solutions for load balancing: 1) software load balancers running on x86 servers [20, 36] and 2) Duet [22] which stores VIP-to-DIP mappings (not per-connection states) in switching ASICs.

2.1 Layer 4 load balancing function

In this paper, we refer load balancers as layer-4 load balancers as opposed to layer-7 ones (e.g., Nginx [14]). A load balancer maintains two tables (Figure 1):

VIPTable: VIPTable maintains the mapping from a VIP (e.g., a service IP:port tuple 20.0.0.1:80) to a DIP pool (e.g., a server pool {10.0.0.1:20, 10.0.0.2:20}). When the first packet of a connection c_1 arrives, the load balancer identifies the DIP pool for its VIP in VIPTable and runs a hash on the packet header fields (e.g., 5-tuple) to select a DIP (e.g., 10.0.0.2:20). Since the hash is performed based on the same packet header

[1] In the case of IPv6 connection, a connection entry takes 37 bytes to store 5-tuple as match key, 18 bytes to store new destination address plus port number as action data, and a couple bytes of packing overhead.

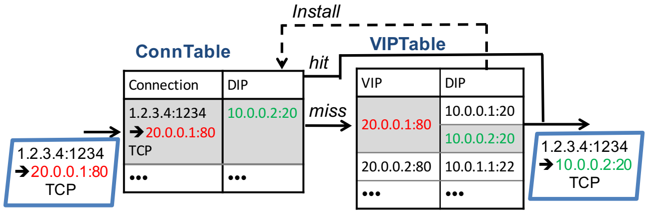


Figure 1: **ConnTable and VIPTable in load balancers.**

fields, all subsequent packets of the same connection pick the same DIP as long as the DIP pool remains static.

When the DIP pool changes for server addition or removal, the packets of the same connection may be hashed to a different DIP, breaking per-connection consistency (PCC). Formally, we define a load balancing (LB) function as a mapping function from packet p_j^i that belongs to connection c_i to DIP d_k. We define PCC as: *for a given connection* c_i, $\forall p_j^i \in c_i$, $LB(p_j^i) = LB(p_0^i)$=d_k.

ConnTable: To ensure PCC during DIP pool updates, a load balancer maintains a table that stores per-connection states. We call it ConnTable. ConnTable maps each connection (e.g., 5-tuple) to a DIP that is selected by the hash result of VIPTable for the first packet of the connection. In Figure 1, when the first packet of c_1 arrives, a match-action rule that maps c_1 to the DIP is inserted into ConnTable (e.g., [1.2.3.4:1234, 20.0.0.1:80, TCP] → 10.0.0.2:20). All the subsequent packets of this connection match the rule in ConnTable and get forwarded to 10.0.0.2:20 consistently.

PCC challenge: Now suppose a new DIP 10.0.0.3:20 is added to the DIP pool for this given VIP, which requires an update of the DIP pool members in VIPTable. The challenge here is, *to guarantee PCC, the VIPTable update must be atomic and synchronous with connection insertions in ConnTable.* In software implementations, the load balancer locks VIPTable and holds new incoming connections in a buffer to prevent them from being processed by VIPTable. Then, the SLB updates VIPTable to the new DIP pool and then releases the new connections from the buffer. This way the SLB ensures PCC for connections arrived both before and after the update, but at the cost of slow-path packet processing by CPU and buffering delay. Later we will also show that this synchronous update between VIPTable and ConnTable is difficult to achieve in switching ASICs (§4.3).

2.2 Limitations of software load balancers

Most cloud data centers today run software load balancers (SLBs) [20, 36] by implementing both ConnTable and VIPTable in software. As mentioned in the Duet paper [22], using software has the following drawbacks:

High cost of server resources: Today's SLBs typically occupy a significant number of servers. For example, a typical 40K-server data center has 15 Tbps traffic for load balancing [22] and requires over 15 Tbps / 10 Gbps=1500 SLB servers or 3.75% of total servers even assuming full NIC line rate processing. In addition to the NIC speed limit for bit-per-second throughput, the state-of-the-art SLBs using 8 CPU cores can only achieve up to 12 Mpps (packet-per-second) throughput [20], three orders of magnitude slower than modern switching ASICs that easily process billions of packets per second.

Cloud traffic volume is rapidly growing (doubling every year in Facebook clusters [11] or growing by 50 times from 2008 to 2014 observed in Google data centers [40]) and lots of the cloud traffic need load balancing (44% of traffic as reported in [36]). Thus, we need even more servers for load balancing, wasting the resources that could otherwise be used for revenue-generating applications. By contrast, if we run load balancing on switches, we process the same amount of traffic with about two orders of magnitude saving in power and capital cost [8, 10].

High latency and jitter: SLBs add a high latency of 50 μs to 1 ms for processing packets in batches [20, 22], which is comparable to the end-to-end RTT in data centers (median 250 μs in [25]). New techniques such as RDMA enable even lower RTT of 2-5 μs [42]. Therefore, SLBs become a severe bottleneck for many delay-sensitive applications [23, 42]. Moreover, fulfilling a service request may trigger a chain of requests for its provider services or third-party services (e.g., storage, data analytics, etc), traversing SLBs multiple times. The accumulated latencies with multiple SLB layers in-between hurt tail latency performance experienced by the application.

Poor performance isolation: When one VIP is under DDoS attacks or experiences a flash crowd, the other VIP traffic served by the same SLB server instance also experience increased delays or even packet drops because of poor resource/performance isolation in x86-based systems. One may employ rate-limiting at SLBs, but software rate-limiting tools (e.g., Linux TC queueing discipline (qdisc)) incur high CPU overhead. For example, metering 6.5 Gbps traffic may require a dedicated 8-core CPU [37]. As a result, we cannot expect fine-grained and efficient performance isolation on SLBs.

2.3 Duet: Storing VIPTable in ASICs

To address the limitations of SLBs, one natural idea is to leverage the high-speed low-cost ASICs that are already available in data center switches. Duet [22] leverages two features in the ASIC: **(1) Generic hash units:** which already exist in ASICs for functions like ECMP, Link Aggregation Group (LAG), checksum verifier, etc. **(2) Match-action tables:** which match on selected packet header fields for various actions and are used for ECMP tables, forwarding tables, etc. Duet uses ECMP hashing and fixed match-action table to implement VIPTable at switches. Due to the limited ECMP table size, Duet only uses switches to handle VIPs with high-volume traffic and employs a few SLBs (with both ConnTable and VIPTable) to handle the other VIPs.

Duet can get around the performance limitations of SLBs. To handle 10 Tbps traffic, the Duet paper claims it forwards most traffic in switches while only needs 230 SLBs to handle around 5% of traffic in software. Duet can achieve a median latency of 474 μs [22]. Duet can also achieve better performance isolation using rate-limiters (meters) at switches. However, as we will show in the next section, Duet cannot

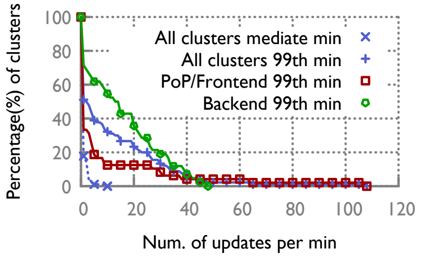


Figure 2: **Frequent DIP pool updates (Y% of clusters have more than X updates per minute in the median or 99th percentile minute in a month.)**

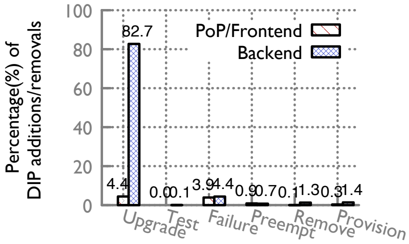


Figure 3: **Distribution of root causes for DIP additions and removals (in a month).**

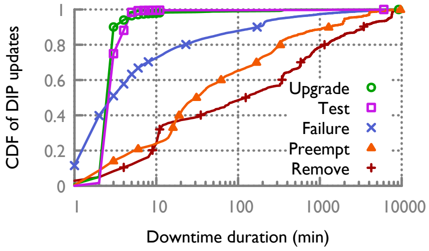


Figure 4: **Distribution of downtime duration with various root causes. (Provisioning does not cause downtime)**

handle frequent DIP pool updates, which is an important scenario in data centers today.

3 CHALLENGES OF FREQUENT DIP POOL UPDATES

From our study of a large web service provider, we observe a major challenge for load balancing functions: frequent DIP pool updates. If we store ConnTable only in SLBs, as proposed in recent works [22], during DIP pool updates, we may incur either a high SLB overhead or many broken connections, which degrade application performance.

3.1 Frequent DIP pool updates

We study about a hundred clusters from a large web service provider. There are three types of clusters: *PoPs* (points of presence) where user requests first arrive at, *Frontends* which serve requests from PoPs, and *Backends* that run backend services. All clusters need L4 load balancing functions.

Frequent DIP pool updates in a cluster: We collect the number of DIP pool update events per minute from the network operation logs in a month. For each cluster, we identify the median and 99th percentile minute in the month. We then draw the distribution across clusters with different update frequency in Figure 2. For example, overall, there are 32% of clusters with more than 10 updates per minute in the 99th percentile minute (which means more than 10 updates per minute for 432 minutes in a month) and 3% of clusters have more than 50 updates. Some clusters experience 10 updates per minute for the median minute (by half of the time).

Backends have more frequent DIP pool updates than PoPs/Frontends. For example, half of the Backends have more than 16 updates in the 99th percentile minute. This indicates a continuous evolution of backend services. But some PoPs/Frontends have more than 100 updates in the 99th percentile minute because there a DIP is often shared by most of the VIPs (similar to [20]) and thus one DIP down or up incurs a burst of updates from all the VIPs.

To understand the burstiness of DIP pool updates across minutes, we also measure the number of DIP pool updates every ten minutes (not shown in the figure). In the 99th percentile ten-minute, 42% of clusters have more than 100 updates and 2% of clusters have more than 500 updates. The latter indicates an average of 50 updates per minute during a 10-minute period.

Why are there frequent DIP pool updates? To understand the sources of frequent DIP pool updates, we analyze service management logs for all clusters in a month. The logs include the transition of operational stages (e.g., reserving the machine, setting up container environment, announcing at the load balancer, etc.) and DIP downtimes. We only select events related to DIP additions and removals.

Figure 3 categorizes the distribution of DIP additions and removals for different root causes. 82.7% of the DIP additions/removals are from VIP service upgrades in Backends, where the service owner issues a command to upgrade *all* the DIPs of the service to use the latest version of the service package and associated configurations.

All the other sources of DIP additions/removals account for less than 13% of the total updates because they affect only a handful of DIPs at a time. For example, testing is a special case of service upgrade and applied only in Backends, where the service owner restarts a subset of its DIPs to run the test version of the service package. Failure (e.g., lost control, application crash, etc.) or preempting (e.g., maintenance, resource contention, etc.) only triggers the restart or migration of the specific DIP (or a few DIPs if the physical machine is failed or preempted). Provisioning or removing is to add or delete a specific DIP in the DIP pool to adjust the service capacity according to traffic changes.

Why cannot we reduce the frequency of DIP pool updates? As we will discuss in §3.2, frequent DIP pool updates add new challenges to load balancing. So one question is if operators can reduce the number of DIP pool updates.

One way is to limit the update rate by delaying the execution of some updates. This smoothing approach may be feasible for some planned upgrades, but delaying updates can badly hurt application performance and reliability [24], especially for updates that swap out a faulty DIP, install critical security patches, or roll back a defective service version.

Another way is to reduce the number of updates by merging multiple DIP updates of the same VIP into a batch. This is not a good choice for both DIP removals and DIP additions. For DIP removals in service upgrades and testing, we need to ensure enough DIPs are up to provide enough service capacity at any time. Thus, the cluster scheduler typically uses the *rolling reboot* strategy, which reboots a fixed number of DIPs in every certain period (e.g., two DIPs every five minutes). For DIP additions, it takes different times for a DIP to come back alive (either finish the reboot or migrate to a new server). For example, for upgrades, the DIP downtime (from reboot to back alive) is 3 minutes in the median but 100 minutes in the 99th percentile as in Figure 4.

For the updates caused by failures, preemption, or DIP removal, we can remove all the related DIPs in a batch to prevent new connections from reaching these DIPs. However, since it takes a variety of time for these DIPs to come back (Figure 4), we have to handle DIP additions separately.

3.2 Problems of storing ConnTable in SLBs

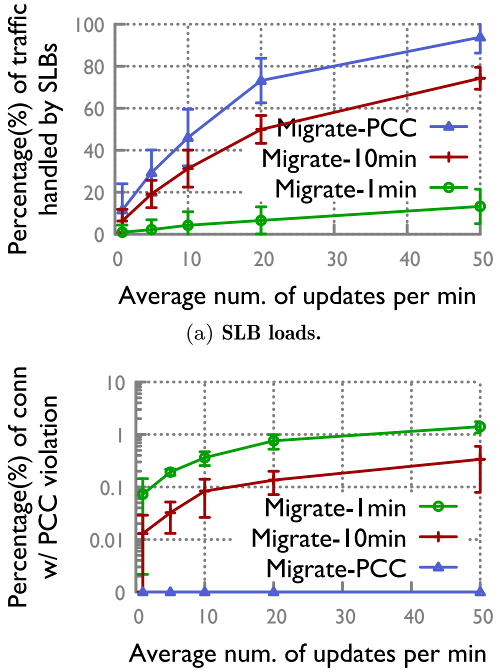
To ensure PCC during DIP pool updates, we rely on ConnTable to remember which DIP a connection mapped to. The SLB stores ConnTable at the server software but has performance limitations (§2.2). Another option as used in Duet [22] is to maintain VIPTable at switches but still leave ConnTable at SLBs. Thus, to update the DIP pool of a VIP, we need to redirect all the traffic of that VIP to SLBs to build up a ConnTable there to perform the DIP pool update.[2]

The main question is when to migrate the VIP from SLBs back to switches: If we migrate immediately, many remaining *old connections* (mapping to the old DIP pool) can get hashed to different DIPs under the new DIP pool at switches and violate PCC. If we migrate later, we need to handle more traffic in SLBs consuming more software resources. In addition, it is hard to find the right time to migrate all the VIPs with different DIP pool update timings. Hence, we can either periodically migrate VIPs to the switches as used in Duet [22], or wait until all the *old connections* have finished.

To illustrate the dilemma between PCC violations and SLB loads, we run a flow-level simulation using one-hour traffic traces collected from one PoP cluster with 149 VIPs. The cluster has an average of 18.7K new connections per minute per VIP and an average rate of 19.6 Mbps per VIP per top-of-rack (ToR) switch. We simulate different DIP pool update frequencies with an average of 1 to 50 updates per minute (as indicated in Figure 2). We simulate Hadoop traffic with a median flow duration of 10 seconds as in [39].

We evaluate three settings of storing ConnTable at SLBs: (1) *Migrate-10min*: periodically migrating VIPs back in every ten minutes as used in Duet; (2) *Migrate-1min*: migrating back in every minute; and (3) *Migrate-PCC*, where we wait until all the *old connections* have terminated before migrating the VIP to switches to ensure PCC.

The default Migrate-10min has a high SLB load. Figure 5a shows that, under 50 updates per minute, Migrate-10min handles 74.3% of the total traffic volume in SLBs. Note that, even coordinating with the service upgrade system does not help to decide when to migrate VIPs back, because it has to wait for the old connections to finish. As a result, this load in SLBs affects not only during the minute of bursty updates but also up to ten minutes until next VIP migration event. Worse still, operators have to provision a large number of SLBs all the time to handle the burst of updates because it is hard to instantiate SLBs and announce BGP paths fast enough to react to the burst. Migrate-10min also has a high SLB load for VIPs under *rolling reboot*. For example, a large VIP with hundreds of DIPs may take a couple of hours to upgrade all of its DIPs. Besides, the large VIP often has a large volume of traffic which leads to a high SLB load.



(a) **SLB loads.**

(b) **PCC violations.**

Figure 5: **The dilemma between SLB loads and PCC violations.**

To reduce SLB overhead, one may use Migrate-1min. In fact, Migrate-1min reduces the portion of traffic handled in SLBs down to 13.2% for 50 updates per minute. However, Migrate-1min causes more PCC violations than Migrate-10min. Figure 5b shows that Migrate-1min has 1.4% of connections broken for 50 updates per minute. Even for Migrate-10min, it has also 0.3% of connections with PCC violation.

PCC violations significantly degrade the tail latency for cloud services. Different from packet drops that can be recovered by TCP within sub-millisecond, for a broken connection, it often takes subseconds to seconds for applications to establish a new connection (e.g., one second in Wget). Broken connections also violate the service level agreements for users and affect the cloud revenue. Migrate-PCC avoids PCC violations, but it causes 93.8% traffic handled in SLBs for 50 updates per minute (Figure 5a).

To be conservative, our experiment is using Hadoop traffic with a short flow duration (a median of 10 seconds). For other traffic with longer flow durations, the number of PCC violations is much larger because there are more *old connections* when the SLBs migrate VIPs to switches. For example,

[2] Before the update, the SLB has to first wait long enough to ensure it sees at least one packet from each ongoing connections to have an entry for them in ConnTable. This is another problem of redirecting traffic between switches and SLBs.

we also simulate the cache traffic in [39] with a median flow duration of 4.5 minutes. Migrate-10min has 53.5% of total connections with PCC violation for 50 updates per minute.

In summary, the fundamental problem of storing ConnTable only in SLBs is that during DIP pool updates, the connections have to transfer between switches and SLBs. Instead, if we store ConnTable in the switches, we can avoid both the SLB overhead and the broken connections.

4 SILKROAD DESIGN

In this section, we present the SilkRoad design which implements both ConnTable and VIPTable on switching ASICs. Given the recent advances on ASICs with larger SRAMs (§4.1), it is now the right time to make the design choice of storing ConnTable in switches. Unlike the recent approach of maintaining VIPTable at switches [22], which still handles some packets of a connection at SLBs during DIP pool updates, SilkRoad ensures that *all* the packets of a connection are always handled at switches. In this way, we always gain the benefits of high-speed low-cost ASICs such as high throughput, low latency and jitter, and good performance isolation, while ensuring PCC.

We address two major challenges in SilkRoad: (1) To store millions of connections in ConnTable with tens of MB SRAM, we propose to store a hash digest of a connection rather than the actual 5-tuple to reduce the match field size, while eliminating the impact of false positives introduced by the digests. We also store a DIP pool version rather than the actual DIPs and allow version reuse to reduce the action field size. (§4.2) (2) To ensure PCC during frequent DIP updates we should handle the limitation of slow ConnTable insertion by a switch CPU. We use a small bloom filter to remember the new connections arrived during DIP pool updates, thus providing consistent DIP mappings for those connections in the hardware ASIC (§4.3).

In this section, for simplicity of presentation, we assume SilkRoad is deployed only at the top-of-rack (ToR) switches. We will discuss more flexible deployment scenarios in §5.

4.1 Features in commodity switching ASICs

Modern switching ASICs bring a significant growth in processing speed (up to 6.5 Tbps [1]) and also provide resources and primitives that enable us to implement PCC at large scale, via a careful co-design of switch data plane and control plane. Here we describe the notable characteristics of modern switching ASICs, which serve as enablers and also pose challenges to our design.

Increasing SRAM sizes: The SRAM size in ASICs has grown by five times over the past four years and reach 50-100 MB (Table 1), to meet the growing requirements to store a large number of L2/L3 forwarding and ACL entries. We later discuss this trend in more detail (§7). Existing fixed function ASICs often assign dedicated SRAM (or TCAM) blocks to each function. Emerging programmable ASICs [1, 5] allow network operators to flexibly assign memory blocks (from multiple physical stages) to user-defined match-action tables, which gives enough room to fit many connection states into on-chip SRAM with careful engineering.

ASIC generation	Year	SRAM (MB)
<1.6 Tbps [7, 9]	2012	10-20
3.2 Tbps[5, 8]	2014	30-60
6.4+ Tbps [1, 4, 13]	2016	50-100

Table 1: **Trend of SRAM size and switching capacity in ASICs. (SRAM does not include packet buffer; estimated based on table sizes claimed in whitepapers.)**

Connection learning and insertion: The key-value mapping semantics of ConnTable requires an exact matching table, which is typically implemented as a hash table on SRAM. Hash table implementations differ in their ways of handling hash collisions [34, 35]. Modern switching ASICs often take an approach known as cuckoo hashing that rearranges existing entries over a sequence of moves to resolve a collision [19, 35]. The cuckoo hash table provides a high packing ratio and memory efficiency but at the cost of running a complex search algorithm (breadth-first graph traversal) to find an empty slot. The time/space complexity of the algorithm is too high to run on switching ASICs at line rate. Hence, the entry insertion/deletion is the job of the software running on a switch management CPU.

Unlike routing table entries whose insertions are triggered by software routing protocols, the entry insertion into ConnTable is triggered by hardware: the event for the first packet of each connection hitting the ASIC. For this, we leverage the *learning filter* available in switching ASICs for L2 MAC learning. The learning filter usually batches a sequence of new events paired with additional metadata (e.g., mac-address-to-port mapping) and removes duplicate events. The CPU reads the arrival events from the filter and runs the cuckoo algorithm to insert new entries into the hardware table.

The slow connection learning and insertion time via CPU is not a problem for MAC learning because the frequency of new mac address or server/VM migration is relatively low. However, L4 load balancing needs to insert a new entry for every new L4 connection. This brings a new challenge for ensuring PCC which we will address in §4.3.

Transactional memory: Switching ASICs maintain an array of thousands of counters and meters [19] to collect various statistics and limit traffic rates. Tharray provides packet transactional semantics [19, 31]: the update on a counter by a previous packet can be immediately seen and modified by the right next packet, i.e., read-check-modify-write is done in one clock cycle time. P4 [18] exposes the generalized idea of transactional stateful processing as *register* arrays. By using the register primitive, we can implement a simple bloom filter and ensure PCC during DIP pool updates by remembering pending connections there.

4.2 Scaling to millions of connections

Motivation: millions of active connections: To understand the total number of *active* connections that we need to store in ConnTable, we take snapshots of ConnTable in all the SLBs every minute for each cluster. Figure 6 calculates the median

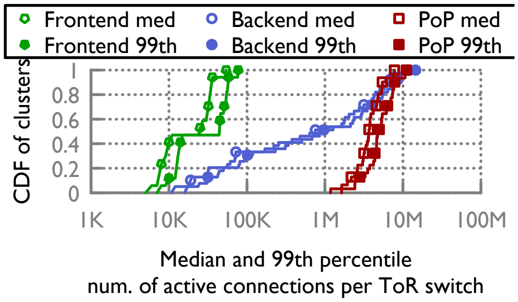


Figure 6: **Num. of active connections per ToR switch across clusters.**

and 99th percentile numbers of active connections normalized by the number of ToR switches in each cluster and draws the CDF across clusters. The 99th percentile number shows the worst-case ConnTable size we need to provision in each ToR switch if we deploy SilkRoad in the cluster. Among the PoPs and Backends, the most loaded clusters have around 10M connections. Frontends have fewer connections than PoPs because PoPs merge many user-facing TCP connections to a few persistent connections to Frontends.

It is challenging to store 10M connection states in a switch. For IPv6 connections, we need to store a 37-byte connection key (5-tuple) and an 18-byte action data (DIP and port) in each connection entry. This means we need at least 550 MB memory for ConnTable, which is far more than 50-100 MB SRAM available in switches today. Therefore, we use compact data structures to reduce the memory usage for both the match key and action data fields of each connection entry.

Compact connection match keys by hash digests: To reduce the size of the match field of each connection entry, we store a hash digest instead of the 5-tuple as proposed in [21]. For example, rather than storing 37 bytes of a 5-tuple of IPv6 connection, we only need a hash digest with 16 bits.

Using hash digests introduces a false positive if two connections hash to the same digest in the same hash location. When a new connection falsely hits at ConnTable, the connection uses the DIP assigned for the collided existing connection. Thus, it is unable to get its own DIP for the correct VIP due to the bypass of VIPTable. The good news is that the chance of false positives is low (0.01% of total connections using a 16-bit digest as shown in §6) and we can resolve them with a marginal software overhead.

As the first packet of a connection, TCP SYN matching on an existing entry is a good indication of a false positive. We redirect such a TCP SYN packet that matches an existing entry in ConnTable to the switch CPU. The switch software has complete 5-tuple information for each entry in ConnTable and thus can identify the existing entry that causes the false hit for the new connection.

The software resolves the hash collision by leveraging the multi-stage architecture of a match-action pipeline and relocating the existing colliding entry to another stage. In switching ASICs, a large exact-match table like ConnTable is instantiated on multiple physical stages, each storing a portion of ConnTable in the SRAM blocks of the stage [19, 27]. We can use a different set of hash functions for each stage. Hence, when a TCP SYN packet collides with an existing entry, the switch software migrates the existing entry to another stage. At that stage with a different hash function, these two connections are hashed to separate entry locations, resolving the hash collision.

After relocating the existing entry and inserting a new entry for the SYN packet, the software sends the SYN packet back to the switch, which then hits on the right entry. This adds a few milliseconds delay to the redirected TCP SYN packet. This marginal overhead is well justified by the high compression ratio (e.g., 2B/37B for IPv6 connections) and the low collision chance. Note, the SYN packet should trigger normal connection learning and is sent out immediately by ASIC if it does not falsely hit on ConnTable.

Compact action data with DIP pool versioning: To reduce the size of action data part of each connection entry, we map each connection to a *version of DIP pool* instead of actual DIP. When a DIP pool is updated, we actually create a new DIP pool by applying the change to a copy of the original DIP pool. We then assign a new version to the new pool and program VIPTable to map new incoming connections to the newest DIP pool version. Once a DIP pool is created and has active connections that still use it, the DIP pool never changes to provide consistent hashing to the active connections. A connection 'uses' a DIP pool if the connection arrives when the pool was the newest, and thus VIPTable maps the connection to the pool.

A DIP pool is destroyed when the connections that use it are timed-out and deleted from ConnTable. When the pool is destroyed, the version of the pool is also released and returned to a ring buffer so it can be reassigned to a newly created DIP pool. The switch software tracks the connection-to-pool mappings and manages DIP pool creation/deletion and as well as the ring buffer that stores available version numbers. From the large web service provider data, we observed 6-bit version number is big enough to handle many DIP pool update scenarios. Since most inbound connections are short-lived, each DIP pool and its version do not need to last for long. Using a 6-bit version field reduces the action data size to 1/24 in case the DIPs are IPv6 (16B IP + 2B port).

Since we introduce another level of indirection (pool version between connections and DIPs), we maintain the version-to-pool mappings in a new table called DIPPoolTable.[3] Figure 7 shows an example of our DIP selection design with DIPPoolTable. DIPPoolTable incurs an extra memory consumption to maintain a set of multiple (active) DIP pools for each VIP. The additional overhead is easily amortized by the savings from ConnTable when the number of connections mapped to each DIP pool version is large and they are short-lived, as observed from the web service provider data in §6. If the number of active connections is small and they are long-lived, we fall back to the design that maps each connection to the actual DIP instead of version.

[3] DIPPoolTable is similar to an ECMP table that maps ECMP group ID to a set of ECMP members (routing next hops).

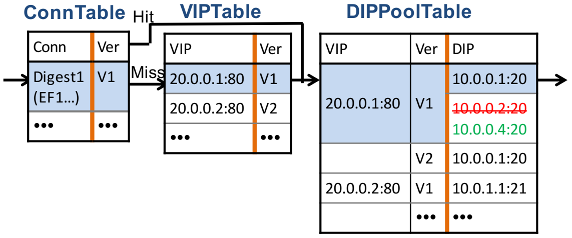


Figure 7: **Using digest and version in DIP selection.**

There are a few ways to further reduce the number of active versions, thus decreasing the size of version bits. One is modifying an existing DIP pool and reusing it as the current/newest pool when it is possible, instead of blindly creating a new DIP pool and assigning a new version. It is possible when a new DIP is added to substitute a previously removed DIP in the pool, which is usually the case of *rolling reboot* discussed in §3.1. For example, in Figure 7, VIP 20.0.0.1:80 has two DIPs in version $V1$. When DIP 10.0.0.2:20 has failed, we remove it from the VIP and create another DIP pool version $V2$. Existing connections to 10.0.0.1:20 still use $V1$ to ensure PCC and new connections use $V2$ to ensure no more new connections towards the failed DIP 10.0.0.2:20. When we add a new DIP 10.0.0.4:20 to the VIP, instead of creating a new DIP pool version, we reuse the old $V1$ and replace DIP 10.0.0.2:20 with 10.0.0.4:20. Now new connections use $V1$ to select the new DIP. [4]

4.3 Ensuring per-connection consistency

Motivation: many new connections during DIP pool update: In SilkRoad, for the first packet of each new connection, the ASIC selects a DIP and sends out the packet immediately. In the meanwhile, the ASIC notifies software for entry insertion into ConnTable. Given a short RTT in data centers (250 μs in the median and 2-5 μs with RDMA), a new connection can have many packets arrived before the software completes entry insertion into ConnTable. When overlapped with a DIP pool update, the subsequent packets of the connection can be mapped to a different DIP, violating PCC.

To better understand the PCC problem, we define *pending connections* at time t as those connections that arrive before t but have not been inserted in ConnTable. We cannot apply a DIP pool update when there are pending connections because the first few packets of these connections already match the old DIP pool, and if ConnTable is not ready, the follow-up packets would match the new DIP pool. Therefore, *we can only safely apply a DIP pool update only when there is no single pending connection.* However, this may never happen. Suppose at time t_1, we have a set of pending connections C_{t_1}, and the switch software inserts entries for these connections at ConnTable at time t_2. Between t_1 and t_2, other new connections can arrive, which become new pending connections C_{t_2}. This can go on forever if there are continuous new connection arrivals. The number of pending connections during a DIP pool update depends on two factors: the new connections arrival rate and the time the switch takes to learn and insert new connection entries in ConnTable. To quantify the arrival rate of new connections, we measured the number of new connections per minute per VIP for all clusters. Figure 8 shows that a VIP can have more than 50M new connection arrivals in a minute.

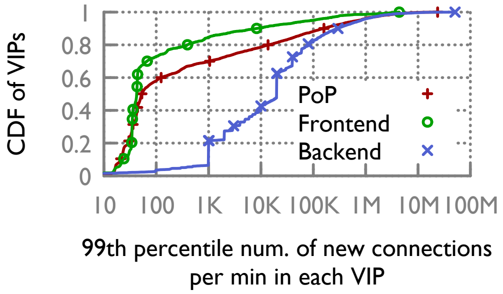


Figure 8: **Number of new connections per VIP in one minute.**

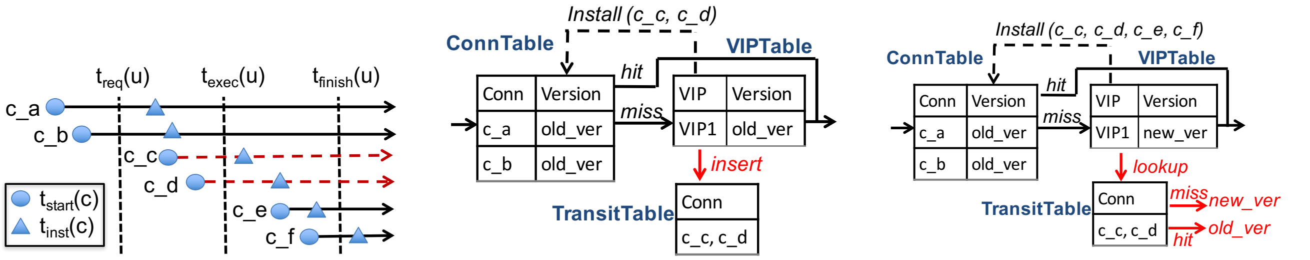
The connection learning and insertion time depends on the ASIC design. As discussed in §4.1, ASICs often batch new connection events in a *learning filter* to avoid frequent interruptions to the switch CPU. The filter also removes duplicate events (from multiple packets of the same connection). The learning filter can store up to thousands of requests and notifies the switch software when the learning filter is full or after a timeout. The timeout value can be set by the network operator, and we expect anywhere between 500 μs to 5 ms. The switch software then reads the new connection events in batches, run the cuckoo hashing algorithm to select empty slots, and inserts the entries in ConnTable.

Suppose there are consistent arrivals of 1M new connection every minute for a VIP at one switch. Whenever we want to update the DIP pool for the VIP, there is always around 8 new connections in the learning filter with 500 μs timeout setting, leaving no time window to perform the DIP pool update with PCC.

3-step PCC update with TransitTable: To be able to update a DIP pool without violating PCC, we need to ensure the switching ASIC can handle the pending connections correctly. We introduce a *TransitTable* that remembers the set of pending connections that should be mapped to an old DIP pool version when VIPTable updates its DIP pool version. In this way, no pending connection is left out: it is either pinned in ConnTable or marked in TransitTable.

We use the transactional memory primitive in switching ASIC (§4.1) to implement TransitTable as a bloom filter. Bloom filter indices are addressed by a number of hashes, and unlike a cuckoo-based exact matching table, the hash collision between different connections is allowed. Hence, bloom filter does not need CPU to run a complex cuckoo algorithm and can do read-check-modify-write in one cycle time, providing the packet transactional semantics. Collisions in all hash indices lead to a false positive, which is kept negligible as long as the filter size is large enough to handle the number of connections. Thus, the main challenge is the

[4] There may have a very rare chance that we use out all the versions because of a few long-lasting connections. We can move those long-lived connections to a small table to fall back to connection-to-DIP mapping.



(a) **Timeline of connections. (c_c and c_d are pending connections at $t_{exec}(u)$)**

(b) **Step 1: between $t_{req}(u)$ and $t_{exec}(u)$**

(c) **Step 2: between $t_{exec}(u)$ and $t_{finish}(u)$**

Figure 9: **TransitTable to ensure PCC during DIP pool updates (c_a–c_f are connection digests).**

size of bloom filter (TransitTable), which can grow as large as ConnTable if designed poorly.

A naive design is to always store every new connection sent to each VIP upon its arrival in TransitTable and keep a record of both the connection and its selected DIP pool version. This allows immediate execution of DIP pool updates for any VIP but requires TransitTable to be large enough to remember all new connection states.

To reduce the memory usage, we only consider the connections to the VIP currently under DIP pool update and only store the pending connections that are mapped to the old DIP pool. The key insight is that there are just two versions of DIP pool involved during an update for a given VIP: the old version before the update and the new version after the update. Thus, we reduce a key-value store problem (the connection to DIP version mapping) to a simple membership set problem. TransitTable only needs to remember the set of connections that are mapped to the old version, where a binary Bloom filter can do in a memory-efficient way.

We take the following 3-step update process to ensure PCC as in Figure 9a, where $t_{start}(c)$ and $t_{inst}(c)$ indicate the time of connection arrival and the time of insertion into ConnTable respectively. **Step 1:** In Figure 9b, when the switch receives a request of DIP pool update ($t_{req}(u)$), we start to remember all the new connections in TransitTable bloom filter. **Step 2:** When all the connections that arrive before $t_{req}(u)$ get inserted in ConnTable, we stop updating the bloom filter and execute the update($t_{exec}(u)$) on VIPTable. After the update, all the packets that miss ConnTable retrieve both old and new versions from VIPTable and then are checked by TransitTable to see if the packets hit the bloom filter. If hit, they use the old version; if miss they use the new version. Note that the bloom filter is read-only in this step while it was write-only in the first step (Figure 9c). **Step 3:** Once all the connections in TransitTable get inserted in ConnTable, we clear TransitTable and finish the process ($t_{finish}(u)$).

Note that using a Bloom filter for TransitTable can cause false positives when a new connection arrives between $t_{exec}(u)$ and $t_{finish}(u)$ (i.e., Step 2 in Figure 9c) falsely matches on TransitTable and take the old version. The chance of false positives is low (with 256-byte bloom filter, no false positive observed in one hour under most frequent updates as shown in §6). To handle it, the ASIC redirects any TCP SYN packet matching on TransitTable at Step 2 to the switch software similar to the solution to the digest collisions problem in ConnTable.

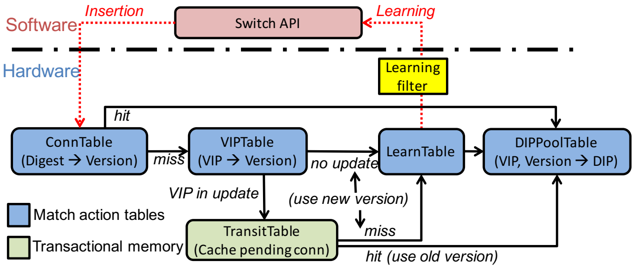


Figure 10: **System architecture.**

Figure 10 shows an overall architecture of SilkRoad, depicting the control flow between the various tables. A simple table (LearnTable) is added to trigger new connection learnings to the switch software.

5 IMPLEMENTATION AND DEPLOYMENT

In this section, we talk about the details of implementing the SilkRoad prototype on a programmable ASIC. We then discuss the prototype performance and evaluate its implementation overhead in terms of hardware resource consumption and software overhead. We also discuss using SilkRoad in a network-wide setting.

5.1 Prototype implementation on high-speed commodity ASICs

We built a P4 prototype of SilkRoad on top of a baseline switch.p4 and compiled on a programmable switching ASIC [1]. The baseline switch.p4 implements various networking features needed for typical cloud data centers (L2/L3/ACL/QoS/...) in about 5000 lines of P4 code. A simplified version of the baseline switch.p4 is open-sourced at [16]. We added ~400 lines of P4 code that implements all the tables and metadata needed for SilkRoad (Figure 10). More details of our prototype are demonstrated in [32].

We implement all the tables as exact-match tables, except for TransitTable as a bloom filter on transactional memory. ASICs often support word packing which allows efficiently matching against multiple words in the SRAM block at a time [19]. We carefully design word packing to maximize the memory efficiency while minimizing false positives [27].

We also implement a control plane in switch software that handles new connection events from learning filter and connection expiration events from ConnTable. The software runs the cuckoo hash algorithm to insert or delete connection entries in ConnTable. Besides, the control plane performs 3-step PCC update for DIP pool updates. The event and update handler is written in about 1000 lines of C code while entry insertion/deletion is part of switch driver software.

5.2 Prototype performance and overhead

Performance: Our prototype shows that SilkRoad achieves full line-rate load balancing with sub-microsecond processing latency. Note that in a pipeline architecture of most switching ASICs today, adding any new logic into the pipeline does not really change the bit/packet processing throughput of a switch as long as the logic fits into the pipeline resource constraints. Switch pipeline latency may slightly increase by up to tens of nanoseconds, which is negligible in end-to-end datacenter latency and three to five orders of magnitude smaller than SLB processing latency [20, 22].

In addition, SilkRoad achieves tighter performance isolation than that in SLBs because it handles all traffic completely in hardware. To throttle a VIP under DDoS attacks or flash crowds, SilkRoad associates a meter (rate-limiter) to a VIP to detect and drop excessive traffic. A meter is marking packets to one of the three colors defined by two rate thresholds [6]. To measure metering accuracy, we generated 10 Gbps traffic to a VIP and measured color marking accuracy with various rate thresholds and burst size settings, and observed less than 1% average error. Creating 40K meter instances consumes 1% of the entire SRAM in the ASIC, providing performance isolations for many VIPs.

ASIC resource consumption: We evaluate the additional resources that SilkRoad needs on top of the baseline switch.p4 mentioned before. Table 2 shows the additional hardware resources used by SilkRoad while storing 1M connections (with 16-bit digest and 6-bit version) compared to the switch.p4. We see that the additional resource usage is less than 50% for all types of hardware resources. The exact-match tables together increase the usage of SRAM and match crossbars, while ConnTable is the major consumer. The VLIW (very long instruction word) actions are used for packet modifications [19]. The hash operations in the exact matching tables and the multi-way hash addressing of bloom filter consume additional hash bits. The bloom filter implementation uses stateful ALUs to perform transactional read-check-update, as meters/counters do in the baseline switch.p4. Our P4 prototype defines a few metadata fields to carry DIP pool version and other information between the tables (Figure 10). The metadata fields consume a negligible amount of PHV (Packet Header Vector) bits [19]. We have also evaluated that up to 10M connections can fit in the on-chip SRAM in our SilkRoad prototype.

Software overhead: The switch employs a standard embedded x86 CPU that connects to the switching ASIC via PCI-E interface. For each new connection, the switch software sends a sequence of moves to the ASIC to make an empty slot for the new entry in ConnTable. The switching ASIC makes sure the execution of these moves does not affect the ongoing traffic matching ConnTable.

Resource	Additional usage
Match Crossbar	37.53%
SRAM	27.92%
TCAM	0%
VLIW Actions	18.89%
Hash Bits	34.17%
Stateful ALUs	44.44%
Packet Header Vector	0.98%

Table 2: **Additional H/W resources used by SilkRoad with 1M connection entries, normalized by the usage of the baseline switch.p4.**

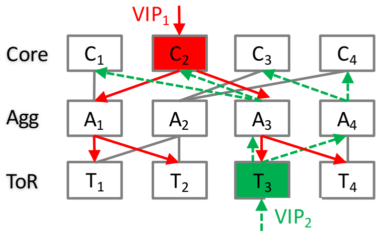


Figure 11: **Network-wide VIP assignment to different layers.**

We measure the connection insertion rate to understand the switch software overhead in managing SilkRoad. In our software prototype using single-core, we found the bottleneck is on the CPU, not the PCI-E interface. Hash computations for cuckoo hashing and connection digest take most of the CPU time while cuckoo search algorithm took the second largest but relatively small time. The CPU overhead increases as ConnTable utilization approaches close to 100%. We expect SilkRoad can achieve ConnTable insertion throughput of 200K connections per second by employing 1) better software library for hash computation and 2) multiple cores to handle insertions into different physical pipes.

5.3 Network-wide deployment

A simple deployment scenario is to deploy SilkRoad at all the ToR switches and core switches. Each switch announces routes for *all* the VIPs with itself as the next hop. In this way, all inbound and intra-DC traffic gets the load balancing function at its first hop switch into the data center network. Intra-DC traffic reaches the load balancing function at the ToR switch where the traffic is originated. Inbound internet traffic gets split to multiple core switches via ECMP and gets load balanced there. In this design, we can easily scale the load balancing function with the size of the data centers.

However, this design is unable to efficiently handle network-wide load imbalance. And the network operator may want to limit the SRAM budget for load balancing function at specific switches. To address this, rather than blindly serving a VIP traffic at the first hop switch in the network, we can decide which layer (e.g., ToR, aggregation, and core) to handle a specific VIP and thus split traffic across multiple switches.

Figure 11 shows a simple example. If inbound traffic to VIP1 cannot meet SRAM budget at core switch C_2, we then

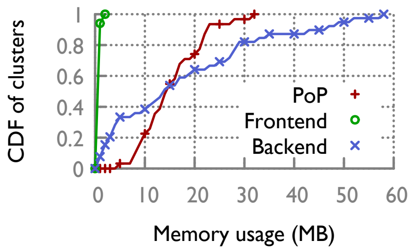


Figure 12: **SRAM usage in SilkRoad deployed on ToR switches across clusters.**

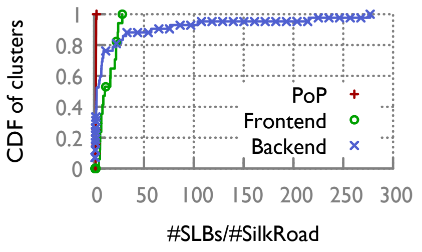


Figure 13: **The ratio of #SLBs and #SilkRoad to support load balancing across clusters.**

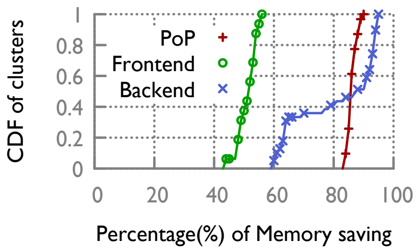


Figure 14: **Memory saving for SilkRoad deployed on ToR switches across clusters .**

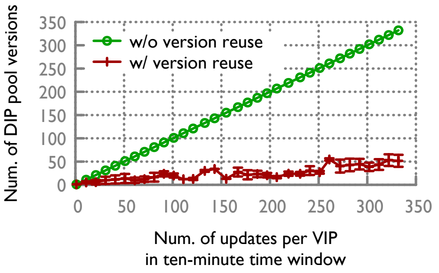


Figure 15: **Benefit of version reuse.**

migrate VIP1 from all core switches (only show C_2 here) to ToR switches. The traffic is balanced via ECMP to the switches at ToR layer, where these switches together have enough memory to handle a large number of connections. Similarly, if a ToR switch T_3 experiences a burst of intra-DC traffic for VIP_2 from its rack servers, SilkRoad can migrate VIP_2 to the multiple core switches.

The adaptive VIP assignment problem can be formulated as a bin-packing problem. The input includes network topology, the list of VIPs, and the traffic for each VIP. The traffic consists of traffic volume and number of active connections. The objective is to find the VIP-to-layer assignment that minimizes the maximum SRAM utilization across switches while not exceeding the forwarding capacity and SRAM budget at each switch. This can also preserve SRAM headroom for operators to expand service capacity or handle failures.

We can extend our design to do *incremental deployment*: where the operator may deploy SilkRoad on a subset of switches or add some new SilkRoad-enabled switches to the network. Our bin-packing algorithm still works to assign VIPs to different layers so as to fit in all switches' memory. The only difference is the traffic for a VIP is then split to only SilkRoad-enabled switches in the assigned layer instead of all switches in that layer.

6 EVALUATION

We build a flow-level simulator to evaluate the memory usage and PCC guarantee of SilkRoad using real traffic traces from the large web service provider we studied. We simulate the SilkRoad on all ToR switches in each cluster. We show that our design achieves 40%-95% SRAM reduction and thus can fit into switch SRAM to support traffic for all clusters. Using the available SRAM in switches, one SilkRoad can replace up to hundreds of SLBs. We show that processing the same amount of traffic in SilkRoad has about two orders of magnitude saving in power and capital cost compared with SLBs. Our design ensures PCC with only a 256-byte bloom filter even under most frequent DIP pool updates observed from the network.

6.1 Scalability

We first evaluate how SilkRoad can scale to cloud-scale traffic using the traffic traces from each of PoPs, Frontends, and Backends during its peak hour of a day. We replay the traffic traces to measure the memory usage on each ToR switch. Most Backends use IPv6 addresses while most PoPs and Frontends use IPv4 addresses. Throughout the simulations, we consider the SRAM word of 112 bits as used in [19]. We configure ConnTable entry as 28 bits with a 16-bit digest, a 6-bit version number, and a 6-bit overhead.[5] In this way, we exactly pack four ConnTable entries in each SRAM word.

SilkRoad can fit millions of connections into the switch SRAM: Figure 12 shows the switch memory usage of SilkRoad for each cluster. In PoPs, the SilkRoad uses 14 MB in a median cluster and 32 MB in a peak cluster. The SilkRoad in Backends has a median of 15 MB and a peak of 58 MB memory usage. Backends have larger SRAM consumption in the peak cluster than PoPs because the peak Backend cluster has more connections (up to 15M) than PoP (up to 11M). SilkRoad in Frontends consumes less than 2 MB SRAM because Frontends has a small number of connections (see Figure 6). Therefore, SilkRoad can fit into ASIC SRAM with 50-100 MB (Table 1).

We investigate the breakdown of ConnTable and DIPPoolTable usage. Take the peak Backend cluster as an example. ConnTable consumes 91.7% of total 58MB memory usage to store up to 15M connections. The DIPPoolTable takes the rest to host 64 versions of 4187 IPv6 DIPs.

SilkRoad can significantly reduce the number of SLBs: The key benefit of SilkRoad is that we can reduce the number SLBs by providing high throughput and low latency load balancing. SilkRoad requires memory resources to store connections at switches while SLBs require CPU resources to process packets at hosts. To understand the tradeoff, we take the peak throughput and the peak number of connections

[5] The overhead bits include an instruction address and a next table address.

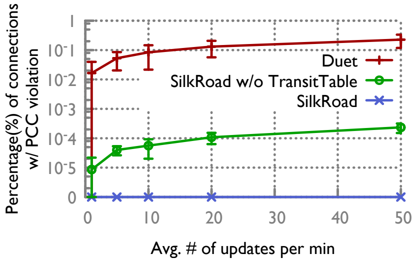


Figure 16: **Effectiveness of ensuring PCC with various update frequencies.**

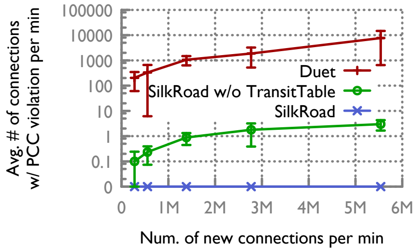


Figure 17: **Impact of new connection arrival rates (10 updates per minute, SilkRoad TransitTable=256B).**

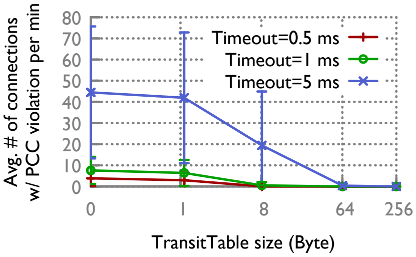


Figure 18: **TransitTable size (10 updates per minute).**

of a day in each cluster and estimate the number of SLBs and SilkRoad switches we need to support load balancing function. We assume that each SilkRoad can handle 10M connections. For SLB we use the state-of-the-art performance 12 Mpps for 52-byte packets using 8 cores reported by Google [20]. The results are shown in Figure 13. For PoPs where most traffic is short user-facing connections, we need 2-3 times more SLBs compared to SilkRoad. Frontends can replace 11 SLBs with one SilkRoad in the median because Frontends receive a small number of persistent connections with large volume from PoPs. In Backends, one SilkRoad can replace 3 SLBs in the median cluster and 277 SLBs in the peak cluster. The need for a large number of SLBs in some peak Backends is because connections there are typically volume-centric traffic across services (e.g., storage) and the prevalent use of persistent connections for low latency. Generally, it is more suitable to deploy SilkRoad in those clusters with more volume-centric traffic.

SilkRoad saves the cost and power for running load balancing as well. To support the state-of-the-art performance of 12 Mpps for 52-byte packets, a typical SLB with Intel Xeon Processor E5-2660 costs around 200 Watt and 3K USD [10, 20]. By contrast, SilkRoad with 6.4Tbps ASIC can achieve about 10 Gpps with 52-byte packets, consuming around 300 Watt and 10K USD [8]. So processing the same amount of traffic in ASIC consumes about 1/500 of the power and 1/250 of the capital cost compared to SLBs.

Using digest and version save memory: SilkRoad reduces the memory usage of ConnTable by using digest for the key field and DIP pool versions for the value field. Figure 14 quantifies the memory savings of this design. All the clusters have more than 40% of memory reduction through using digest with or without version. PoPs have a consistent memory reduction by around 85% from using both digest and version. Frontends have around 50% memory saving from only using digest. Backends receive 60%-95% of memory saving.

Version reuse: To quantify the benefit of reusing the versions (see §4.2), we consider all the VIPs in Backends. For each ten-minute time window, we count the number of DIP pool versions before and after version reuse mechanism. Here, we choose ten minutes as time window to cover the lifetime for most of the connections [39]. Figure 15 shows that a VIP can have up to 330 DIP pool updates in ten minutes and thus need 330 versions and 9 version bits. With version reuse, we only need to use 6 version bits to handle up to 51 DIP pool versions. Consider a cluster with 10M connections and 4K DIPs, reducing the number of bits for versions can reduce ConnTable by 7.5 MB and the DIPPoolTable by 4.5 MB, with a total of 74.6% memory reduction.

Tradeoffs of digest sizes and false positives: To understand the false positives, we evaluate the memory and false positives for one PoP cluster with 2.77M new connections per minute per ToR switch. If we use 32 MB SRAM with 16-bit digest, there are an average of 270 (0.01%) false positives per minute. If we use 42.8 MB SRAM with 24-bit digest, we have 1.1 (0.00004%) false positives per minute. All false positives are resolved via switch software with no PCC violation (§4.2).

6.2 Ensuring PCC

Now we evaluate the effectiveness and overhead to ensure PCC across different solutions. We conduct experiments on following scenarios: *(1) Duet*: Duet design which migrates VIPs back to switches every ten minutes; *(2) SilkRoad without TransitTable*: SilkRoad without using TransitTable to ensure PCC; *(3) SilkRoad*: SilkRoad with 3-step PCC update process using TransitTable.

We use a one-hour traffic trace from one PoP cluster as introduced in §3.2. The trace consists of 149 VIPs and has a peak of 2.77M new connections per minute per ToR switch. We generate traffic and updates independently because we intend to evaluate the range of changes, instead of a specific combination in our dataset. By default, we use learning filter size of 2K insertions with 1 ms timeout and TransitTable size of 256 bytes. The software insertion rate is set to 200K entries per second as we discussed in §5.2.

SilkRoad ensures PCC for various DIP pool update frequencies: Figure 16 shows SilkRoad always ensures PCC with 256-byte TransitTable even under the most frequent updates. For 10 updates per minute, Duet incurs PCC violations in 0.08% of total connections and SilkRoad without TransitTable breaks 0.00005% of total connections. SilkRoad without TransitTable has about three orders of magnitude fewer PCC violations than Duet because the DIP pool update in SilkRoad affects only new connections during their insertion period (a few milliseconds). In contrast, Duet affects existing connections (running for seconds to minutes) when it migrates VIPs back to switches.

SilkRoad ensures PCC for various new connection arrival rates: We now vary the arrival rate of the number of new connections by scaling the traffic of 2.77M new connections per minutes by a factor of 0.1 to 2. Figure 17 shows the average number of connections with PCC violation per minute across traffic intensities. SilkRoad with 256-byte TansitTable has no PCC violation. With the connection arrival rate increases, SilkRoad without TansitTable has more PCC violations because there are an increasing number of *pending connections* in the learning filter. Duet also has increasing PCC violations because with more new connections arrive, there are more *old connections* at SLBs when we migrate the VIP back to switches. **SilkRoad ensures PCC using small TransitTable:** Figure 18 shows that SilkRoad requires a small size of TransitTable to ensure PCC during a DIP pool update. For example, during the simulation period of one hour, TransitTable with only 8 bytes prevents PCC violation for learning filter timeout within 1 ms. With a larger timeout of 5ms, there are 20 connections with PCC violations with just 8-byte TransitTable and no violations with 256-byte TransitTable. This should be easily supported because today's ASICs already have the transactional memory for thousands of counters.

7 DISCUSSION

How much memory in ASICs can we use for load balancing? We expect in the future the SRAM size in switching ASICs will continue to grow. This is because there is a strong requirement of a large memory for building various functions for diverse markets, such as storing 100Ks of Internet IP prefixes in ISP edge routers, maintaining a large MAC table and a large access control list (ACL) in enterprise switches, and storing MPLS labels in backbone switches. However, the memory requirements for data center switches are relatively small compared to other markets [22]. This is because data centers use simple ECMP-based routing and push access control rules to host hypervisors. We expect a fair amount of memory in switching ASICs available for the load balancing function or for offloading other connection-tracking functionalities hosted in traditional middleboxes or hypervisor.

In addition, traditional fixed function ASICs often waste memory space because they have dedicated tables for each function. Emerging programmable switching ASICs [1, 5] allow network operators to use memory flexibly, which leaves more room for load balancing functions. Moreover, we can use the feature of multiple stages in the pipeline to further optimize the tradeoffs between memory usage and false positives. For example, we can use different digest sizes in different stages to reduce the overall false positives. When there is a small number of connections, we insert new connections to stages with larger digest sizes (i.e., low false positives). When the number of connections increases, we use stages with smaller digest sizes to scale up.

Handle DIP failures: To detect DIP failures, each SilkRoad can perform the health check on DIPs. Today's switches already have health check for BGP sessions. Many switches today offer an ability to offload BFD (Bidirectional Forwarding Detection) [2]. This mechanism can be leveraged to perform a fast health check. To perform the health check for 10K DIPs in every 10 seconds with 100-byte packets [12], switches only need around 800 Kbps bandwidth.

After we find a DIP failed or unreachable, SilkRoad switch quickly removes the DIP from the DIP pool. To reduce the number of DIP pool versions, we can continue to use the same DIP pool version and use resilient hashing [3] to maintain existing connections to other DIPs. This is an alternative way for version reuse.

Handle switch failures: If a SilkRoad switch fails, the existing connections on this switch get redirected to other switches via ECMP and get load balanced there because all the switches use the same latest VIPTable. Thus if a connection was using the latest version of VIPTable at the failed switch, it would get the same VIPTable at the new switch and thus ensure PCC. However, since we lose the ConnTable at the failed switch, those connections that used an old DIP pool version may break PCC. This is the same issue with an SLB failure in the software load balancing case.

Combine with SLB solutions: We propose SilkRoad as a new primitive to implement load balancing in switches for better performance. In practice, operators can choose to use SilkRoad only or combine it with SLBs to best meet their traffic scenarios. For example, when ConnTable in SilkRoad is full, SilkRoad can redirect extra connections to either the switch software or SLBs (basically treating SilkRoad ConnTable as a cache of connections). Or we can use SilkRoad to handle VIPs with high traffic volume and use SLBs to handle those VIPs with a large number of connections. We can enable this hybrid setting by withdrawing those VIPs from switches and announcing them from SLBs via BGP. Different from Duet [22], we do not need to migrate VIPs during DIP pool updates and always ensure PCC.

8 RELATED WORK

Load balancing: Beyond SLBs [15, 20, 30, 36] and Duet [22], there are other works [26, 28, 41] which use OpenFlow switches to implement load balancing. They either leverage the controller to install flow rules based on incoming packets which are too slow due to the slow switch-controller channel, or pre-install wildcard rules that are hard to change during traffic dynamics. Instead, SilkRoad supports line-rate packet processing during traffic dynamics and DIP pool updates.

Consistent updates: The paper [38] introduced per-packet and per-flow consistency abstractions for updates on a network of switches. These inconsistency problems are caused by different rule update rates across switches. The recent load balancer paper [20] leverages consistent hashing to ensure that all SLBs select DIPs in the same way when DIP pool changes and ECMP rehashing happen at the same time. Instead, SilkRoad focuses on per-connection consistency (PCC) for updating VIPTable and ConnTable inside a single switch. PCC problem is caused by the slow insertion time of switch software. Thus, we introduce a TransitTable that stores pending connections to ensure PCC.

Programmable ASICs: Recently, due to more control requirements of switch internals from major cloud providers (e.g., [11, 40]), switch vendors (e.g., Cavium [5], Barefoot [1], Intel [9]) start to expose low-level hardware primitives of high-speed low-cost ASICs to customers. Recent research works such as reconfigurable match-action tables (RMT) [19] and Protocol-Independent Switch Architecture (PISA) [18, 27] are built on those primitives. SilkRoad focuses on the load balancing function, which is critical for data centers, and leverages existing features of ASICs. Thus SilkRoad can be either implemented with either small modifications of fixed function ASICs or built directly on top of programmable ASICs in the market.

9 CONCLUSION

L4 load balancing is a critical function for data centers but becomes increasingly challenging to build with the growth of traffic and the constantly changing data centers. To address these challenges, SilkRoad leverages the increasing SRAM sizes in today's ASICs and stores per-connection states at ASICs. In this way, SilkRoad inherits all the benefits of high-speed low-cost ASICs such as high throughput, low latency and jitter, and better performance isolation, while ensuring per-connection consistency during DIP pool changes, as demonstrated by our extensive simulations and a P4 prototype on a programmable ASIC.

ACKNOWLEDGMENTS

We thank our shepherd Teemu Koponen, the anonymous SIGCOMM reviewers, Patrick Bosshart, Remy Chang, Ed Doe, Blake Matheny, Vincent Maugé, Nick McKeown, Xuehai Qian, Dileep Rao, and Nikita Shirokov for their valuable feedbacks. This research is partially supported by CNS-1712674, CNS-1701923, CNS-1413978, and Facebook.

REFERENCES

[1] Barefoot Tofino: programmable switch series up to 6.5Tbps. https://barefootnetworks.com/media/white_papers/Barefoot-Worlds-Fastest-Most-Programmable-Networks.pdf

[2] Bidirectional Forwarding Detection (BFD). https://tools.ietf.org/html/rfc5880

[3] Broadcom Smart-Hash technology. https://goo.gl/LXtq16

[4] The Broadcom StrataXGS BCM56970 Tomahawk II Switch Series. https://goo.gl/a9vCgo

[5] Cavium XPliant™ Ethernet Switch Product Family. http://www.cavium.com/XPliant-Ethernet-Switch-Product-Family.html

[6] A Differentiated Service Two-Rate, Three-Color Marker with Efficient Handling of in-Profile Traffic. https://tools.ietf.org/html/rfc4115

[7] High Capacity StrataXGS®Trident II Ethernet Switch Series. https://www.broadcom.com/products/ethernet-connectivity/switch-fabric/bcm56850/

[8] High-Density 25/100 Gigabit Ethernet StrataXGS Tomahawk Ethernet Switch Series. https://www.broadcom.com/products/ethernet-connectivity/switch-fabric/bcm56960

[9] Intel FlexPipe. https://goo.gl/kUqpU7

[10] Intel Product Specifications. http://ark.intel.com/

[11] Introducing data center fabric, the next-generation Facebook data center network. https://goo.gl/makVDo

[12] Load-Balancer-as-a-Service configuration options. https://docs.openstack.org/ocata/config-reference/networking/lbaas.html

[13] Mellanox Spectrum™ Ethernet Switch. https://goo.gl/SsVXcM

[14] Nginx. https://nginx.org/en/

[15] NSX Distributed Load Balancing. https://goo.gl/GWcJMT

[16] Open-source P4 implementation of features typical of an advanced L2/L3 switch. https://github.com/p4lang/switch

[17] M. Alizadeh, et al. 2013. pFabric: Minimal Near-optimal Datacenter Transport. In *ACM SIGCOMM '13*.

[18] P. Bosshart, et al. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* (2014).

[19] P. Bosshart, et al. 2013. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *ACM SIGCOMM Computer Communication Review*.

[20] D. E. Eisenbud, et al. 2016. Maglev: A Fast and Reliable Software Network Load Balancer. In *NSDI*.

[21] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher. Cuckoo Filter: Practically Better Than Bloom. In *ACM CoNEXT 2014*.

[22] R. Gandhi, et al. Duet: Cloud scale load balancing with hardware and software. In *Proceedings of the 2014 ACM conference on SIGCOMM*.

[23] P. X. Gao, et al. 2016. Network Requirements for Resource Disaggregation. In *USENIX OSDI*.

[24] R. Govindan, et al. Evolve or Die: High-Availability Design Principles Drawn from Googles Network Infrastructure. In *ACM SIGCOMM 2016*.

[25] C. Guo, et al. 2015. Pingmesh: A large-scale system for data center network latency measurement and analysis. *ACM SIGCOMM Computer Communication Review* (2015).

[26] N. Handigol, et al. 2009. Plug-n-Serve: Load-balancing web traffic using OpenFlow. *ACM SIGCOMM Demo* (2009).

[27] L. Jose, L. Yan, G. Varghese, and N. McKeown. Compiling packet programs to reconfigurable switches. In *NSDI 15*.

[28] N. Kang, et al. 2015. Efficient Traffic Splitting on Commodity Switches. In *ACM CoNEXT*.

[29] A. Kaufmann, et al. 2016. High performance packet processing with flexnic. In *ACM SIGPLAN Notices*. ACM.

[30] T. Koponen, et al. 2014. Network Virtualization in Multi-tenant Datacenters.. In *NSDI*.

[31] L. Lamport. 1985. *Interprocess Communication.* Technical Report. DTIC Document.

[32] J. Lee, et al. 2017. Stateful Layer-4 Load Balancing in Switching ASICs. In *ACM SIGCOMM demo*.

[33] B. Li, et al. 2016. Clicknp: Highly flexible and high-performance network processing with reconfigurable hardware. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*. ACM.

[34] M. Mitzenmacher. 2001. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems* (2001).

[35] R. Pagh and F. F. Rodler. 2004. Cuckoo hashing. *Journal of Algorithms* (2004).

[36] P. Patel, et al. 2013. Ananta: Cloud scale load balancing. In *ACM SIGCOMM Computer Communication Review*.

[37] S. Radhakrishnan, et al. 2014. SENIC: scalable NIC for end-host rate limiting. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*.

[38] M. Reitblatt, et al. Abstractions for network update *(SIGCOMM '12)*. ACM.

[39] A. Roy, et al. Inside the Social Network's (Datacenter) Network *(SIGCOMM '15)*. ACM.

[40] A. Singh, et al. 2015. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *ACM SIGCOMM '15*.

[41] R. Wang, D. Butnariu, and J. Rexford. 2011. OpenFlow-based server load balancing gone wild. Hot-ICE.

[42] Y. Zhu, et al. 2015. Congestion control for large-scale RDMA deployments. In *ACM SIGCOMM Computer Communication Review*.