SlimeMold: Hardware Load Balancer at Scale in Datacenter

Ziyuan Liu[1,2], Zhixiong Niu[2], Ran Shu[2], Liang Gao[3], Guohong Lai[3], Na Wang[4], Zongying He[4]
Jacob Nelson[2], Dan R. K. Ports[2], Lihua Yuan[5], Peng Cheng[2], Yongqiang Xiong[2]
[1]Beihang University SKLSDE, [2]Microsoft Research, [3]Ragile Networks Inc., [4]Broadcom Inc., [5]Microsoft

ABSTRACT

Stateful load balancers (LB) are essential services in cloud data centers, playing a crucial role in enhancing the availability and capacity of applications. Numerous studies have proposed methods to improve the throughput, connections per second, and concurrent flows of single LBs. For instance, with the advancement of programmable switches, hardware-based load balancers (HLB) have become mainstream due to their high efficiency. However, programmable switches still face the issue of limited registers and table entries, preventing them from fully meeting the performance requirements of data centers. In this paper, rather than solely focusing on enhancing individual HLBs, we introduce SlimeMold, which enables HLBs to work collaboratively at scale as an integrated LB system in data centers.

First, we design a novel HLB building block capable of achieving load balancing and exchanging states with other building blocks in the data plane. Next, we decouple forwarding and state operations, organizing the states using our proposed 2-level mapping mechanism. Finally, we optimize the system with flow caching and table entry balancing. We implement a real HLB building block using the Broadcom 56788 SmartToR chip, which attains line rate for state read and >1M OPS for flow write operations. Our simulation demonstrates full scalability in large-scale experiments, supporting 454 million concurrent flows with 512 state-hosting building blocks.

CCS CONCEPTS

• **Networks** → **Data center networks**; **Network management**.

KEYWORDS

Load Balancing, Programmable Switches

ACM Reference Format:
Ziyuan Liu[1,2], Zhixiong Niu[2], Ran Shu[2], Liang Gao[3], Guohong Lai[3], Na Wang[4], Zongying He[4] and Jacob Nelson[2], Dan R. K. Ports[2], Lihua Yuan[5], Peng Cheng[2], Yongqiang Xiong[2]. 2023. SlimeMold: Hardware Load Balancer at Scale in Datacenter. In *7th Asia-Pacific Workshop on Networking (APNET 2023), June 29–30, 2023, Hong Kong, China.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3600061.3600067

1 INTRODUCTION

In cloud data centers, layer-4 load balancers are critical to scale-out services which distribute application traffic to backend servers by mapping virtual IPs (VIPs) to direct IPs (DIPs) [17]. They help improve the service capacity and reliability of applications. Stateful load balancers are widely used in production, which keep the load balancing decisions for each connection and ensure the consistency of connections when adding or removing backend servers [12–14, 17, 19]. As data center applications' scale and capacity increase, applications require larger-scale load balancers with greater bandwidth, higher connection per second (CPS), and larger total concurrent connections. The required bandwidth can reach up to O(Tbps), the maximum concurrent flows can be hundreds of millions, and the CPS can reach tens of millions [1].

In recent years, software-based load balancer (SLB) has been widely used by the industry due to its agility and reliability [12, 17]. However, the SLB can incur significant costs for data centers due to its inefficiency. For instance, a single server of Maglev can only saturate a 10Gbps link [12] and it requires hundreds of servers to support data center scale traffic. Programmable switches have emerged as a highly promising solution due to their programmable capabilities and high throughput density, enabling them to handle orders of magnitude higher traffic while reducing costs. Recent works have attempted to accelerate these software LBs with new programmable switches [11, 13, 16, 19].

Despite this, the programmable switch's ability to support CPS and concurrent connections remains restricted in comparison to its overall throughput, as its primary purpose is to process and forward packets with limited functionality using stateful objects. There are typically about 10MB [20] on-chip tables and registers to store states. For example, a switch can only support up to 200K concurrent flows if a table entry takes 50B [19]. Such capacity is far less than the requirement of layer-4 load balancers which is hundreds of millions. As a consequence, data center operators still need multiple nodes to scale out. However, existing scale-out solutions use ECMP to distribute traffic to different nodes and it will cause Per-Connection Consistency violations due to network changes or changes in the load balancer pool [12]. PCC violation means that a flow served by a backend server is changed to another one, which breaks the session context as the new backend does not have the corresponding state of the flow.

To address this issue, we propose SlimeMold, a solution to solve capacity problems in hardware load balancers. Our approach differs from existing ones by advocating for collaborative HLBs for entire HLB performance, rather than solely focusing on improving individual LB. In this architecture, each LB on a programmable switch serves as a building block of the entire HLB. As such, the total capacity of the load balancer is proportional to the number of building blocks in our design. In other words, our goal is to maintain simplicity in the HLB rather than striving for peak performance in a single HLB unit. Instead, we are taking a flexible approach by composing HLBs within the network as a whole to achieve maximum performance, allowing for collaborative work by utilizing the

APNET 2023, June 29–30, 2023, Hong Kong, China

ACM ISBN 979-8-4007-0782-7/23/06…$15.00
https://doi.org/10.1145/3600061.3600067

resources of other building blocks through our design in scheduling and communication. As a result, with the increasing deployment of programmable switches in data centers, the performance of the existing large-scale HLB can be effortlessly improved.

To achieve the SlimeMold, we first present a new HLB paradigm that serves as a fundamental building block of the entire HLB system. To this end, we have chosen the Ragile Smart Switch with the Broadcom 56788 SmartToR chip [2], which is designed for next-gen ToR switches. It can support millions of flow entries [3] and support Network Programming Language (NPL) [6]. NPL is a high-level language for high-performance, feature-rich networking platforms. We deploy the building block by replacing the existing switches in the topology. Given the high market share of Broadcom chips in data centers, it is possible to use other existing switches that support NPL programming but have fewer table entries as additional building blocks. Apart from providing basic LB features like packet rewrite, encapsulation, and state learning, we have also designed and implemented a methodology for building blocks to create and fetch states remotely from another building block via the data plane, thus enabling each building block to work collaboratively.

Secondly, to distribute the HLB at scale, we have decoupled the role of an HLB and designed a two-level mapping mechanism to organize all building block connection tables as a unified resource pool. The two mechanisms separate the functions of a stateful load balancer into forwarding and state management and also overcome the capacity limitations of a single building block by distributing the substantial load of an entire LB system across multiple building blocks. Each building block can simultaneously serve as a Forwarder, State Owner, and Secondary Lookup. The Forwarder broadcasts the VIPs and forwards packets to the State Owner. The State Owner maintains the mapping state between VIP and DIP for each flow and can forward packets to their DIP. The Secondary Lookup serves as an intermediate point to build a two-level lookup structure which reduces the size of flow to State Owner table.

Finally, to further optimize the system, we have designed two mechanisms, ConnTable cache on Forwarders and flow table balancing on State Owners, aimed at minimizing traffic between building blocks and maximizing resource utilization within each block.

To summarize, the main contributions of our work are:

- We design and implement a real building block with Broadcom 56788 SmartToR chip, which can serve as a basic load balancer and also achieve flow states write and fetch remotely.
- We have accomplished a scalable design for building blocks by separating load balancing into forwarding and state management and designed a two-level state owner mapping mechanism to effectively organize flow states.
- Additionally, we introduce cache and flow table balancing mechanisms to further optimize efficiency.
- We have conducted a real evaluation of the building block. Its performance can reach more than 1 million operations per second (MOps) for state writing and line rate for state reading via the data plane. Additionally, we have simulated the entire system for large-scale topology. The results demonstrate that it is fully scalable, and when using 512 State Owners, it can support 454 million concurrent flows.

2 BACKGROUND AND RELATED WORK

Most production load balancers are stateful load balancers [4, 12, 17]. Compared to stateless load balancers, stateful load balancers can achieve Per-connection Consistency (PCC) by utilizing Connection Table (ConnTable), which stores the flow-to-server mappings and can support server churn naturally. PCC is a crucial requirement for load balancers, as it ensures that all packets from a flow are delivered to the same server since only that server can correctly handle the connection once it is established.

The workflow of stateful LBs can be divided into two steps: DIP decision, and forwarding. In the DIP decision process, when a flow reaches an LB to access a VIP, the LB assigns a real server which is the DIP to the flow based on specific policies (e.g., hash or round-robin) and inserts an entry for the flow into the ConnTable. In the forwarding process, the LB forwards all packets of a flow according to the DIP recorded in its ConnTable entry to maintain PCC.

Modern LBs are predominantly implemented in software [4, 12, 17], which offers flexibility to accommodate the numerous emerging features demanded by data center operators. However, software-based load balancers also contribute to increased operational expenses, which is undesirable for data centers. To mitigate this cost, recent works have explored leveraging switches to accelerate stateful LBs. Duet [14] and Rubik [13] enhance traffic scalability by offloading VIPs with stable DIP pools onto switches, using software-based LBs as backups to handle complex corner cases. SilkRoad [16] employs a programmable switch to fully offload the LB logic onto hardware, further improving scalability and DIP update frequency. Tiara [19] combines programmable switches, FPGAs, and servers to create a hybrid load balancer solution boasting high bandwidth, a large ConnTable, and increased CPS.

As single node performance of both SLB and HLB still cannot meet the overall requirement of data center LB service, data center operators need scale-out LBs. However, scaling out stateful LB while keeping PCC is non-trivial. Existing scale-out load balancer solutions use ECMP to distribute flow table entries among multiple LB nodes. However, as the system scale changes, ECMP may direct a flow to an LB node that does not own its flow table entry thus causing PCC violation. Resilient ECMP [5] is proposed to mitigate the issue and it can direct a flow to the same LB node provided that it is not affected when removing an LB node. However, when adding an LB node, Resilient ECMP can still direct a flow to a different LB node. In this case, though Consistent Hashing can be applied to increase the chance to assign the same DIP to a flow, PCC is still not guaranteed. This issue is well recognized by researchers [12, 17].

Distinct from the aforementioned works, SlimeMold focuses on utilizing simple HLB units and breaking the boundaries of individual HLBs, enabling HLBs within cloud data centers to collaborate and function as a unified, large-scale HLB providing services for the data center while preserving PCC. It can be used as a stand alone load balancer if the available hardware capacity can support data center load balancer requirements, or it can be used as an enhancement to existing load balancer systems. Consequently, as more programmable switches, particularly those with SmartToR chips that function as programmable ToR switches, are deployed in data centers, they can seamlessly enhance the performance of the existing large-scale HLB.

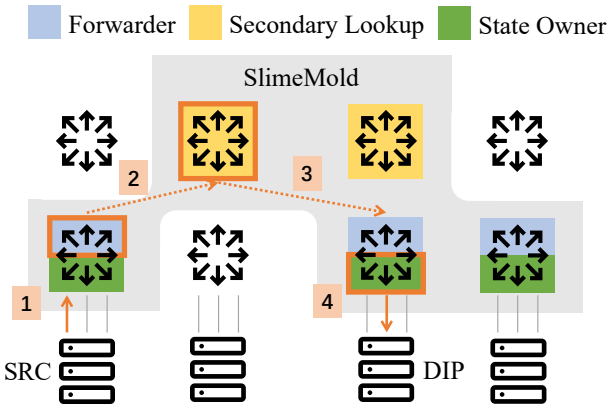

Figure 1: SlimeMold Overview

3 DESIGN

We present the design of SlimeMold in this section. In our discussion, we will address the following questions:

- What is the structure of a large-scale distributed HLB at the data center scale? (§3.1)
- How can we make the distributed HLBs a giant LB? (§3.2)
- How do we design the building block to meet the requirements of a large-scale HLB? (§3.3)
- How can such a large-scale system be further optimized (§3.4, and §3.5)

We also discuss the remained open questions of SlimeMold briefly in §3.6

3.1 Overview

In stateful layer-4 LB systems, forwarding is more resource-consuming because LB needs to look up ConnTable and modify packets using NAT/tunneling for every packet. While the DIP decision only happens at the first packet of a flow, the computation and memory overhead are both small compared to forwarding. In SlimeMold, we focus on scaling out forwarding only. Implementing the DIP decision on each switch node or connecting x86 server to each node like Tiara [19] are two possible solutions, but not all possible solutions. We leave it for future work.

To build a distributed and dynamic load-balancer system while preserving PCC, in SlimeMold, we rethink the functionalities of a load balancer and separate it into two roles: Forwarder and State Owner. Briefly, to a flow, a Forwarder stores a table that can forward any flow to the State Owner that holds the flow's state. Therefore, no matter which Forwarder is reached by a flow, or how we dynamically distribute flow states, a flow can always get its ConnTable entry and thus preserve PCC.

For each Forwarder, it should know which State Owner hosts a flow's states. Direct map between flows and State Owners is meaningless since storing such a table requires the same number of table entries as storing all flow states. Thus we should arrange flows into groups with simple and stateless calculations, and then store mapping between groups of flows and State Owners. We call such a group a segment. A segment is also the unit that we can use to balance the load among State Owners. A large data center would have up to $O(10K)$ switches [18]. Moreover, the number of segments should be large enough (e.g., 10x) to enable fine-grained load adjusting between State Owners. The State Owner table on each Forwarder should have at least $O(100K)$ entries which is a burden to Forwarder table resource. Thus we introduce a two-level table to reduce the required table size and put the second-level table on another logical node called Secondary Lookup.

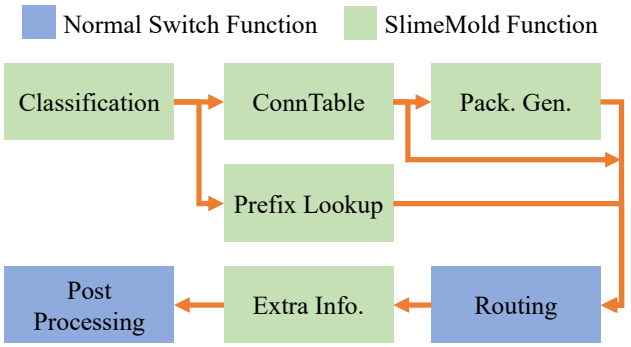

Figure 2: SlimeMold Building Block

An example workflow is shown in Figure 1. (1) A packet of a flow from the source first reaches a Forwarder (the blue box with orange border) by data center routing and enters SlimeMold system. (2) As the Forwarder does not have the flow's ConnTable entry, it directs the packet to a Secondary Lookup. (3) Then Secondary Lookup forwards the packet to its State Owner (the green box with orange border). (4) As the State Owner has the flow's entry, it directs the packet to the DIP and the packet exits SlimeMold. The PCC is guaranteed because no matter at which Forwarder the flow's packets enter SlimeMold, they can always be forwarded to its State Owner to get the corresponding DIP.

3.2 2-Level State Owner Mapping

To group flows into segments, we need simple stateless calculations. Address prefix is the easiest way to divide flows into segments. However, the flow address in a data center has non-uniform distribution, simply using prefix causes serious load imbalance between State Owners. Thus, we use hash to map a flow to a group. We use CRC32 to hash the 5-tuple of flows and use the prefix of the hash result to distinguish a flow belongs to which segment. We define the prefix as the first p bits of the hash result. Thus the number of segments is 2^p. Please note that by this design, the number of segments should be a power of 2.

To divide the State Owner mapping into two levels, we divide the prefix into two parts, prefix 1 (the first p_1 bits of the prefix) and prefix 2 (the remaining p_2 bits of the prefix). Prefix 1 is used on Forwarder to find the Secondary Lookup. Prefix 2 is for Secondary Lookup to find the State Owner. We call the mapping table from prefix 1 to Secondary Lookup address Secondary Lookup table. Similarly, Secondary Lookup uses State Owner table to map prefix 2 to State Owner. Please note that we do not carry the 5-tuple hash in the packet, thus Secondary Lookup needs to recalculate it to get prefix 2.

3.3 Single Building Block

A single building block is a logically basic unit that can be configured as a specific role (Forwarder, Secondary Lookup, and State

Owner) in SlimeMold. In this section, we describe all functionalities required by different SlimeMold roles and summarize what a SlimeMold building blocks should support. The pipeline of SlimeMold building block's functions are shown in Figure 2. Specifically, the functions required are: 1) packet classification: distinguish different types of packets in SlimeMold; 2) ConnTable; 3) packet generation: generate an extra packet to other roles (e.g., cache refresh packet from Forwarder to State Owner, and cache update packet from State Owner to Forwarder); 4) prefix lookup; 5) routing; 6) extra information addition; 7) packet post-processing (e.g., NAT/tunneling to DIP). When assigned one or some specific roles, the pipeline may be simplified if some features are not used.

ConnTable. There are four operations required by ConnTable management: insert, query, delete, and aging. Insert, query, and delete of a specific ConnTable entry are triggered by packets which means they are data plane operations. Aging is an operation to prevent dead entries due to abnormal flow close. It removes a ConnTable entry after an aging period if no queries access it. We design it as another operation other than delete since it needs to closely rely on switch firmware to manage the timer for each entry. The ability to manage ConnTable is mandatory for State Owner and optional for Forwarder (as ConnTable cache, see §3.4). A physical switch can serve as a State Owner and a Forwarder simultaneously. We let them share the ConnTable to improve efficiency. We divide ConnTable entries into two types: normal entry and cache entry. A normal entry can swap out a cache entry, but not vice versa. Such table sharing is an optional feature and should be only enabled if a physical switch plays as both State Owner and Forwarder.

Prefix Lookup. It is required by Forwarder and Secondary Lookup because they need to find other roles in SlimeMold based on flow hash. A single building block should be able to use a key obtained by part of the packets to perform a lookup whose result may be one or a set of addresses of the next logical hop. The building block should be able to use the address returned by the lookup to perform routing as a normal switch.

Carry Extra Information. A single building block should be able to modify packets to append/remove additional headers which carry extra information to the next logical hop. The ability to append/remove additional headers is required by Forwarder and State Owner because they need to carry SlimeMold's internal information (e.g., packet type, whether to update the cache of a Forwarder, and the address of the Forwarder) to other roles.

3.4 ConnTable Cache

Instead of going directly to the DIP from a load balancer node in traditional systems, a flow will be detoured in SlimeMold. Specifically, a flow will be transmitted from a Forwarder to its State Owner to query its ConnTable entry, which will increase the traffic volume in the network and may lead to congestion. This also adds network latency which may downgrade application performance, especially for those latency-intensive applications. To alleviate the problem, we add a ConnTable cache on each Forwarder: If a flow finds its entry in Forwarder, it will go to DIP directly; Otherwise, it will go to State Owner to get its DIP.

To avoid unnecessary traffic, in SlimeMold, we let the cache be managed by Forwarder. If the cache misses for a packet on a Forwarder, it will send the packet to State Owner with a mark that indicates if it needs to get the flow state to update its cache. To update the cache, State Owner should send an additional packet carrying the ConnTable entry back to the Forwarder. We use this design instead of letting the original packet piggyback states to avoid extra packet detour and possible out-of-order caused by the detour.

According to switches' limit programmability, they can only implement simple cache policies i.e., direct-mapped cache (associates a flow to one specific cache line using hash). Hash collision is easy to happen in a direct-mapped cache. To avoid frequent cache update which causes flow path changes, we do not allow cache eviction on hash collision. This means that a flow will be admitted into the cache only if the corresponding entry is empty. Our results in §4.2.1 show this simple cache policy already has good performance gain.

Load balancers usually delete a flow table entry if no traffic uses it for a fixed period which is called aging. To invalidate an entry in the cache, a Forwarder uses the same aging timer value as State Owner. This causes an issue in our design: if a flow hits a Forwarder's cache for a duration longer than the aging threshold, its State Owner will delete the entry since no packet arrives from this flow. Thus in SlimeMold, a Forwarder should refresh State Owner before the aging timer expires if a flow entry is cached. We let Forwarder send an extra packet to a flow's State Owner to refresh the aging timer of this flow on State Owner. To avoid failed refresh caused by network latency and packet loss, the refresh timer is set to 1/4 of the aging timer to balance robust considerations and network overhead. As the typical aging threshold is usually seconds level, this won't add much network traffic overhead.

3.5 Table Entry Balance

A flow's ConnTable entry is managed by a specific State Owner according to its flow hash. Because of different flow sizes and live time, the load of State Owners may vary, in terms of traffic volume served and the number of concurrent ConnTable entries. Without load balancing between State Owners, resource utilization will be affected since the system load depends on the switch with the maximum load.

Simply moving existing entries to a new switch and then modifying the 2-level State Owner mapping will cause PCC violation during the moving process. SlimeMold introduces an elegant segment migration scheme to balance load among State Owners. Specifically, we partition the segment transfer into three phases: mirroring, modification, and recycling. In the mirroring phase, the queries to a transferred segment are still served by the original State Owner and mirrored to the new State Owner. To achieve this, SlimeMold controller configures the original State Owner to forward packets that will change a ConnTable entry (e.g., a packet may trigger the creation or deletion of an entry, refresh the aging timer) to the new State Owner. After at least one complete aging period, all valid entries have been mirrored to the new State Owner and the execution goes to the modification phase. In this phase, SlimeMold controller first lets the original State Owner give the ownership of this segment by forwarding all packets belonging to this segment to the new State Owner instead of mirroring operations. Then, it modifies the segment's State Owner in all related Secondary Lookup's State

Owner table. After the State Owner mapping update, migration enters the last phase where SlimeMold controller stops the forwarding from the original State Owner to the new State Owner by deleting the related table entry. SlimeMold controller also recycles the related table entries on the original State Owner.

SlimeMold controller collects load metrics from all switches it managed and uses them to balance table entry. Choosing load metrics and adjusting algorithms are out of this paper's scope. Here we give a simple load-adjusting algorithm as a baseline. The load adjusting algorithm has a parameter called load imbalance threshold r and will adjust loads based on State Owners' ConnTable utilization ratio. If the algorithm finds that the ratio of the minimal to maximal ConnTable usage among all State Owners is less than r, it will transfer one segment from the most utilized State Owner to the least utilized State Owner. The process will be repeated several times until the ratio of the minimal to maximal is greater than r. Because the maximal ConnTable usage should not exceed a State Owner's physical capacity, the peak resource utilization which can be achieved by SlimeMold is average/maximal, i.e. $(1 + r)/2$.

3.6 Open Questions

There are still open questions we have not solved in the current SlimeMold design. Here we list and briefly discuss them.

Role Placement. Current SlimeMold design only contains the logical relation between roles but does not include how to place them physically in a data center. Different placements may influence SlimeMold's performance as the number of physical hops between two nodes differs. A good placement algorithm should minimize packet detour caused by SlimeMold.

Routing between Roles. Different roles in SlimeMold may not be physically connected in the data center. We need to route packets to the destination role through the existing data center network. How to work with the underlying network architecture efficiently remains a question.

Decide the Number of Segments. A large number of segments can provide fine-grained ConnTable balance but causes high prefix lookup table overhead. We need further investigations on the proper number of segments.

Table Entry Balance Algorithms. SlimeMold provides a framework and is open to using advanced algorithms to balance the load between State Owners. How to design such an algorithm and what is the potential trade-off is one of our future works.

Heterogeneous Switches. Data center may use different types of switches, and available resources on different types or even the same type of switch may differ. How to make SlimeMold be aware of heterogeneous switches and consider them when taking actions requires further investigations.

Failure Detection and Handling. As a distributed system, single point of failure and network failure will cause serious problems. How to detect failures? How to avoid state loss if State Owner fails? How to route packets to avoid failure nodes and links? These questions are valuable to consider to make SlimeMold robust.

DIP Decision. Current SlimeMold only focuses on scale-out of forwarding in load-balancer. Efficiently scaling out DIP decision and efficient interactions between these two components are still open questions.

	Throughtput	P99 lat.	CT entries
SlimeMold BB	8Tbps	< 2us	1M

Table 1: Performance of SlimeMold Building Block

	Query	Insert	Delete
OPS	line rate	1.485M	~ 0.6M
Latency	< 2us	167ns	< 140ms

Table 2: Performance of ConnTable Operations

4 EVALUATION

We build an Ethernet switch using Broadcom SmartToR [8] programmable switch chip which is an 8T switch device with support of 25G, 100G, 200G, and 400G ports. We implement basic features of SlimeMold including Classification, ConnTable, Prefix Lookup, and Carrying Extra Information on our testbed. We use Spirent FX3-100GQ-T2 test module [7] to evaluate the prototype's characteristics.

We also implement a flow-level simulator in C++ for large-scale simulation. We use Fat-Tree topology [9] with size parameter K set to 32 (8192 servers, and 512, 512, 256 for ToR, Aggregation, and Core switches, respectively). We use two widely used realistic workloads web search [10] and data mining [15] as the size distribution of generated traffic. We set the number of VIPs as 10x of the number of servers, and randomly choose them from 172.0.0.1 to 172.255.255.255. We assign server IP sequentially from 10.0.0.1, and the sender and DIP are chosen from all servers uniformly. By default, SlimeMold uses all ToR and Aggregation switches in the topology. Each ToR switch acts as both a Forwarder and a State Owner. Each Aggregation switch acts as a Secondary Lookup. The number of available CAM table entries in each switch is 1.5 million. 2/3 of a switch's available table resource is intended for State Owner's ConnTable, and the rest 1/3 is intended for Forwarder's cache. As the number of segments is only up to 8192 ($> 10 \times 512$), we omit the table resource overhead of 2-level State Owner mapping in the calculation. The flow hash function is CRC32. We use the example table entry balance algorithm described in §3.5, and the imbalance ratio r is set to 0.8.

4.1 Building Block Performance

We measure the performance of a SlimeMold building block using traffic generated by multiple Spirent test modules. One Spirent test module is connected to two ports and sends bidirectional traffic to test the bandwidth and latency between these ports. We first insert 50% flow entries to ConnTable. Then we generate 1500B sized packets whose destination address is in the ConnTable and measure the performance. The results are shown in Table 1. The throughput reaches 8Tbps which is the line rate of the test switch. The latency has little variation but all results are less than 2us. We also show the total ConnTable capacity we support in the table which is 1 million.

We also measure each ConnTable operation performance and list the results in Table 2. Because we have not implemented aging, we do not evaluate it. We use the same method as bandwidth to test query performance under various packet sizes. Its OPS can reach the theoretical bound and latency is up to 2us. To measure insert

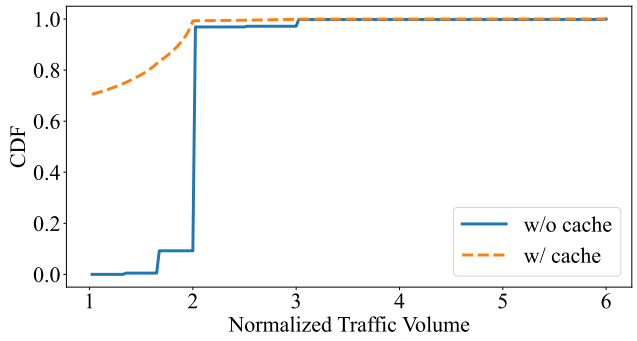

Figure 3: Normalized traffic volume distribution

OPS, we program the chip to record the first insert time and last time during inserting 0.1M, 0.2M, . . . , 1M entries into ConnTable. To measure the latency of an insert, we program the chip to set a bit in the packet to one if ConnTable hits and set it to zero otherwise, and measure the time difference between the first sent packet and the first packet with the bit set on the receiver. The method to measure deletion is similar. The insert speed is much higher than control plane based method as mentioned in Silkroad [16] as all hash computation and hash table management are done in hardware provided by Broadcom SmartToR chip [2]. Unlike the insert operation, the delete operation speed shows a slightly increasing trend ([0.587M, 0.612M]) if there are more used entries in the table. The latency of deletion is longer as we implement it in batch due to the hardware's characteristics. Specifically, a delete operation will set a bit in a bitmap to indicate that an entry should be deleted, and we use the R5 ARM core on the chip to scan the bitmap periodically and execute deletions. Because the latency variation is large due to batch, we only provide an upper bound. This long deletion latency makes the table occupancy higher than ideal, thus may affect total CPS and the number of concurrent flows. We leave the detailed impact to future evaluation.

4.2 Large Scale Simulation

4.2.1 Extra Traffic Overhead. As SlimeMold routes a flow passes Forwarder and State Owner (if cache misses) which may not be on the optimal path between the source and DIP, the traffic volume in the data center network is enlarged as the hops of a flow increase. Here we define the traffic volume of a flow as the number of hops multiples the total bytes in a flow. Please note that a flow may change its path due to a cache hit or miss. The actual calculation is more complex. Let b_i denote each byte in a flow and n_i denote the number of hops b_i passes. The traffic volume of this flow is $\sum n_i$.

To show how much extra traffic SlimeMold will add to the data center network, we conduct a simulation and get the number of hops each flow passes. We use the web search traffic pattern to generate flows. We normalize the traffic volume of each flow as its optimal path and plot the CDF in Figure 3. From the result, we can see that without cache, about 90% flows will have at least 2x hops compared with the optimal. While, even with the simplest cache policy, over 70% flows can be cached and go through the shortest path between the source and DIP.

4.2.2 Table Entry Balance. In this experiment, we show SlimeMold can effectively avoid hot spots which can improve the overall table utilization. We measure the imbalance index (defined as the maximum load divided by the average load of the State Owner table) of SlimeMold switches with and without the table entry balance algorithm under two traces. The results are shown in Table 3. Without table entry balance, the imbalance index is over 1.2. If we enable table entry balance, the imbalance index drops to about 1.13. As we set the imbalance ratio r to 0.8, the average load among all switches is expected to be about 0.9 of the max load, and the ideal imbalance index should be about $1/0.9 = 1.11$. In conclusion, SlimeMold table entry balance algorithm can effectively balance table entries close to the target.

	w/o Balance	w/ Balance
Web Search	1.2	1.13
Data Mining	1.22	1.128

Table 3: Imbalance (max load / avg load) before / after applying table entry balance algorithm

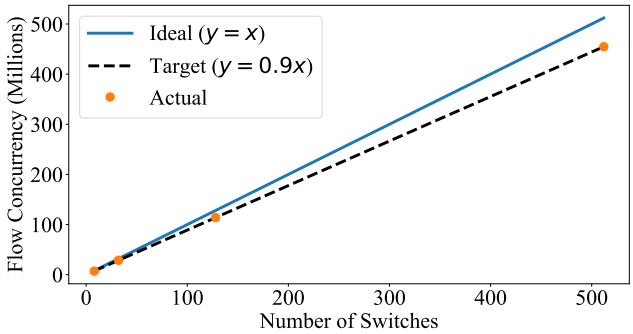

Figure 4: Scalability

4.2.3 Scalability. In this experiment, we run simulations to evaluate SlimeMold's scalability. We vary the size parameter $K = 4, 8, 16, 32$ of Fat-Tree topology (having 8, 32, 128, 512 ToR/Aggregation switches, respectively), and adjust the traffic load to achieve the maximum flow concurrency supported by SlimeMold while not exceeding any State Owner's physical capacity. The results are shown in Figure 4. We can see that SlimeMold can scale linearly as the number of switches in the system increases, and can achieve about 0.89 flow state table utilization under the setting. The resource utilization is quite close to the theoretical value which is $(1 + 0.8)/2 = 0.9$.

5 CONCLUSION

In this paper, we have presented SlimeMold, a scalable hardware load balancer for data centers. Our approach distinguishes itself from existing ones by promoting collaborative HLBs to enhance overall HLB performance, rather than focusing solely on improving individual LBs. In this architecture, each LB on a programmable switch serves as a building block for the entire HLB. Consequently, the total capacity of the load balancer is proportional to the number of building blocks in our design. We implement a real HLB building block using the Broadcom 56788 SmartToR chip, which achieves line rate for state read and >1M OPS for flow write operations. In large-scale experiments, our simulation demonstrates full scalability, supporting 454 million concurrent flows with 512 State Owner nodes.

REFERENCES

[1] 2018. Unveiling the Networks behind the 2018 Double 11 Global Shopping Festival. https://www.alibabacloud.com/blog/594167?spm=a2c5t.11065265.1996646101.searchclickresult.289b2f0575gg5Z.

[2] 2023. BCM56780 Series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56780.

[3] 2023. Broadcom Breaks New Ground with Trident SmartToR, Converging Switching, Routing, and L4-L7 Services. https://investors.broadcom.com/news-releases/news-release-details/broadcom-breaks-new-ground-trident-smarttor-converging-switching.

[4] 2023. DPVS is a high performance Layer-4 load balancer based on DPDK. https://github.com/iqiyi/dpvs.

[5] 2023. Equal Cost Multipath Load Sharing - Hardware ECMP. https://docs.nvidia.com/networking-ethernet-software/cumulus-linux-43/Layer-3/Routing/Equal-Cost-Multipath-Load-Sharing-Hardware-ECMP/.

[6] 2023. NPL – Open, High-Level language for developing feature-rich solutions for programmable networking platforms. https://nplang.org/.

[7] 2023. Spirent FX3 2-Port Quint-Speed QSFP28 Modules. https://www.spirent.com/assets/u/spirent_fx3_hse_module_datasheet.

[8] 2023. Trident SmartToR. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/smarttor.

[9] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. 2008. A scalable, commodity data center network architecture. *ACM SIGCOMM computer communication review* 38, 4 (2008), 63–74.

[10] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data center tcp (dctcp). In *ACM SIGCOMM (2010)*.

[11] Tom Barbette, Chen Tang, Haoran Yao, Dejan Kostić, Gerald Q Maguire Jr, Panagiotis Papadimitratos, and Marco Chiesa. 2020. A high-speed load-balancer design with guaranteed per-connection-consistency. In *USENIX NSDI (2020)*.

[12] Daniel E Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. 2016. Maglev: A fast and reliable software network load balancer. In *USENIX NSDI (2016)*.

[13] Rohan Gandhi, Y Charlie Hu, Cheng-kok Koh, Hongqiang Harry Liu, and Ming Zhang. 2015. Rubik: Unlocking the power of locality and end-point flexibility in cloud scale load balancing. In *USENIX ATC (2015)*.

[14] Rohan Gandhi, Hongqiang Harry Liu, Y Charlie Hu, Guohan Lu, Jitendra Padhye, Lihua Yuan, and Ming Zhang. 2014. Duet: Cloud scale load balancing with hardware and software. *ACM SIGCOMM Computer Communication Review* (2014).

[15] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. 2009. VL2: A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*. 51–62.

[16] 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 *ACM SIGCOMM (2017)*.

[17] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, et al. 2013. Ananta: Cloud scale load balancing. *ACM SIGCOMM Comput. Commun. Rev* 43, 4 (2013), 207–218.

[18] Cheng Tan, Ze Jin, Chuanxiong Guo, Tianrong Zhang, Haitao Wu, Karl Deng, Dongming Bi, and Dong Xiang. 2019. NetBouncer: Active Device and Link Failure Localization in Data Center Networks.. In *USENIX NSDI (2019)*.

[19] Chaoliang Zeng, Layong Luo, Teng Zhang, Zilong Wang, Luyang Li, Wenchen Han, Nan Chen, Lebing Wan, Lichao Liu, Zhipeng Ding, et al. 2022. Tiara: A scalable and efficient hardware acceleration architecture for stateful layer-4 load balancing. In *USENIX NSDI (2022)*.

[20] Lior Zeno, Dan RK Ports, Jacob Nelson, Daehyeok Kim, Shir Landau-Feibish, Idit Keidar, Arik Rinberg, Alon Rashelbach, Igor De-Paula, and Mark Silberstein. 2022. {SwiSh}: Distributed Shared State Abstractions for Programmable Switches. In *USENIX NSDI (2022)*.