SlimeMold: Scaling Distributed Hardware Load Balancers via a Unified Giant Connection Table

Ziyuan Liu
Beihang University SKLCCSE & Microsoft Research Asia
Beijing, China
ziyuanliu@buaa.edu.cn

Zhixiong Niu
Microsoft Research Asia
Beijing, China
zhniu@microsoft.com

Ran Shu
Microsoft Research Asia
Beijing, China
rashu@microsoft.com

Jianqin Zhang
Ruijie Networks
Fuzhou, China
Zhangjianqin@ruijie.com.cn

Guohong Lai
Ruijie Networks
Fuzhou, China
Nanxi@ruijie.com.cn

Liang Gao
Ruijie Networks
Fuzhou, China
Gaol@ruijie.com.cn

Na Wang
Broadcom Inc.
Shanghai, China
na.wang@broadcom.com

Zongying He
Broadcom Inc.
Shanghai, China
zongying.he@broadcom.com

Jacob Nelson
Microsoft Research
Redmond, USA
Jacob.Nelson@microsoft.com

Dan R. K. Ports
Microsoft Research
Redmond, USA
Dan.Ports@microsoft.com

Lihua Yuan
Microsoft
Redmond, USA
Lihua.Yuan@microsoft.com

Peng Cheng
Microsoft Research Asia
Vancouver, Canada
pengc@microsoft.com

Yongqiang Xiong
Microsoft Research Asia
Beijing, China
yongqiang.xiong@microsoft.com

Abstract

In cloud data centers, stateful load balancers (LBs) are crucial for scaling applications and ensuring availability. Hardware load balancers (HLBs) implemented on programmable switches are increasingly adopted for their high throughput and low latency. However, existing approaches to scaling HLBs largely rely on VIP-based partitioning and ECMP-driven scale-out. Such methods do not adequately accommodate traffic dynamics or HLB-node heterogeneity, and are prone to per-connection consistency (PCC) violations.

To address these issues, we present SlimeMold, a scale-out HLB architecture that provides a global and unified **Giant Connection Table (GCT)** across a multi-node HLB deployment. SlimeMold partitions the GCT into fine-grained segments and assigns segments to nodes in proportion to their capacities (CPS and connection-table size). To make this practical, we design low-overhead node lookup, efficient rebalancing with minimal state movement, and fast fault tolerance with low performance impact. We implement SlimeMold on the Broadcom BCM56788 SmartToR switch. Results from a real testbed demonstrate near-linear scalability up to four logical nodes and fast failover with minimal performance degradation. Large-scale simulations show that SlimeMold achieves over 98% resource utilization despite node heterogeneity and dynamic traffic.

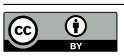
CCS Concepts

• **Networks** → **Network services**; • **Computer systems organization** → **Distributed architectures**.

Keywords

L4 Load Balancing, Hardware Load Balancers, Giant Connection Table, Programmable Switches

ACM Reference Format:
Ziyuan Liu, Zhixiong Niu, Ran Shu, Jianqin Zhang, Guohong Lai, Liang Gao, Na Wang, Zongying He, Jacob Nelson, Dan R. K. Ports, Lihua Yuan, Peng Cheng, and Yongqiang Xiong. 2026. SlimeMold: Scaling Distributed Hardware Load Balancers via a Unified Giant Connection Table. In *ACM Symposium on Cloud Computing V.1 (SoCC '26), November 18–20, 2026, Singapore, Singapore.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3815789.3827959



SoCC '26, Singapore, Singapore

ACM ISBN 979-8-4007-2735-1/2026/11
https://doi.org/10.1145/3815789.3827959

1 Introduction

In modern cloud data centers, load balancers (LBs) play an essential role in efficiently scaling cloud services by distributing incoming application traffic across backend servers, by mapping virtual IP addresses (VIPs) to direct IP addresses (DIPs) [30]. Typically, LBs use

simple functions such as hashing to distribute traffic. However, DIP changes caused by elasticity or failures can cause stateless decisions to change. Thus, production systems widely use stateful layer-4 load balancers that track prior load-balancing decisions using connection tables [13–15, 30, 34]. With the rapid growth of applications in data centers, the requirements placed on load balancers have increased significantly. Such systems must simultaneously meet three critical requirements: *multi-terabit-per-second bandwidth*, *high connections-per-second (CPS) rates*, and *large connection tables* [28, 34].

Cloud providers built software-based load balancers (SLBs) such as Maglev [13] and Ananta [30] to meet hyperscale demands. While effective, these systems incurred substantial infrastructure cost, often requiring hundreds of servers to sustain traffic loads [15, 28]. Recent years have seen growing interest in hardware load balancers (HLBs) that exploit programmable switches to offload LB logic, aiming for line-rate performance with lower cost and latency [10, 14, 28]. Compared to software LBs, these systems achieve impressive per-node gains. Duet [15] introduced a hybrid design that shifted stable flows to switches, but struggled to maintain per-connection consistency (PCC) during frequent DIP pool updates—a common occurrence in dynamic cloud environments [28]. SilkRoad [28] is the first system to embed PCC directly into programmable switches. Tiara [34] pushes further by augmenting switches with FPGA accelerators and servers, significantly extending both connection-table capacity and CPS throughput. Quantitatively, SilkRoad and Tiara deliver up to 1.6 Tbps throughput and microsecond-level tail latency (vs. tens of Gbps and hundreds of μs in software). SilkRoad supports around 200K CPS with a 100 MB connection table, and Tiara sustains up to 1.8M CPS with a 4 GB table.

Despite these achievements, SilkRoad and Tiara remain fundamentally scale-up designs, improving per-node CPS and connection table size. SilkRoad makes a limited attempt at scale-out by assigning or migrating VIPs across ToR, aggregation, and core switches, with each switch independently maintaining per-connection state. When a VIP's load becomes too large, it must be reassigned or migrated across different switch layers, reflecting the difficulty of predicting VIP-level load distribution. This VIP-based distribution assumes predictable flow patterns and stable VIP popularity [11, 13], leading to frequent migrations, load imbalance, and added complexity, while still falling short of connection-level distribution and leaving scalability constrained by per-device limits. Tiara's reliance on specialized FPGA+switch hybrids, while powerful, bypasses the abundant programmable resources already embedded in the datacenter fabric and requires costly hardware upgrades. As a result, large-scale scalability is still fundamentally bounded by the physical limits of individual devices.

This contrast motivates our key question: *Can hardware load balancers move beyond scale-up and adopt a true scale-out architecture, unifying distributed switch resources into one logical system?*

To answer the question, we propose SlimeMold, a new paradigm for HLB systems in data centers. SlimeMold pools the CPS and connection-table capacity of individual switches into a global, unified **Giant Connection Table (GCT)** for the entire HLB. This GCT is divided into **segments**. The controller assigns each switch a number of segments based on its connection-table size and CPS capacity. Each SlimeMold node is aware of the segment assignments and can direct incoming traffic to the corresponding node with the required connection state. This assignment is independent of topology, flow distribution, traffic volume, and VIP variations, ensuring linear scaling for the entire system and balanced resource usage across all nodes. With the GCT, we can also utilize nodes with heterogeneous connection tables and CPS capacities and achieve fine-grained migration and low-overhead fault tolerance.

SlimeMold provides efficient algorithms for GCT partitioning and assignment. To divide the GCT, SlimeMold sets the total number of segments according to the scale of the network, guaranteeing both high utilization and low lookup overhead on switch table resources. Furthermore, SlimeMold introduces an efficient segment rebalancing algorithm to maintain high utilization during node changes or failures.

SlimeMold adopts a fast/slow-path design in which the software slow path handles VIP-to-DIP decisions and maintains a full replica of the connection state, making the GCT abstraction practical: migration and failover reduce to copying connection-table state in the slow path and then loading the affected entries into the switch pipeline on demand. For fault tolerance, SlimeMold uses standard nodes as backups without dedicated standby devices. Backups are organized at segment granularity, where each segment is replicated to two backup nodes and a node's segments are distributed across many backups to avoid hotspots. This design is feasible because the slow path has ample memory to hold the replicated state, while the hardware fast path reserves only a small fraction of entries to absorb traffic from failed segments, enabling low-overhead recovery.

We design an efficient SlimeMold node pipeline that leverages the characteristics of programmable architectures, incorporating packet filters, multiple tables, and packet modifiers to realize the required packet processing functions. For implementation, we select the Ragile SmartToR Switch equipped with the Broadcom BCM56788 SmartToR chip [1], a practical programmable switching ASIC intended for next-generation ToR deployments. This chip supports the Network Programming Language (NPL) [3], making it suitable for realizing SlimeMold on production-grade hardware. Finally, we validate the feasibility and scalability of SlimeMold through both a real testbed and large-scale simulations.

This paper makes the following contributions.

- We propose SlimeMold, a new paradigm for hardware load balancing that enables *connection-level scale-out*. By introducing a global GCT, SlimeMold aggregates the CPS and connection table capacities of multiple programmable switches into a single logical HLB.
- We design key mechanisms for practical and efficient operation of the GCT, including segment partitioning, capacity-aware assignment, minimal-movement rebalancing, and efficient fault tolerance.
- We implement SlimeMold on a Broadcom 56788 SmartToR switch with production-grade programmability, demonstrating the feasibility of deploying GCT-based scale-out in real hardware.
- Our evaluation shows that SlimeMold achieves *linear scaling* across up to four logical nodes in a real testbed, fast failover with minimal performance impact, and above 98% resource utilization

in large-scale simulations under heterogeneous and dynamic workloads.

2 Background and Motivation

2.1 Limitations of Existing HLB Designs

LBs are essential components in data centers as they distribute client requests across multiple backend servers, enhancing service reliability and scalability. The operation of an LB can be intuitively understood as involving two primary steps: server selection (DIP decision) and packet forwarding. During the server selection phase, when a new client flow arrives targeting a VIP, the LB selects a backend server (DIP) based on certain policies, such as hashing or round-robin selection. Subsequently, during packet forwarding, all packets belonging to the established flow are consistently routed to the chosen DIP.

Among various LB designs, stateful load balancers [2, 13, 30, 34] are widely adopted because they maintain a connection table to track flow-to-server mappings. This approach ensures per-connection consistency (PCC), meaning that once a connection is established, subsequent packets are consistently forwarded to the same server, even if the backend server pool changes. PCC is essential because flows carry state (e.g., TCP state machines) that only the selected server can correctly process.

Historically, many production LBs have been implemented using software solutions [2, 13, 30]. However, software-based solutions often incur significant operational expenses, primarily due to their high resource consumption and maintenance overhead.

2.2 Hardware LBs

Recent years have witnessed the emergence of HLBs, which offload LB logic to programmable networking devices. Compared to software LBs, HLBs deliver substantial advantages: higher throughput, lower latency, and improved cost and energy efficiency. For example, Duet [15] and Rubik [14] improve efficiency by offloading stable traffic to switches and using SLBs only for complex exceptions.

Most HLBs, however, follow a *scale-up* design philosophy, pushing more functionality and resources into a single node. SilkRoad [28] illustrates this by embedding LB logic directly into programmable switches. Tiara [34] exemplifies pure scale-up, combining programmable switches with FPGAs and servers to build a powerful single-node LB. These designs deliver dramatic performance gains but remain fundamentally bounded by device-level capacity.

Quantitatively, SilkRoad and Tiara achieve up to 1.6 Tbps throughput and sub-4μs tail latency, far surpassing software solutions such as SMux (38 Gbps, ~100μs). These results firmly establish HLBs as the next generation of LB architecture.

Nevertheless, both SilkRoad and Tiara remain constrained by the physical limits of a single device. Programmable switches today only offer tens of MBs of on-chip memory (50–100MB) for connection tables [15, 22, 24], and their CPS capacity remains bounded at around 100K [10, 34]. While SilkRoad makes limited attempts at distributing VIPs across nodes, this approach remains coarse-grained and cannot scale effectively. Tiara, on the other hand, demonstrates the peak of single-node scale-up, but fails to leverage the abundant programmable resources already present in the datacenter fabric.

We next examine the fundamental limitations of both scale-up designs and limited attempts at scale-out, which reveal why neither approach suffices as a long-term solution.

2.2.1 Limits of VIP-based Scale-out. Solutions like SilkRoad [28] attempt to enhance scalability by deploying HLBs across different network tiers (e.g., ToR, aggregation, core) and dynamically migrating VIP traffic between them. While this design partially realizes scale-out, its traffic distribution remains largely constrained by VIP-granularity assignments. This can lead to several issues: **(1)** It is difficult to reliably enforce PCC in this setting: it relies on ECMP at in-path switches to distribute traffic when two HLBs serve the same VIP; however, this ECMP-based distribution can violate PCC under failures, since in-path ECMP is stateless—changes in links or HLB membership can reshuffle paths and break PCC [13]. **(2)** It is difficult to allocate VIPs optimally *a priori* because VIP demand is inherently dynamic: fluctuating traffic patterns [11, 13], unpredictable VIP popularity [34], and varying numbers of active VIPs [34] all contribute to load imbalance across nodes; and **(3)** Migration overhead can be substantial [15]: suboptimal VIP assignments force frequent cross-layer migrations that can create bottlenecks and prevent full utilization of the switch pipeline. Thus, VIP-granularity distribution cannot provide robust, system-wide scalability.

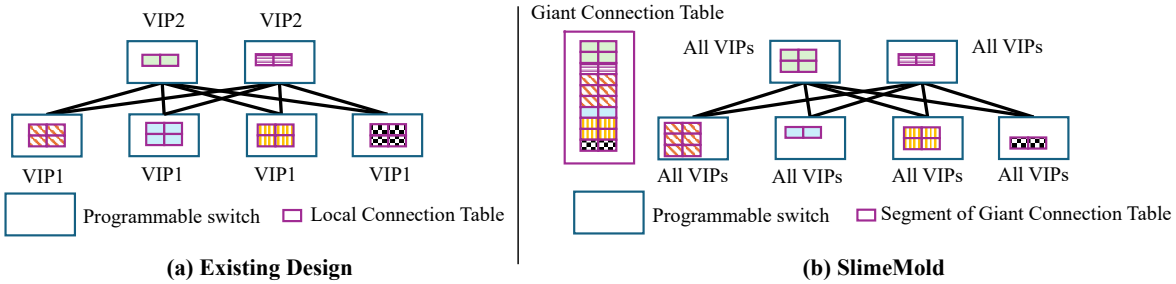
2.2.2 Limits of Scale-up: Underutilized In-network Resources. Scale-up designs such as Tiara [34] deliver impressive single-node performance by concentrating functionality into specialized hardware or new hybrid devices (e.g., FPGA–switch combinations). However, this approach overlooks the abundant programmable switches already widely deployed in datacenter fabrics—including Broadcom's Trident series [7], Nvidia's Spectrum series [6], and emerging disaggregated designs such as Sirius [9]. As a result, significant in-network resources remain underutilized, while operators must still invest in costly and power-hungry additional hardware. Furthermore, even the most aggressive scale-up strategies cannot eliminate the inherent single-node bottlenecks of CPS, connection-table size, and bandwidth. This raises a question: how can we effectively achieve scale-out to overcome the limitations of scale-up designs?

2.3 Opportunities and Challenges

With the widespread deployment of programmable networking devices, a natural opportunity arises to move beyond scale-up designs. A true scale-out architecture would enable multiple heterogeneous switches to cooperate as a unified logical LB, pooling their CPS, connection-table, and bandwidth resources to overcome the inherent limits of individual nodes.

However, enabling true scale-out is non-trivial. Several challenges must be addressed: **(1)** Distributed resources need to be globally coordinated to appear as one logical system, **(2)** Heterogeneous hardware capacities require careful capacity-aware allocation, and **(3)** Robust mechanisms are needed to support dynamic scaling, flow migration, and fault tolerance in a distributed and heterogeneous environment.

We propose SlimeMold (Fig. 1(b)) to address these challenges. Unlike existing approaches that distribute traffic at VIP granularity



(a) Existing Design **(b) SlimeMold**

Figure 1: Difference Between Existing Approaches and SlimeMold

and rely on ECMP for node selection (Fig. 1(a)), SlimeMold introduces a global connection table (GCT), logically segmented across multiple switches and centrally managed. Each switch is assigned GCT segments proportional to its connection-table size and CPS capacity. This design enables efficient resource utilization across heterogeneous devices, simplifies fault-tolerant scaling and flow migration, and significantly improves the scalability of LB systems.

3 SlimeMold Design

In this section, we overview SlimeMold's architecture and workflow (§3.1), and introduce the Giant Connection Table (GCT) that pools resources across all nodes (§3.2). Next, we detail the hybrid fast/slow-path node design (§3.3), the failover mechanism for high availability (§3.4), and existing switch coexistence (§3.5).

3.1 SlimeMold Overview

To maximize the utilization of programmable switch resources, SlimeMold is designed to be deployed on all available programmable switches within the data center. Nodes with insufficient resources to host connection table entries participate in the system solely for table lookups. Switches at the boundary of the SlimeMold deployment (i.e., ToR and core switches) announce the full range of VIP addresses; thus, traffic destined for a VIP is routed to the nearest SlimeMold node via standard routing protocols.

SlimeMold aggregates the memory resources of all participating nodes to form a logical Giant Connection Table (GCT). Connection entries are distributed across these nodes. Each node maintains a lookup mechanism to identify and forward packets to the specific node hosting the relevant connection entry.

Figure 2 illustrates the architecture and packet workflow: (1) An incoming packet reaches an entrance SlimeMold node, which identifies the target node responsible for the flow's segment. (2) The packet is forwarded to this target node, which queries its local connection table to retrieve the Direct IP (DIP) and forwards the packet to the backend server. SlimeMold encapsulates packets between nodes to facilitate routing and stage identification.

SlimeMold employs hashing to partition the GCT. The key space is divided into equal-sized ranges, referred to as *segments*. Each segment is exclusively assigned to a single SlimeMold node, though ownership can be transferred during rebalancing or failure recovery.

To determine segment ownership, each node maintains a *segment lookup table*—a complete mapping of segments to their host nodes. We opted for this single-level lookup design to strike a balance between memory overhead and the latency induced by routing detours.

Regarding the extra hops caused by these detours, our approach incurs overhead comparable to existing solutions. Conventional systems typically partition VIPs across the network, meaning the node announcing a VIP is not necessarily on the shortest path for all incoming traffic. While locality optimizations are possible, maintaining them under dynamic traffic conditions is challenging. Since SlimeMold also distributes state globally, the degree of traffic detouring remains similar to that of standard VIP partitioning schemes. Furthermore, production cloud LBs commonly enable Direct Server Return (DSR), where distinct client requests traverse the LB while large server responses bypass it entirely [13, 30]; this standard practice further mitigates the impact of LB-side bandwidth consumption in our system. In terms of memory, this single-level lookup table incurs approximately 1.6% capacity overhead in a typical deployment of one thousand switches. While a two-level lookup scheme could further reduce this memory footprint, it would impose a significantly higher hop penalty, making the single-level approach more favorable.

A logically centralized global controller manages the initial distribution (Algorithm 1), rebalancing (Algorithm 2), and failure-handling preparation (Algorithm 3) of GCT segments, detailed in Appendices B–D of the extended version [27]. While the controller interacts with SlimeMold nodes as a single logical entity, it can be implemented as a distributed system to ensure high availability.

3.2 Giant Connection Table

One of the key questions SlimeMold should address is how to partition the GCT under dynamic workloads and heterogeneous node capacities. Specifically, we divide the decision into two steps: 1) decide the total number of segments (§3.2.1) and 2) assign segments to each node and rebalance segments if the requirements change (§3.2.2). Additionally, SlimeMold provides a mechanism for migrating GCT entries between nodes during rebalancing.

3.2.1 GCT partitioning. Hashing effectively distributes load across segments. When distributing load at the segment granularity, the total number of segments must be sufficiently large to ensure balanced distribution among nodes with varying capacities. However, we observe that due to traffic randomness, resource utilization saturates beyond a certain number of segments.

Let N denote the number of nodes in SlimeMold. We introduce a scaling ratio r to determine the total number of segments relative to the cluster size. The total number of segments, N_s, is calculated

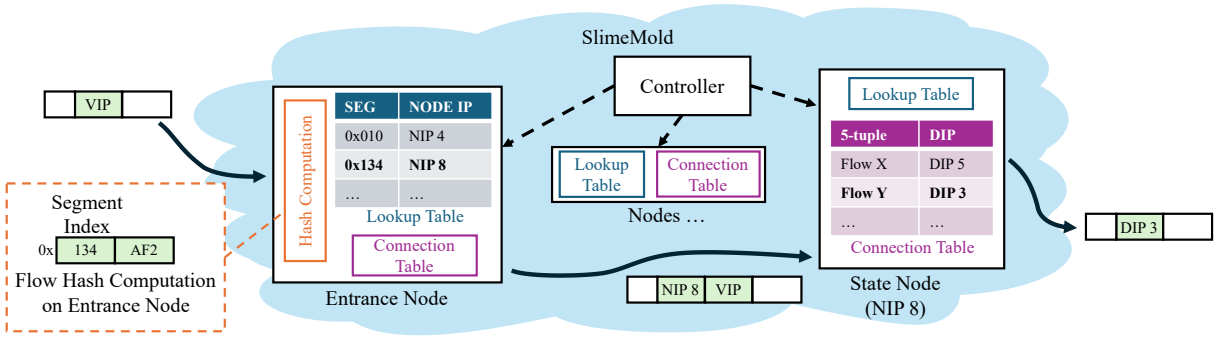


Figure 2: SlimeMold System Architecture Overview

as $N_s = 2^{\lceil \log_2(rN) \rceil}$. We enforce N_s to be a power of two to facilitate efficient mapping of flow hash values to segment indices via bitwise masking on switch hardware.

Determining the optimal r analytically is difficult due to variations in workload patterns and resource fragmentation. Instead, our experiments show that $r = 200$ is the smallest value that achieves near-maximum resource utilization across diverse traffic distributions (detailed in §6.4).

3.2.2 GCT assignment. The objectives of GCT assignment are twofold: 1) to maximize resource utilization by maintaining balanced load distribution across nodes, and 2) to minimize the number of migrated segments during rebalancing by maximizing locality (i.e., minimizing state migration).

While Consistent Hashing [21] and Rendezvous Hashing [33] favor locality, they often sacrifice precise load balancing. Conversely, Maglev Hashing [13] provides superior load balancing but exhibits poor locality, resulting in unnecessary state migration during updates. SlimeMold introduces an optimized assignment strategy to overcome these limitations.

When a node change occurs, Maglev hashing regenerates the mapping between the segments (buckets in the Maglev paper) and the nodes (backends in the Maglev paper) for all segments [13]. Although the assignment algorithm attempts to keep most segments unaffected, there are still many unnecessary movements, which is far from optimal. Instead, SlimeMold divides the assignment into two separate steps: deciding the number of segments on system initialization and moving a minimal number of segments during rebalancing. The details of the two steps are as follows.

Segment allocation. Since hashing ensures load uniformity across segments, the specific identity of segments assigned to a node is irrelevant for load balancing; only the *number* of segments matters. Let the capacity of the i-th node be c_i, and the total capacity be $C = \sum c_i$. We allocate segments as follows: First, each node i is assigned a baseline of $s_i = \lfloor (c_i/C) \times N_s \rfloor$ segments. Any remaining unassigned segments are distributed greedily to nodes starting with the highest capacity. This approach minimizes the relative impact of the additional load, as adding a segment to a high-capacity node causes a smaller percentage increase in load than adding it to a low-capacity node. Appendix B of the extended version [27] provides the algorithm details.

Segment rebalancing. When segment rebalancing is required, our goal is to minimize the number of migrated segments by moving segments only when a node's allocated quota changes. Let s^1 and s^2 denote the segment allocations before and after a change, respectively. We define the discrepancy for node i as $\delta_i = s_i^2 - s_i^1$. The minimum number of segments to transfer, N_{trans}, is $\sum_{\delta_i > 0} \delta_i$. To generate the migration plan, we iteratively transfer segments from nodes with a surplus ($\delta_j < 0$) to nodes with a deficit ($\delta_i > 0$) until all discrepancies are resolved, ensuring that exactly N_{trans} segments are moved (detailed in Appendix C of the extended version [27]).

Given SlimeMold's effective load balancing capabilities, rebalancing is triggered solely by changes in node capacity—such as failures or resource reallocation for shared services—rather than by transient workload dynamics. While shared services may exhibit oscillating resource requirements, we assume that data center operators adjust resource allocations over intervals significantly longer than the SlimeMold migration duration (typically tens of seconds). Consequently, this rebalancing mechanism avoids inducing system-wide oscillation.

3.2.3 Segment migration. During segment migration, the source node redirects packets belonging to the migrating segments to the destination node instead of routing them directly to the DIP. If the source node already holds a connection table entry for a flow, it piggybacks this state information onto the redirected packet to synchronize the destination. This is achieved by setting a specific migration flag in the packet header. Upon receiving a flagged packet, the destination node queries its local connection table and inserts the new entry only if a matching record is absent. This mechanism effectively prevents redundant state insertions.

The migration process continues for the duration of one aging period—the maximum idle time configured by the network operator (e.g., 10 seconds)—after which inactive flow states are discarded. Throughout this window, the source node forwards packets and piggybacks state data to the destination. By the end of the aging period, all active flows are synchronized, while idle flows naturally expire. This ensures that the destination node possesses all valid states, allowing for a safe transition of segment ownership.

It is important to note that while this design balances system complexity with consistency, it does not guarantee absolute strict consistency in one edge case: a flow's Per-Connection Consistency (PCC) could be violated if every single packet of that flow fails to reach the destination throughout the entire migration period. However, for active flows, the probability of complete packet loss over a full aging cycle is negligible.

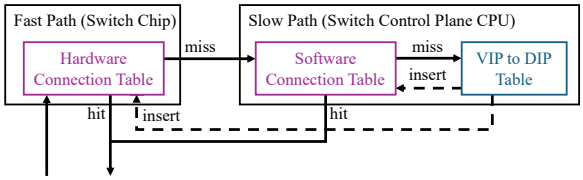


Figure 3: SlimeMold Node Connection Table Architecture

3.3 SlimeMold Node

Each SlimeMold node comprises a *fast path* built within the programmable switch pipeline and a *slow path* running on the switch control-plane. Given the limited memory resources of the switch pipeline, we carefully partition functionality between these two components. In this subsection, we focus on the primary feature of the fast path: the hardware connection table. (The detailed fast path design is provided in Appendix A of the extended version [27], and other modules like packet filters are omitted here for brevity). The slow path handles VIP-to-DIP mapping decisions and maintains a *software connection table*. This software table serves a dual purpose: it acts as an overflow buffer for the hardware table to absorb traffic bursts and stores state replicas from remote nodes to ensure fault tolerance.

Figure 3 illustrates the detailed architecture of a SlimeMold node and the connection table lookup workflow. In SlimeMold, if a lookup in the hardware connection table misses, the packet is forwarded to the slow path on the switch's control-plane CPU. The slow path software first queries its software connection table using the packet's 5-tuple. A hit indicates that the entry is either a recovered backup from another node or an overflow from the fast path. The lookup result is encapsulated in the packet header and sent back to the switch, which installs the entry into the hardware connection table and forwards the packet.

Conceptually, the SlimeMold switch control-plane functions analogously to a software Layer-4 load balancer, albeit without the responsibility of packet forwarding. Since modern software LBs achieve significantly higher CPS and support much larger connection tables than hardware-constrained LBs [34], the control-plane does not constitute a performance bottleneck in our design.

Multiple packets from the same flow may arrive before the hardware connection-table insertion completes, because both the slow-path lookup and the table update take time. Multiple inserts on the same hash key may waste the insertion bandwidth of the connection table. We add a timestamp to the software connection table to mitigate the problem. Once a hardware connection table insert is triggered, the timestamp is stored in the flow's software connection table. The slow path only issues a new table insert for a flow if the difference between the current time and the timestamp is greater than a given threshold.

3.4 Fault Tolerance

Node failures are inevitable in distributed systems. SlimeMold's fault tolerance mechanism aims to prevent connection state loss, minimize the impact on system capacity, and preserve availability during recovery.

In SlimeMold, we eliminate the need for dedicated backup nodes by utilizing standard nodes to serve as mutual backups for one another. In this model, the role of a 'backup node' is logical rather than physical, defined on a per-segment basis. To prevent resource hotspots during failures, the backups for segments from a single primary node are distributed across multiple different nodes. Conversely, each node acts as a backup for segments from multiple primary nodes. By default, we maintain two backups per segment, which implies a threefold capacity requirement for the software connection table in the slow path compared to a system without replication. However, since the slow path runs on the control-plane with ample memory, this software overhead is negligible. The hardware fast path, which is more constrained, only needs to reserve a small buffer to absorb traffic from potentially failed segments during recovery.

While our design prioritizes high availability, SlimeMold adopts a pragmatic consistency model. Specifically, PCC is maintained for the vast majority of flows, but the system does not strictly guarantee zero state loss during failures. Since state synchronization occurs asynchronously, flows established immediately prior to a node failure—within the replication latency window (typically sub-millisecond)—may not be preserved. However, given that flow states are immutable once created, complex synchronization mechanisms like versioning are unnecessary. This trade-off simplifies the design while maintaining acceptable reliability for layer-4 load balancing.

In the rare event of state loss for a recent flow, SlimeMold treats the subsequent packet as a new connection request and initiates the standard load balancing process. Given the infrequency of these events—occurring only during the narrow replication window of a node failure—the performance impact on switch nodes is negligible. Furthermore, because the load balancing algorithm ensures uniform DIP distribution, these occasional recalculations do not create traffic hotspots or microbursts among backend servers.

The remainder of this subsection details the failover design, comprising three mechanisms: efficient connection table backup (§3.4.1), optimized backup node selection (§3.4.2), and strategies to preserve service availability during recovery (§3.4.3).

3.4.1 States replication. Existing solutions for backing up programmable network states often introduce significant latency or resource overhead. For instance, RedPlane [23] relies on external storage, leading to seconds-long recovery times and unacceptable packet drops. Alternatively, active-standby approaches [9] can achieve high availability but require reserving 50% of the connection table capacity for backups, directly contradicting our goal of maximizing resource utilization.

SlimeMold leverages its distributed architecture and hybrid fast/slow path design to implement an efficient fault tolerance scheme. Instead of using dedicated backup nodes, we replicate connection tables to the slow path of remote nodes. State synchronization is periodically triggered with a replication latency window (typically sub-millisecond). Since we allow minor state anomalies, we employ a lightweight asynchronous replication mechanism over a reliable channel (TCP) rather than a heavy strong consistency protocol.

During recovery, the backup node functions similarly to a migration destination. Traffic redirected to the backup node initially causes a hardware table miss. The node then retrieves the corresponding flow state from its local software backup, installs it into

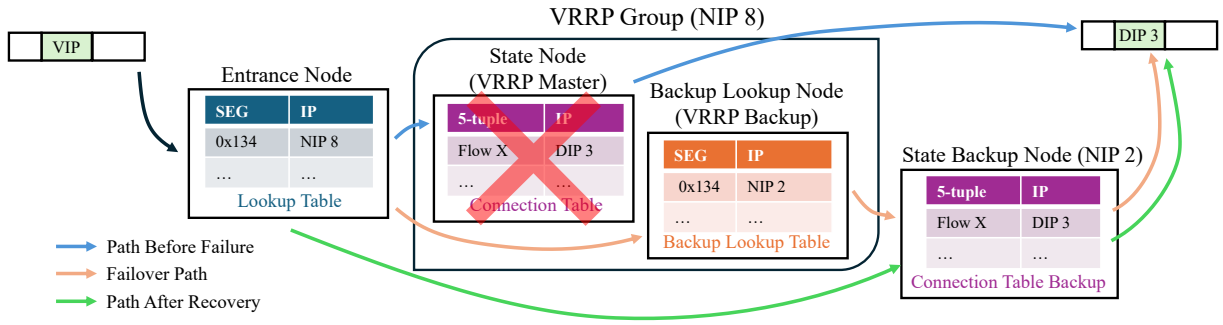


Figure 4: Workflow of SlimeMold Failover

the hardware connection table, and forwards the packet, effectively restoring the data-plane path as illustrated in Figure 3.

3.4.2 Backup node selection. To avoid making a specific backup node the system hotspot during failure recovery, SlimeMold distributes the backup responsibility for a node's segments across multiple peers. Backup selection is performed at the segment level, allowing for granular traffic redirection via simple updates to the lookup table. The SlimeMold controller manages backup placement globally with two key objectives: 1) maximize the distribution of a node's backups to prevent hotspots during single-node failures, and 2) evenly balance the total backup load across all nodes to mitigate the impact of concurrent failures. By default, SlimeMold maintains two backup replicas per segment to tolerate simultaneous node failures, a parameter configurable by data center operators.

The backup selection process follows a greedy approach to ensure balanced distribution. For each segment, the controller assigns backup nodes that currently host the fewest backup segments. This strategy ensures that backup responsibilities are uniformly distributed across the cluster. The detailed procedure is outlined in Appendix D of the extended version [27]. Notably, since concurrent failures are statistically rare, we prioritize uniform distribution over strictly optimizing for available fast path capacity.

We classify failures into two categories: temporary and long-term. Temporary failures (e.g., reboots) are resolved automatically; once the node recovers, ownership of its segments is restored with a segment migration process. Long-term failures require manual intervention. Since the initial backup assignment prioritizes dispersion rather than optimal hardware utilization, a long-term failure may result in suboptimal resource usage on backup nodes. In such cases, the controller initiates a rebalancing process to optimize the GCT layout. Given the infrequency of switch failures [16, 32], this occasional rebalancing imposes minimal overhead.

3.4.3 Preserving System Availability. Standard routing protocols may continue forwarding packets to a failed node until convergence, requiring a mechanism to maintain service availability during this window. While updating the global lookup tables to redirect traffic to backup nodes is the ultimate solution, this process involves network-wide state updates and is too slow for immediate failure response. Consequently, a faster, lower-level mechanism is required to bridge this gap.

SlimeMold leverages the Virtual Router Redundancy Protocol (VRRP) [29] for rapid failure reaction. As shown in Figure 4, every SlimeMold node operates as a VRRP master for a specific virtual IP group. For each group, the controller assigns a VRRP backup node. Upon detection of a failure, the VRRP backup immediately assumes the master role, attracting all traffic originally destined for the failed node.

A key design challenge is the mismatch between backup granularities. State backups are distributed per-segment across multiple nodes (approx. $r = 200$ segments/node), whereas VRRP operates at per-interface or per-group granularity. Coupling VRRP directly with state backups would require maintenance of hundreds of VRRP groups per node, each with a unique virtual IP—a configuration that is both complex and resource-intensive. To resolve this, SlimeMold decouples the fast-response VRRP backup from the state preservation backup. A failed node's designated VRRP peer acts as a temporary funnel; it receives the aggregate traffic for the failed node and forwards packets to their respective segment owners (the state backups). This is achieved via a compact forwarding table on the VRRP peer (orange line in Figure 4). The size of this table corresponds to the number of segments owned by the failed node, imposing negligible memory overhead.

While this indirection preserves availability, it introduces a "detour" latency. Therefore, this mechanism serves as a temporary bridge. To minimize the duration of the detour, the VRRP backup node actively notifies the controller upon receiving redirected traffic. As the controller updates the global lookup tables, traffic is progressively shifted directly to the new segment owners. Once the global update is complete, the VRRP peer ceases its forwarding role. For nodes that previously used the failed node as their VRRP backup, the controller establishes new VRRP groups to restore redundancy.

To simplify management, the controller employs a greedy algorithm to select per-node VRRP backups from adjacent peers, prioritizing those with the fewest existing VRRP backup responsibilities. While the minimal configuration requires one VRRP backup per group, operators can configure multiple VRRP backups to enhance resilience against simultaneous failures.

3.5 Deployment and Coexistence

We discuss three practical considerations for deploying SlimeMold: the scenarios it targets, how it coexists with existing switch functions, and how it generalizes across programmable ASICs.

3.5.1 Concrete application scenarios. SlimeMold targets cloud L4 load-balancing scenarios where a hot VIP, bursty short connections,

or heterogeneous HLB devices can exceed the connection-table or CPS capacity of a single switch. In such cases, VIP-level partitioning is too coarse-grained and may cause imbalance or unnecessary migration. A network-wide Giant Connection Table allows multiple programmable switches to pool their connection-table and CPS resources and distribute state at fine-grained segment granularity.

3.5.2 Coexistence with switch-level policies such as ACLs and QoS. SlimeMold complements existing switch functions rather than replacing them. It is triggered only for traffic matching LB VIP prefixes, while non-LB traffic follows the normal routing pipeline. For LB traffic, ACLs and classification can be applied before SlimeMold processing. QoS metadata can be carried over to the traffic manager for buffer allocation, queue scheduling, shaping, and congestion/flow-control mechanisms such as PFC and ECN, and can be preserved across encapsulation and decapsulation. SlimeMold likewise does not override the fabric's routing or ECMP/WCMP-based traffic-engineering decisions: it relies on standard routing to reach the nearest entrance node and to forward from the state node to the DIP, so operator-configured route weights and TE policies continue to govern inter-switch forwarding. In other words, SlimeMold only determines *which* node holds a flow's state, not *how* packets are routed between switches.

3.5.3 Generalization across programmable ASIC products and generations. Although our prototype uses Broadcom BCM56788-specific resources such as FORTE tiles and hardware learning, SlimeMold does not rely on Trident-specific semantics. It only requires common programmable-switch primitives, including VIP classification, flow-key hashing, match-action or hash-table lookup, packet encapsulation/decapsulation, and control-plane table updates. Differences across ASICs are captured by SlimeMold's capacity-aware segment assignment.

4 Implementation

We implement a SlimeMold prototype on the Broadcom BCM56788 SmartToR chip [1]. Users can program their own data-plane behaviors on this chip using the Network Programming Language (NPL). This chip also integrates multiple on-chip Cortex-R series ARM cores for real-time assistance in data-plane and control-plane processing. We implement all features of SlimeMold nodes with 3.7K lines of NPL code for the programmable data-plane and 1.7K lines of C code for the ARM cores. All tables and packet processing logic are implemented on the programmable data-plane. The C code on the ARM cores assists with connection-table insertion and deletion operations.

For lookup tables, we use the on-chip hash computation module to calculate the hash of the segment index, and use the hash table resource to implement lookup tables. The VIP table is implemented using the hash table. Each of these tables has 10K entries. Because the hash computation module must work after the normal switch routing table in the switch pipeline, we cannot leverage the normal routing table for packets that pass the lookup tables. Instead, we directly encode the egress port as the value in the lookup tables.

We use the FORTE tile resource in BCM56788 to implement a large connection table with 1M entries. The FORTE tiles are SRAM tiles shared across multiple stages in the ingress/egress pipeline.

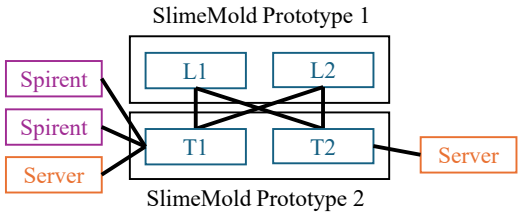


Figure 5: Real Testbed Topology

Each entry has a 5-tuple (source IP, source port, VIP index in VIP table, protocol flag (UDP or TCP)) as the key, and a DIP index as the value (or policy action in Broadcom's terminology). The DIP index is used as an array index to query another hash table with 64K entries to get full information about the DIP. We implement a fast indexing pass in the switch hardware pipeline and a slow index update pass in the ARM software. Thus, the query operation is performed purely on the hardware. Creation and deletion are implemented using a hybrid hardware/software path.

There is a hardware learning module on the chip to accelerate insertion into FORTE, which can manage the insertion entirely in hardware. We take advantage of this module to implement the creation operation. The module can learn the required keys from the packet and insert an entry into the connection table with the default policy action. The ARM core then updates the policy action to the target value.

We implement a SlimeMold controller in 1.5K lines of C++ including all algorithms designed for SlimeMold. Communication between the controller and SlimeMold's nodes is facilitated via gRPC [5]. Upon setup, a node establishes a gRPC channel to connect with the controller, and uses an RPC to register itself. This channel also serves for the node to inform the controller about any failures in its BFD pair. To direct a node's actions based on events such as node failure or load rebalancing, the controller establishes an additional channel with the node, invoking the node's RPCs to manage its future behavior.

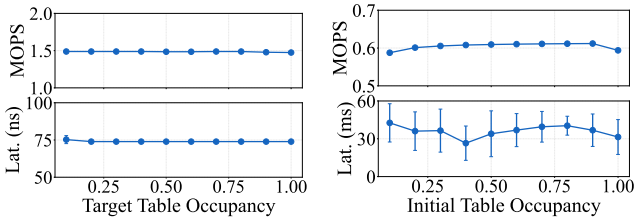
5 Real Testbed Evaluation

In this section, we evaluate the design of the SlimeMold system on real hardware. Specifically, we evaluate the scalability of SlimeMold in a four-node setting. In addition, we evaluate single-node performance, especially its ability to process connection-table entries during essential operations.

We built an Ethernet switch using the Broadcom BCM56788 [1] programmable switch chip with 32 100Gbps ports. The switch ran the SlimeMold prototype that we implemented. We manufactured two switches and divided them into four logical SlimeMold nodes: T1 combined with T2 on a single switch, while L1 and L2 shared another switch. SlimeMold nodes on different switches were connected by 100Gbps links in a full mesh topology, shown in Fig. 5. We implemented static hashing for the VIP-to-DIP decision on the switch control-plane CPU. We used two Spirent FX3-100GQ-T2 test modules [4] to evaluate the characteristics of the prototype. One operated as a traffic generator, and the other performed measurement. In addition, there were two servers connected to T1 and T2, respectively. We set the table insert threshold to 5 μs because the RTT and table insert latency are very low in our testbed. Our hardware testbed validates SlimeMold's data-plane mechanisms,

Table 1: Performance of Four Logical Nodes

# of Nodes	1	2	3	4
# of Connections	444K	893K	1333K	1770K
Scalability Factor	100%	100.6%	100.1%	99.7%



(a) Insert (b) Delete

Figure 6: Performance of Different Operations

local failover behavior, and four-logical-node scaling, but it does not reproduce a full deployment with many independent ingress nodes. We therefore evaluate distributed behavior using the large-scale simulator in §6.

5.1 SlimeMold System Performance

In this evaluation, we test whether SlimeMold can fully utilize the connection-table resources on all available nodes and achieve linear scaling. To measure the total capacity of the Connection Table of SlimeMold, we increase the number of long-lived connections injected into the system at a granularity of 1K, and stop when the packet drop rate exceeds 1%. To generate traffic, we uniformly choose the source IP from 192.0.0.1 to 192.255.255.255, and fix the destination IP as 172.0.0.1. The traffic protocol is TCP. The source port is randomly selected from the range 1024 to 65535, and the destination port is fixed at 80.

The available connection-table capacity on each logical node is about 500K. However, because of severe hash collisions when table occupancy is high, insertions may fail after exceeding the retry limit. We treat the node as fully utilized when the drop rate is over 1%. Due to randomness in multiple factors, there is always some variance in the results. For n nodes, the scalability factor is defined as the measured performance divided by n times the single-node performance. The results are summarized in Table 1. The results demonstrate that SlimeMold achieves linear scaling from one to four logical nodes.

5.2 Single Node Performance

In this section, we measure the performance characteristics of a single SlimeMold node in detail. We measure the performance of a SlimeMold node using traffic generated by multiple Spirent test modules. A Spirent test module is connected to two ports and sends bidirectional traffic to test the throughput and latency between these ports. We first insert 50% flow entries into the Connection Table. Then we generate 1500-byte packets whose destination address is in the Connection Table and measure throughput and latency. The throughput is the sum of the line rate of the two connected ports. For latency, the results have little variation but all measurements are less than 2us. The result demonstrates that SlimeMold's logic does not have a large impact on the network performance. Because we do not have enough traffic generators, we cannot test

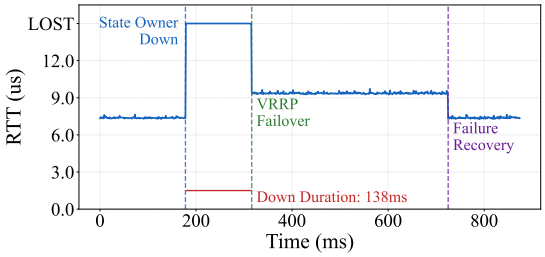


Figure 7: SlimeMold Failover Time

larger throughput. As FORTE tiles can achieve line rate lookup, we expect to achieve the maximum chip performance 8Tbps with a larger testbed.

We further evaluated the performance of Connection Table operations, specifically insertions and deletions, with results plotted in Fig. 6. Error bars indicate standard deviation across five repeated experiments. The standard deviation is below 0.5% of the mean for all metrics except the 10% occupancy point in insertion and the deletion latency, so the error bars are largely invisible. To quantify the insertion capability (Operations Per Second, OPS), we inserted 1 million entries into the SlimeMold node and measured the average insertion rate at varying occupancy levels (from 0.1M to 1M in 0.1M increments). To measure insertion latency, we configured the switch to set a flag bit upon a Connection Table hit, then calculated the time elapsed between sending the initial packet and receiving the first flagged packet. The average insertion speed observed was 1.487MOPS, significantly outperforming existing control-plane-based methods like SilkRoad [28]. This acceleration is attributed to the offloading of hash computation and table management to the Broadcom SmartToR hardware [1]. It is important to note that SlimeMold decouples flow insertion from state replication; reliability is maintained via asynchronous software backups on the slow path. Consequently, these metrics strictly reflect the hardware connection table's insertion performance.

The measurement methodology for deletion operations follows a similar approach. In contrast to insertion, deletion throughput is lower, ranging from 0.587MOPS to 0.612MOPS, and exhibits a slight upward trend as table occupancy increases. Deletion latency measures approximately 36ms with large variance. The maximum measured duration slightly exceeds the configured 50ms batch deletion interval, as it includes the overhead required for scanning the validity bitmap.

5.3 SlimeMold Failover

This section assesses the failover performance of SlimeMold in handling a node failure event. We use a single flow with ping-pong traffic between the client and the DIP to test SlimeMold's failover performance. We configure the segment-assignment algorithm to assign the role of each physical node. Specifically, T1 works as the entrance node. L1 holds the flow state as the state node. T2 is the backup lookup node that forwards corresponding traffic to the state backup node (L2). See Fig. 4 for the node type definition. The original path is client–T1(entrance node)–L1(state node)–T2–DIP. If L1 fails, SlimeMold detects failure by BFD and uses VRRP to change the path to client–T1(entrance node)–L2–T2(backup lookup node)–L2(state backup node)–T2–DIP. This detour adds one extra hop

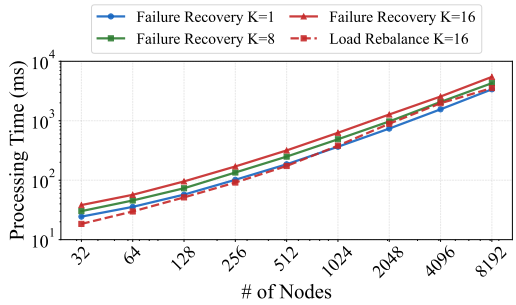


Figure 8: Controller Scalability of $K = 1, 8, 16$ Node Failures and Load Rebalance Events (in log-scale)

through T2, temporarily raising latency. Once notified of the state node's failure, the SlimeMold controller modifies lookup tables on T1 to change the path to client–T1(entrance node)–L2(state backup node)–T2–DIP–client to fully recover from the failure. With this setup, we expect similar network latency before failure and after recovery. The latency should be slightly higher during failover. We assume a 1024-node setting to emulate the controller latency for lookup-table changes, which is 373ms (number from §5.4). We employ the switch's vendor-provided BFD implementation with 50ms heartbeat intervals and a master-down threshold of 3 missed heartbeats.

The results of SlimeMold's reaction to the failure of one node are presented in Fig. 7. The recovery procedure has multiple stages. At 178ms, the node fails and the packets from the client to the DIP are lost. At 316ms, SlimeMold detects the state node failure and forwards packets to its backup lookup node then to the flow state backup node. The detour through the backup lookup node raises the latency to 9.4us compared with 7.5us for direct redirection to the state backup node. The service recovery time is 138ms. The controller completes full recovery within 373ms after the failure is detected, updating the lookup tables on T1 so that traffic bypasses the detour and latency returns to initial levels. This suggests that BFD-enhanced VRRP effectively minimizes service downtime, reducing it by 72.85% in this case, with potential greater benefits in larger SlimeMold topologies. Notably, BFD-enhanced VRRP could accelerate failure response if implemented on the programmable data-plane, albeit at the cost of additional system overhead.

5.4 SlimeMold Controller Performance

This subsection presents an evaluation of the controller's scalability. Since our focus is on the controller's performance, our evaluation employs a separate node to simulate all SlimeMold nodes by returning a success signal upon receiving an instruction from the controller. Essentially, this evaluation measures the time spent by the controller and the associated communication costs. We varied the number of nodes in powers of two within the range $[32, 8192]$, instructing the controller to manage K simultaneous node failures (for $K \in \{1, 8, 16\}$) followed by a load rebalancing process (aimed at mitigating the suboptimal segment assignment resulting from node failures) and recorded the time taken to complete each event. Each configuration was executed 5 times, and the average and standard deviation were calculated. The measured relative standard deviations range from 1% to 5% for failure recovery and 2% to 7% for load rebalancing, indicating stable results.

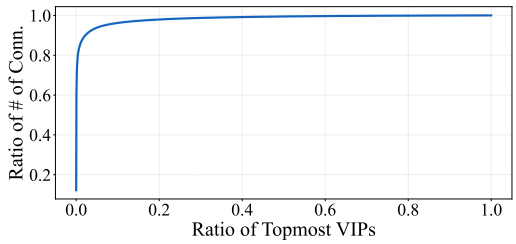


Figure 9: Number of Connections Distribution of Topmost VIPs

The results are presented in Fig. 8. The figure indicates that the controller performance cost exhibits an almost linear trend. The peak completion time for the single-node failure event is 3.38s for 8192 nodes, transferring about 17MiB across all nodes, not per node, for a node failure event. For $K = 16$ simultaneous failures, the peak completion time increases to 5.47s, a ~62% increase, while the data transfer volume grows proportionally. For the load rebalancing process after a single-node failure, the peak appears at 2.98s for 8192 nodes, transferring only a small amount of data in total (less than 3KiB). After $K = 16$ failures, the rebalance time increases modestly to 3.67s, since the rebalance data remains small regardless of K. Breakdown analysis reveals that the most time-intensive actions during the evaluation are the gRPC calls between the controller and all nodes, which are necessary to relay table updates (accounting for over 95% of the total time). The time difference between a single-node failure event and the load rebalancing procedure is attributed to the different amounts of data transferred in total. All other operations (core computation: segment reallocation and backup selection) take less than 1ms. The results suggest that managing SlimeMold data structures in a centralized controller is not a time-consuming task, and future deployments can use advanced technologies to optimize communication costs.

6 Performance at Large Scale

Due to limitations in hardware quantity, we implement a flow-level simulator in C++ to evaluate SlimeMold at scale and to understand the impact of various parameter settings on its performance. Specifically, we ask four questions: (1) whether SlimeMold can efficiently utilize available resources, (2) what hop-count cost its detour routing introduces, (3) how much overhead load rebalancing introduces, and (4) whether our parameter choices are sound.

As stated in §3.1, bandwidth would not be a bottleneck in the data center with SlimeMold deployed; thus, we can omit the network topology and bandwidth. In the simulation, we assume that either CPS or table resources can be the bottleneck, and treat the other as infinite on all nodes. In the following, we use the term capacity to represent the capacity of the bottleneck resource.

To generate a node-capacity configuration, we let each node draw a sample from $\mathcal{N}(0, 1)$ and quantize it into 10 levels from 10% to 100% of 1M capacity over $[-3\sigma, 3\sigma]$ (covering 99.7% probability). If the sample falls outside $[-3\sigma, 3\sigma]$, the capacity is clamped to 10% or 100%.

To generate traffic, we use a one-hour trace from a production datacenter. As shown in Fig. 9, a small number of VIPs carry about 80% of connections, and the top 40% of VIPs have almost all flows. In simulation, we generate 30K VIPs from 172.0.0.0/8 with traffic

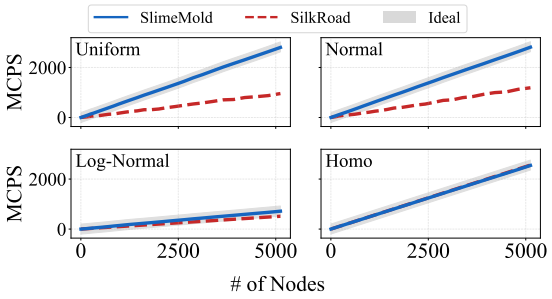


Figure 10: CPS Scaling under Different Capacity Distribution

volumes conforming to this distribution; each flow uses a random source IP from 192.0.0.0/8 with TCP to a fixed destination port 80.

6.1 CPS Scaling and Table Resource Efficiency

In our large-scale evaluation, we compare the performance of SlimeMold with SilkRoad [28]. SilkRoad does not specify a detailed algorithm for distributing VIPs across data-center nodes, especially under heterogeneous capacity configurations. We adopt the First-Fit-Decreasing algorithm [20] to solve the VIP-assignment problem, and assume that SilkRoad can predict traffic load perfectly.

In simulation, we use the Fat-Tree topology with parameter $k = 64$, which has 32 pods, 2048 ToR switches, 2048 Aggregation switches, and 1024 Core switches. The total number of switches in the topology is 5120. To generate VIP traffic, in addition to the setup described in §6, we assume that the traffic within one VIP is distributed evenly on each ToR.

Besides normal, we also evaluate SlimeMold and SilkRoad under three other capacity distributions: uniform, log-normal, and homogeneous. Uniform and log-normal use the same 10-level quantization as normal, over $[0, 1]$ and $[\exp(-3\sigma), \exp(3\sigma)]$ respectively. In the homogeneous distribution, all nodes are assigned the same capacity.

We evaluate both the CPS bottleneck and connection table bottleneck of SlimeMold. For the CPS bottleneck, since connections-per-second is independent of service time distribution, we generate connections one by one, route each through SlimeMold's two-phase routing to a state node, and stop when any node's CPS count exceeds its capacity. For the table bottleneck, where concurrent connections $= \lambda \cdot E[S]$ depend on service time distribution S, we use time-stepping simulation with Poisson arrivals and M/G/∞ departures. We derive the service-time distribution S from the DCTCP workload [8]. Because both metrics yield consistent results, we present only the CPS comparison here.

Fig. 10 compares SlimeMold and SilkRoad on Fat-Tree topologies with $k \in \{2, 4, \ldots, 64\}$ across four node-capacity distributions. SlimeMold maintains efficiency above 98% regardless of distribution, confirming that its aggregate CPS scales linearly with cluster size. In contrast, SilkRoad's efficiency varies from 34% (uniform) to almost 100% (homogeneous) because its VIP-level assignment is sensitive to node heterogeneity. The gap is most pronounced under heterogeneous distributions where SilkRoad's First-Fit-Decreasing heuristic cannot fully absorb capacity imbalance. Only under the homogeneous case do the two systems converge, confirming that the difference is driven by SlimeMold's segment-level allocation, which decouples VIP popularity from per-node capacity.

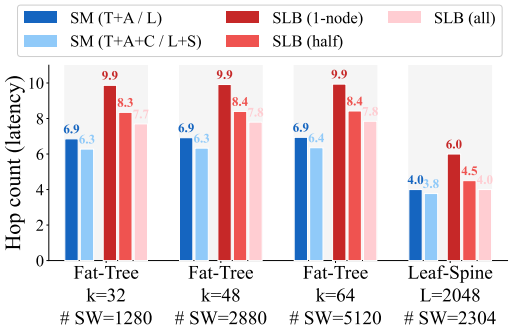


Figure 11: Extra Hops

6.2 Forwarding Overhead Evaluation

SlimeMold routes packets through an *entrance node*—the nearest switch within scope—which then forwards to a state node for connection lookup before reaching the DIP. This indirection introduces a detour compared to a direct path from ingress to DIP. To quantify this cost and compare it against server-based alternatives, we measure the average hop count on three Fat-Tree topologies ($k \in \{32, 48, 64\}$, corresponding to 1,280, 2,880, and 5,120 switches) and one Leaf-Spine topology ($L = 2048$, $S = 256$, 2,304 switches). **SlimeMold** uses two-phase routing as described in §3. We evaluate two scopes: `tor+agg` (ToR and Aggregation switches) and `tor+agg+core` (all switches). **Server-based LB** (encompassing both Tiara [34] and Maglev [13], which share the same forwarding model) uses ECMP to select the nearest server, which performs the connection lookup, then forwards the packet to the DIP. We evaluate three deployment scales: 1-node (only a single lookup server in one specific rack), half-pod (servers spread across half the pods), and all-pod (servers spread across every pod). For each topology we generate 10^6 random 5-tuple flows, and record the hop count from the source ToR to the DIP's ToR of each flow. Each configuration is repeated 5 times with different random seeds; the standard deviations are negligible (< 1% of the mean) and omitted from the figure for clarity. The results are shown in Fig. 11.

On the $k = 64$ Fat-Tree, SlimeMold (`tor+agg`) averages 6.94 hops and `tor+agg+core` averages 6.35 hops. The server-based LB at all-pod deployment averages 7.85 hops, and at 1-node deployment 9.94 hops. The gap persists across all Fat-Tree sizes ($k = 32$: 6.87 vs. 7.70; $k = 48$: 6.92 vs. 7.80). On the Leaf-Spine, SlimeMold (`leaf`) averages 4.00 hops and the server-based LB at all-pod also reaches 4.00 hops; including the spine switches in the scope (`leaf+spine`) further reduces SlimeMold to 3.78 hops.

The key difference is *where* the lookup happens. In SlimeMold, the state node is typically a ToR or Aggregation switch in another pod (for cross-pod flows, the average distance to the state node is ~ 4 hops on a $k = 64$ Fat-Tree). The packet traverses switch-to-switch fabric links only; there is no server round-trip. In the server-based LB, the lookup resides on a server connected to a ToR: the packet must traverse a ToR-to-server link (+1 hop) for the connection lookup, then return to the ToR (+1 hop) before reaching the DIP. This +2 hop penalty is architectural and does not diminish with deployment scale—at 64 pods the server-based LB achieves its lowest switch-fabric hop count (5.85 hops), but the +2 overhead brings the total to 7.85, still 0.91 hops above SlimeMold. On the Leaf-Spine, the +2 penalty is offset by the shorter cross-leaf path (2

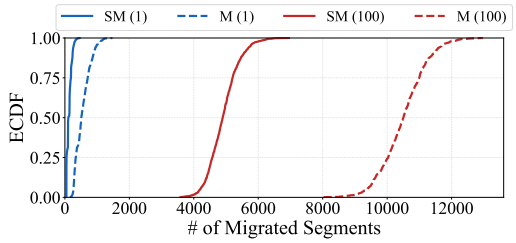


Figure 12: Segment Affinity to Node Capacity Changes

hops), which is why the two systems converge at full deployment; SlimeMold's `leaf+spine` scope breaks the tie by expanding the state distribution to include the spine layer.

Each additional switch-fabric hop adds approximately 1.2 μs of forwarding latency, as measured on our testbed. The hop-count differences reported above therefore translate into forwarding latency differences.

6.3 Segment Affinity During Rebalancing

SlimeMold rebalances GCT distribution when node changes occur. SlimeMold aims to minimize the number of segments migrated to reduce the migration overhead. In this section, we evaluate the segment affinity of SlimeMold, and compare it with the Maglev hashing algorithm [13]. The Maglev hashing algorithm described in the original paper is only for homogeneous node capacity settings. The paper only introduces a rough idea of how to extend it for heterogeneous node settings. We implement a heterogeneity-aware Maglev hashing algorithm.

We set the initial number of nodes to 1000. The number of segments is 262144, accordingly. We generate 1000 groups of node capacity configurations. For each group, we randomly regenerate the capacity configuration of 1 or 100 nodes to simulate node capacity changes. We use both the SlimeMold algorithm and Maglev hashing to decide the new segment assignment. We measure the number of migrated segments as the metric of segment affinity.

We find that Maglev always migrates many more segments than SlimeMold does. We plot the ECDF of the number of segments migrated by the two algorithms in Fig. 12 (SM for SlimeMold, M for Maglev; (1) and (100) denote the number of nodes with a changed capacity). SlimeMold ensures that only the necessary number of segments is moved, representing an optimal scenario. In contrast, Maglev moves 4.21x and 2.13x as many segments as the optimal when changing 1 and 100 nodes, respectively. This difference arises because Maglev recreates the entire assignment based on preference rather than moving only the affected segments.

6.4 Rationale of Setting r

In this subsection, we provide the rationale for setting r in SlimeMold. Given r and a node count N, the number of segments is $2^{\lceil \log_2(rN) \rceil}$, the smallest power of two $\geq rN$. We sweep r from 1 to 500. For each r, we evaluate 20 randomly chosen node counts N from 100 to 1000. For each (r, N) pair, we run 100 simulations with different random seeds under each of the three distributions (uniform, normal, log-normal). For a given r, we first take the minimum utilization across all seeds for each N, then take the minimum across all N values. As shown in Fig. 13, utilization saturates at approximately $r = 200$ across all three distributions. Beyond this point,

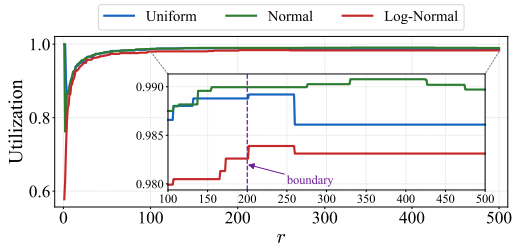


Figure 13: Performance of Different r under Various Node Capacity Distribution

increasing r yields negligible improvement. This upper bound arises from the inherent randomness of hash-based flow distribution. We therefore set $r = 200$ as the default.

7 Related Work

This section discusses work related to global state in the switch data plane, switch-centric distributed systems, and network functions (NFs) on programmable switches.

Global NF States on Switch Data Plane. The management of global NF state on servers is a well-explored area. However, because of the superior processing capabilities of switches, researchers have recently begun to handle these states on programmable switches. SwiSh [35], NetChain [18], and P4xos [12] realize state coordination protocols in the switch data plane to achieve reliable global NF state on multiple switches. Unlike earlier research focusing on reliability, SlimeMold aims first to expand state capacity with a global perspective. Each node retains only a data segment and is part of a mutual awareness framework. These techniques can help improve Connection Table migration and role backup within SlimeMold.

Switch-based Distributed System. Recently, there have been many works on switch-based distributed systems. For example, distributed machine learning [17, 25, 31] and key-value stores [19, 26] are at the application level, while NetRPC [36] and ExoPlane [24] attempt to provide some general abstractions of such systems to reduce user effort. Sirius [9] can achieve resource disaggregation in stateful NFs. SlimeMold is another instance of a switch-based distributed system and offers a new state management scheme in the data plane to design such systems.

8 Conclusion

We presented SlimeMold, a scale-out HLB architecture that unifies per-node state into a *GCT* and allocates load proportional to node capacity through segment partitioning, capacity-aware assignment, minimal-movement rebalancing, and low-overhead failover. A Broadcom BCM56788 prototype shows linear scaling to four logical nodes with minimal failover impact, and large-scale simulations demonstrate >98% resource utilization across diverse capacity distributions and traffic patterns. **This work does not raise any ethical issues.**

Acknowledgments

We thank the anonymous SoCC reviewers and shepherd for their constructive comments. Zhixiong Niu and Ran Shu are the corresponding authors.

References

[1] 2023. BCM56780 Series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56780.

[2] 2023. DPVS is a high performance Layer-4 load balancer based on DPDK. https://github.com/iqiyi/dpvs.

[3] 2023. NPL – Open, High-Level language for developing feature-rich solutions for programmable networking platforms. https://nplang.org/.

[4] 2023. Spirent FX3 2-Port Quint-Speed QSFP28 Modules. https://www.spirent.com/assets/u/spirent_fx3_hse_module_datasheet.

[5] 2025. gRPC. https://grpc.io/.

[6] 2025. NVIDIA Spectrum SN5000 Series Switches. https://nvdam.widen.net/s/mmvbnpk8qk/networking-ethernet-switches-sn5000-datasheet-us.

[7] 2025. Trident 5 / BCM78800 Series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm78800.

[8] 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)*.

[9] Deepak Bansal, Gerald DeGrace, Rishabh Tewari, Michal Zygmunt, James Grantham, Silvano Gai, Mario Baldi, Krishna Doddapaneni, Arun Selvarajan, Arunkumar Arumugam, et al. 2023. Disaggregating Stateful Network Functions. In *USENIX NSDI (2023)*.

[10] 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)*.

[11] Theophilus Benson, Aditya Akella, and David A Maltz. [n. d.]. Network traffic characteristics of data centers in the wild. In *ACM IMC (2010)*.

[12] Huynh Tu Dang, Pietro Bressana, Han Wang, Ki Suh Lee, Noa Zilberman, Hakim Weatherspoon, Marco Canini, Fernando Pedone, and Robert Soulé. 2020. P4xos: Consensus as a network service. *IEEE/ACM Transactions on Networking* 28, 4 (2020), 1726–1738.

[13] 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)*.

[14] 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)*.

[15] 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).

[16] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. 2011. Understanding network failures in data centers: measurement, analysis, and implications. In *Proceedings of the ACM SIGCOMM 2011 Conference*. 350–361.

[17] Richard L Graham, Devendar Bureddy, Pak Lui, Hal Rosenstock, Gilad Shainer, Gil Bloch, Dror Goldenerg, Mike Dubman, Sasha Kotchubievsky, Vladimir Koushnir, et al. 2016. Scalable hierarchical aggregation protocol (SHArP): A hardware architecture for efficient data reduction. In *2016 First International Workshop on Communication Optimizations in HPC (COMHPC)*. IEEE, 1–10.

[18] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. 2018. NetChain: Scale-Free Sub-RTT Coordination. In *USENIX NSDI (2018)*.

[19] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. Netcache: Balancing key-value stores with fast in-network caching. In *ACM SOSP (2017)*.

[20] David S Johnson. 1973. *Near-optimal bin packing algorithms*. Ph. D. Dissertation. Massachusetts Institute of Technology.

[21] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. 654–663.

[22] Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. 2020. Tea: Enabling state-intensive network functions on programmable switches. In *ACM SIGCOMM (2020)*.

[23] Daehyeok Kim, Jacob Nelson, Dan RK Ports, Vyas Sekar, and Srinivasan Seshan. 2021. Redplane: Enabling fault-tolerant stateful in-switch applications. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 223–244.

[24] Daehyeok Kim, Vyas Sekar, and Srinivasan Seshan. 2023. ExoPlane: An Operating System for On-Rack Switch Resource Augmentation. In *USENIX NSDI (2023)*.

[25] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael M Swift. 2021. ATP: In-network Aggregation for Multi-tenant Learning.. In *USENIX NSDI (2021)*.

[26] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. 2019. Distcache: Provable load balancing for large-scale storage systems with distributed caching. In *USENIX FAST (2019)*.

[27] Ziyuan Liu, Zhixiong Niu, Ran Shu, Jianqin Zhang, Guohong Lai, Na Wang, Zongying He, Jacob Nelson, Dan R. K. Ports, Lihua Yuan, Peng Cheng, and Yongqiang Xiong. 2026. SlimeMold: Scaling Distributed Hardware Load Balancers via a Unified Giant Connection Table (Extended Version). https://aka.ms/slimemold.

[28] 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)*.

[29] Stephen Nadas. 2010. Virtual Router Redundancy Protocol (VRRP) Version 3 for IPv4 and IPv6. RFC 5798. doi:10.17487/RFC5798

[30] 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.

[31] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. 2021. Scaling Distributed Machine Learning with In-Network Aggregation. In *USENIX NSDI (2021)*.

[32] Rachee Singh, Muqeet Mukhtar, Ashay Krishna, Aniruddha Parkhi, Jitendra Padhye, and David Maltz. 2021. Surviving switch failures in cloud datacenters. *ACM SIGCOMM Computer Communication Review* 51, 2 (2021), 2–9.

[33] David G Thaler and Chinya V Ravishankar. 1998. Using name-based mappings to increase hit rates. *IEEE/ACM Transactions on networking* 6, 1 (1998), 1–14.

[34] 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)*.

[35] 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)*.

[36] Bohan Zhao, Wenfei Wu, and Wei Xu. 2023. {NetRPC}: Enabling {In-Network} Computation in Remote Procedure Calls. In *USENIX NSDI (2023)*.

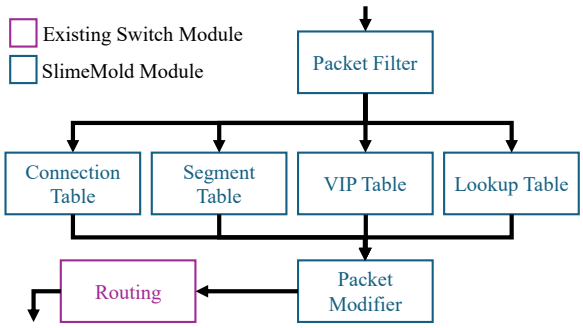


Figure 14: SlimeMold Fast Path Architecture

Appendices

A SlimeMold Fast Path

We now detail the essential functionalities and inherent capabilities of the SlimeMold fast path. Figure 14 illustrates the functional pipeline. Upon entering a SlimeMold node, the **Packet Filter** first checks for a SlimeMold encapsulation header; if present, the header is stripped. The Packet Filter then dispatches the packet to one or more processing tables: the VIP Table, Lookup Table, Segment Table, and Connection Table. Based on the filter's selection and subsequent table lookup results, the **Packet Modifier** executes the appropriate actions, such as performing load balancing, forwarding to the slow path on a miss, or encapsulating packets for inter-node tunneling. Finally, the standard routing module determines the egress port. Crucially, a SlimeMold node simultaneously operates as a standard data center switch; non-LB traffic bypasses SlimeMold modifications and is routed normally. The specific roles of each module are detailed below:

Packet Filter. The Packet Filter determines the packet's processing path by inspecting the encapsulation status. If no SlimeMold header is present, the packet is either entering the SlimeMold system for the first time or is unrelated to load balancing; its disposition is determined by the VIP Table. For encapsulated packets, the filter decodes the SlimeMold header to identify the processing stage (e.g., LB forwarding or connection table insertion) and directs the packet to the appropriate subsequent tables.

Connection Table. The Connection Table stores the VIP-to-DIP mappings that constitute the core resource of SlimeMold, supporting query, insert, and delete operations. Packets flagged for lookup trigger a query to check for existing states. Packets carrying a 'DIP insert' directive trigger an insertion in the data plane. Connection termination packets (e.g., TCP FIN) trigger entry deletion.

VIP Table. Data center VIPs are typically assigned within a unified prefix range [13], allowing the VIP Table to be compact. SlimeMold leverages standard Longest Prefix Matching (LPM) to efficiently identify packets destined for VIPs announced by the node.

Lookup Table. The Lookup Table maps flow hash prefixes to the IP address of the responsible SlimeMold node (next hop). Ideally, if the destination is the local node, the Segment Table (see below) will also register a hit, superseding the Lookup Table result to optimize local processing.

Segment Table. A packet may arrive directly at the node that owns its corresponding segment. The Segment Table tracks all locally owned segments. If both the VIP Table and Segment Table register a hit, the node immediately processes the packet locally, avoiding unnecessary forwarding or pipeline recirculation.

Packet Modifier. For unencapsulated packets, the Packet Modifier performs actions only if there is a VIP Table hit. Packets designated for Connection Table lookup are either processed for load balancing (on a hit) or redirected to the slow path (on a miss). Packets carrying connection table insertion commands require no header modification.

B Segment Allocation

Algorithm 1 details the segment allocation process.

Algorithm 1: Decide Number of Segments on All Nodes

Input: number of nodes N, nodes capacity configuration $\{c_n\}_{n=1}^{N}$, number of segments N_s
Output: assigned number of segments $s : [1, N] \to [0, N_s]$

Calculate overall capacity $C = \sum_n c_n$;
Initialize $s[1 \ldots N]$ to all zero;
Initialize number of left segments l to N_s;
foreach *node* n **do**
  $s[n] \leftarrow \lfloor c_n / C \times N_s \rfloor$;
  $l \leftarrow l - \lfloor c_n / C \times N_s \rfloor$;
end
Put all nodes in a priority queue Q;
while l *is not zero* **do**
  Pop the node n with the largest capacity from Q;
  Assign one segment to the node n by letting $s[n] \leftarrow s[n] + 1$;
  $l \leftarrow l - 1$;
end
Return s;

C Segment Rebalancing

Algorithm 2 details the process of segment rebalancing.

D Backup Node Selection

Algorithm 3 details the backup-node-selection procedure.

Algorithm 2: Decide Migrated Segments

Input: number of nodes N, number of segments N_s, two assigned number of segments $\{s_n^1\}_{n=1}^N$, $\{s_n^2\}_{n=1}^N$, segment to node mapping $M^1 : [1, N_s] \to [1, N]$ corresponding to s^1

Output: segment to node mapping $M^2 : [1, N_s] \to [1, N]$ corresponding to s^2

Initialize $\delta[1 \dots N]$ to all zero;
foreach *node n* **do**
    $\delta[n] \leftarrow s^2[n] - s^1[n]$;
end
Initialize M^2 as M^1;
while δ *is not all zero* **do**
    Randomly choose a node n_1 with $\delta[n_1] > 0$;
    Randomly choose a node n_2 with $\delta[n_2] < 0$;
    Randomly choose a segment seg_idx from those assigned to node n_2;
    Migrate segment seg_idx from node n_2 to node n_1, and change mapping M^2 by letting M^2[seg_idx] $\leftarrow n_1$;
    $\delta[n_1] \leftarrow \delta[n_1] - 1$;
    $\delta[n_2] \leftarrow \delta[n_2] + 1$;
end
Return M^2;

Algorithm 3: Backup Node Selection

Input: number of nodes N, number of segments N_s, segment assignment $M : [1, N_s] \to [1, N]$, number of backup plan P

Output: backup plan $B : [1, N_s] \times [1, P] \to [1, N]$

Initialize backup plan $B[1 \dots N_s][1 \dots P]$ to all None;
foreach *node n* **do**
    foreach *segment s whose* $M[s] = n$ **do**
        Select P nodes $n_1, n_2, \dots, n_P$ that currently serve as backups for the fewest segments among all nodes except node n;
        Let s choose the P nodes as the backup, and maintain B by letting $B[s][i] = n_i$ for $i = 1, \dots, P$;
    end
end
Return B;