Toward Distributed Write-back Caching in Programmable Switches

Siyuan Sheng, Jiazhen Cai, Qun Huang, Lu Tang, and Patrick P. C. Lee

***Abstract*—Skewed write-intensive key-value storage workloads are increasingly observed in modern data centers, yet they also incur server overloads due to load imbalance. Programmable switches provide viable solutions for realizing load-balanced caching on the I/O path, and hence implementing write-back caching in programmable switches is a natural approach to absorb frequent writes and improve write performance. However, enabling in-switch write-back caching is challenged by not only the strict programming rules and limited stateful memory of programmable switches, but also the need for reliable protection against data loss due to switch failures. We first propose FarReach, a new caching framework that supports fast, available, and reliable in-switch write-back caching. FarReach carefully co-designs both the control and data planes for cache management in programmable switches, so as to achieve high data-plane performance with lightweight control-plane management. We further extend FarReach into DistReach, which reduces the reliability maintenance overhead via distributed switch deployment. Our experimental results on a Tofino-switch testbed show that FarReach achieves a throughput gain of up to 6.6× over a state-of-the-art in-switch caching approach under skewed write-intensive workloads. Also, DistReach reduces the crash recovery time of FarReach by 77.4%.**

***Index Terms*—Programmable switches, distributed write-back caching, reliability**

I. Introduction

Key-value stores are widely deployed in modern data centers to manage structured data (in units of *records*) for data-intensive applications, such as social networking [2]–[4], web indexing [5], and e-commerce [6]. Practical key-value storage workloads are traditionally read-intensive (e.g., with a read-to-write ratio of up to 30:1 at Facebook's Memcached [4]). However, recent field studies of production key-value stores indicate a shift toward *write-intensive workloads*. For example, more than 20% of Twitter's Twemcache clusters experience more writes than reads [2], and the AI/machine-learning services at Facebook's RocksDB production have 92.5% of read-modify-writes [7]. Also, write-intensive workloads are often *skewed*; for example, 25% of frequently accessed (i.e., *hot*) records dominate the write workloads at Twitter's Twemcache clusters [2].

Enabling high write performance for key-value stores in data centers is challenging. Write requests from a client to a key-value storage server often suffer from long round-trip times (RTTs) due to switch-to-server transmissions and server-side processing. If the server is overloaded, I/O requests may experience long queuing delays or even be dropped. In distributed key-value stores that span multiple servers, a small portion of servers may become bottlenecked by substantial requests for hot records under skewed workloads, thereby leading to load imbalance [8], [9].

Programmable switches [10] offer an opportunity to improve the write performance of key-value stores. By deploying a programmable switch on the I/O path (e.g., as a top-of-rack switch in a rack-based data center), it can intercept I/O requests for all servers within the rack and provide stateful memory that can be programmed to cache frequently accessed records. For each client request, the switch can read or write any of its cached records and directly respond to the client, thereby reducing the overhead of commodity servers (e.g., CPU processing, memory access, and slow PCIe transmission of disk I/O) and eliminating the long RTT to access server-side records. Load balancing is achievable by keeping only $O(m \log m)$ records, where m is the number of servers [11]. Recent studies have shown the effectiveness of load-balanced in-switch caching [12]–[15] for high throughput and sub-RTT latencies. However, existing in-switch caching approaches [12]–[15] target read-intensive workloads and use *write-through* caching (i.e., write requests update records both in the in-switch cache and the server side). Thus, they do not improve write performance compared with no caching under skewed write-intensive workloads.

To address skewed write-intensive workloads, it is desirable to implement in-switch *write-back* caching (i.e., write requests update records in the in-switch cache only without immediately updating the server side) to absorb frequent writes to hot records. However, enabling write-back caching in programmable switches faces several challenges. First, in-switch write-back caching raises the issue of synchronizing records in both the in-switch cache and server-side storage. Without proper synchronization, the latest records may become unavailable to clients during cache eviction. Second, since the in-switch cache keeps the latest records under the write-back policy, protecting against data loss in the in-switch cache during switch failures is critical. However, providing fault

This work was supported by National Key Research and Development Program of China (2023YFB2904600), National Natural Science Foundation of China (62172007 and 62302410), Research Grants Council of Hong Kong (GRF 14201523), and Fundamental Research Funds for the Central Universities under Grant ZK1142.

An earlier version of this paper appeared at the 2023 USENIX Annual Technical Conference (ATC'23) [1]. In this extended version, we propose DistReach, which supports distributed in-switch write-back caching with significantly lower reliability maintenance overhead than FarReach. We also add new evaluation results with both software simulation and hardware deployment.

Siyuan Sheng, Jiazhen Cai, and Patrick P. C. Lee are with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong (e-mail: sysheng21@cse.cuhk.edu.hk; jzcai@cse.cuhk.edu.hk; pclee@cse.cuhk.edu.hk).

Qun Huang is with the Department of Computer Science and Technology, Peking University, Beijing 100871, China (e-mail: huangqun@pku.edu.cn).

Lu Tang is with the Department of Computer Science and Technology, Xiamen University, Xiamen 361005, China (e-mail: tanglu@xmu.edu.cn).

Corresponding author: Lu Tang.

tolerance guarantees for the in-switch cache is challenged by the limited switch resources (e.g., limited stages with only tens of megabytes for stateful memory) [10], [16]. Finally, the strict pipeline programming model and limited resources in programmable switches necessitate a design of simple but efficient caching mechanisms. While programmable switches can be managed with a software controller to relax switch resource constraints [14], [15], the control-plane interaction between the controller and programmable switches can incur long delays and slow down data-plane I/O processing. Even worse, the synchronization and fault tolerance issues complicate cache management with extra control-plane overhead, thereby further degrading I/O performance.

In this paper, we present the design, implementation, and evaluation of in-switch write-back caching frameworks. We first propose FarReach, a fast, available, and reliable in-switch write-back caching framework to improve the I/O performance of key-value stores under skewed write-intensive workloads. FarReach exploits a careful co-design of the control and data planes by offloading cache management to a centralized controller in the control plane, while achieving high data-plane performance with lightweight control-plane management. It comprises the following design features: (i) fast cache admission that admits hot records into the in-switch cache without blocking data-plane I/O traffic; (ii) available cache eviction that ensures the latest records evicted from the in-switch cache remain available to read requests; and (iii) reliable snapshot generation and zero-loss crash recovery for the protection against data loss during switch failures.

FarReach targets single-switch deployment. While FarReach provides reliability guarantees, it incurs extra control-plane overhead in snapshot generation and requires client-side collaboration for zero-loss crash recovery (see §III for details). Thus, we further propose DistReach, which extends FarReach to support the deployment of distributed in-switch write-back caching with multiple programmable switches. DistReach improves the reliability maintenance of FarReach by replicating hot records across multiple switches for reliability, so as to eliminate the control-plane overhead of snapshot generation and the need for client-side collaboration.

We implement the in-switch cache prototypes of FarReach and DistReach in P4 [17], and compile them into the Tofino switch chipset [18] and software switches [19]. We evaluate FarReach and DistReach with YCSB [20] and synthetic workloads. Compared with NetCache [14], a state-of-the-art in-switch caching framework that targets read-intensive workloads and uses write-through caching, FarReach achieves a throughput gain of up to 6.6× across 128 simulated servers under skewed write-intensive workloads. FarReach also has low access latencies, fast crash recovery, and limited switch resource usage. Furthermore, DistReach reduces the crash recovery time of FarReach by 77.4% after a switch failure. We also conduct software simulation using Mininet [21] and show that DistReach achieves a throughput gain of up to 6.8× over DistCache [15], a state-of-the-art distributed in-switch caching framework.

We open-source both FarReach and DistReach (including the prototypes for Tofino-switch hardware evaluation and software simulation) at **https://github.com/adslabcuhk/distreach**.

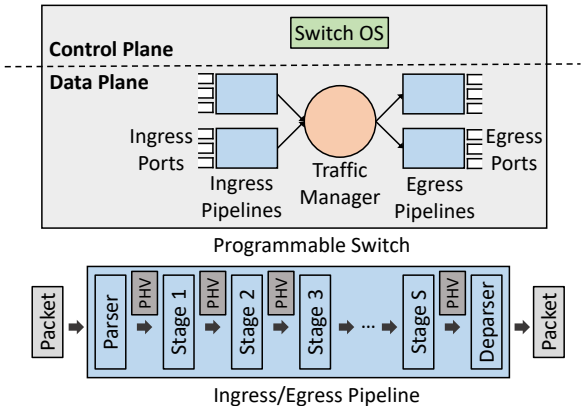


Figure 1: Programmable switch architecture.

II. Background and Motivation

A. Programmable Switches

Figure 1 shows the programmable switch architecture, which consists of both a data plane and a control plane. The data plane processes packets with stringent timing requirements for line-rate forwarding. It contains multiple *ingress* and *egress* pipelines. When a packet arrives at the switch through an ingress port, it first enters the corresponding ingress pipeline, which specifies an egress port. The *traffic manager*, which interconnects the ingress and egress pipelines, then transfers the packet to the egress pipeline corresponding to the specified egress port. Finally, the packet leaves the switch through the egress port. On the other hand, the control plane contains an operating system within the switch, called the *switch OS*, to manage the forwarding rules of the data plane. The switch OS of each switch interacts with a centralized *controller*, which manages the packet processing of all switches in a network-wide manner.

Each ingress/egress pipeline follows a reconfigurable match tables (RMT) model [10]. When a packet enters an ingress/egress pipeline, it is first processed by a *parser*, which extracts packet header fields into the packet header vector (PHV). The pipeline transfers the PHV across a number of *stages*, each with multiple *match-action tables*. Each stage also contains a limited amount of SRAM, composed of tens of memory blocks, for tracking stateful information accessible by the match-action tables. A match-action table can use an ALU to perform arithmetic or logical operations and store the results in the PHV. It matches the fields in the PHV from the previous stage and performs the corresponding action to update the PHV for the next stage, with match-action rules configurable by the switch OS. A match-action table can also use a special kind of ALU, called a stateful ALU, to store results in on-chip memory. To meet stringent timing requirements, the memory blocks associated with a stage cannot be accessed from other stages, and the processing of a packet within a stage can only access a limited number of memory blocks associated with the stage, with each memory block accessed at most once. After being updated by all stages, the PHV is processed by a *deparser*, which reconstructs the new packet header fields. The header fields are combined with the original payload to form the packet to be forwarded.

B. Challenges

Write caching policies can be classified into *write-through* and *write-back*. Write-through synchronously updates records both in the cache and on the server side; in contrast, write-back (a.k.a. delayed-write) updates records in the cache only, and later reflects the updates on the server side. Existing in-switch caches [12]–[15] mainly implement write-through caching. In this work, we focus on write-back caching, as it improves write performance over write-through caching by delaying server-side updates. However, managing write-back caching is non-trivial and faces three unique challenges in programmable switches.

Performance challenge. A programmable switch has a restricted pipeline programming model (i.e., it can only access a limited number of memory blocks) and scarce hardware resources (i.e., it has a limited number of stages and stateful ALUs) [10]. It is necessary to offload switch-level cache management (including cache admission and eviction) to a centralized controller [14], [15], while the switch only updates the cached records in the data plane under the write-back policy. However, due to the high controller-to-switch latency, control-plane processing is much slower than data-plane processing in a programmable switch, thereby bottlenecking I/O performance.

Availability challenge. Under write-back caching, both cache admission and eviction algorithms need careful coordination between the control and data planes, so as to correctly maintain the latest records in either the in-switch cache or server-side storage; otherwise, outdated records may be returned to the client. Such an issue does not exist in write-through caching [14], [15], as it always keeps the latest records on the server side. The availability issue is even more challenging in programmable switches, since the controller needs to manage both the cache and server updates. Also, the controller is not on the packet forwarding path and has no view of the traversed packets in the data plane.

Reliability challenge. Under write-back caching, the latest records may only be kept in the in-switch cache, with updates to the server-side storage delayed. If the switch crashes, all the latest records are lost. Such an issue again does not exist in write-through caching, as the latest records can be persistently kept in server-side storage [14], [15].

III. FarReach Design

A. Design Overview

We present the design of FarReach. Note that FarReach targets single-switch deployment. We extend FarReach to DistReach with multiple-switch deployment in §IV.

Architecture. FarReach is a fast, available, and reliable in-switch write-back cache architecture designed to improve the I/O performance and load balancing of server-side key-value stores. Figure 2 shows the architecture of FarReach, in which clients are connected via the in-switch cache to multiple servers for key-value storage, while the controller is responsible for cache admission and eviction. Specifically, the switch maintains a lookup table for the cached records. For each client request, the switch first checks the lookup table to determine whether the requested record is cached (i.e., a *cache hit*) or not (i.e., a *cache miss*). For each cache hit, the switch loads the cached record and forwards a response to the client. For each cache miss, the switch forwards the request to the corresponding server to access server-side key-value stores. As the controller has no view of the data plane (§II-B), cache management decisions are triggered by the switches (in the data plane) based on the workload patterns.

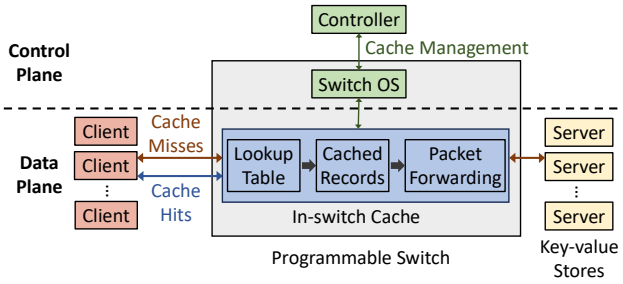


Figure 2: FarReach's architecture.

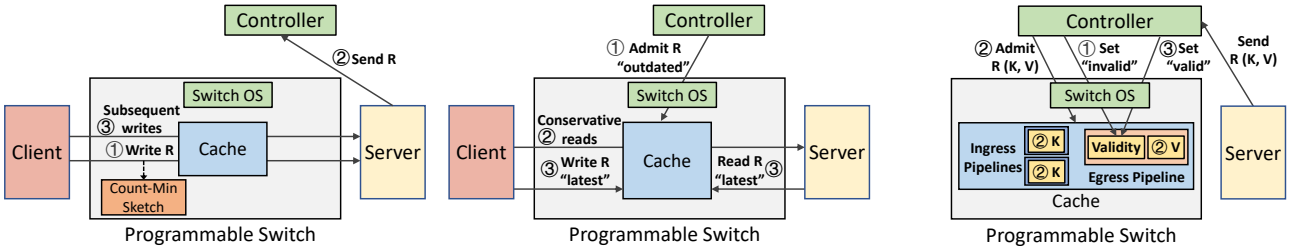
Table I: Summary of design features of FarReach.

Design features	Design details
Non-blocking cache admission (§III-B)	FarReach tracks the "outdated" or "latest" state of each cached record to limit conservative reads. It also associates a validity register with each cached record for atomicity.
Available cache eviction (§III-C)	FarReach uses a "to-be-evicted" flag to make each evicted record available. It identifies latest records by sequence numbers and handles packet loss by record embedding.
Crash-consistent snapshot generation (§III-D)	FarReach reports original cached records to the controller. It recirculates writes for atomicity, and exploits client-side record preservation for zero-loss recovery.

Goals. FarReach's core idea is a careful co-design of the control and data planes. Table I summarizes our design features. FarReach aims for three design goals:

- *Fast access* (§III-B). FarReach supports non-blocking cache admission for admitting hot records into the in-switch cache to achieve high write performance. It also ensures atomicity in cache admission under the multi-pipeline setting of programmable switches.
- *Availability* (§III-C). FarReach ensures that any latest record evicted from the in-switch cache remains available to clients.
- *Reliability* (§III-D). FarReach protects against data loss during switch failures. It uses a crash-consistent snapshot generation algorithm to create snapshots of the in-switch cache state. It also ensures atomicity of snapshot generation in the multi-pipeline setting and achieves zero-loss crash recovery by coupling snapshot generation with upstream backup [22].

Design assumptions. FarReach currently supports a fixed key length of 16 bytes and a variable value length of up to 128 bytes due to limited switch resources; the same constraint is also assumed in NetCache [14] and DistCache [15]. Thus, FarReach is suitable for workloads dominated by small records (e.g., ZippyDB and UP2X in Facebook's RocksDB production [7]). For large records, FarReach simply relays them between clients and servers without caching.



(a) Before cache admission (b) After cache admission

Figure 3: Non-blocking cache admission in FarReach. Before admitting a record R, the switch forwards subsequent writes to the server in a non-blocking manner. After admitting R, the switch conservatively forwards reads for R to the server until it receives a new write from the client or a read from the server; it also marks R as "latest".

Figure 4: Atomic validity control in FarReach. For a record R with a key K and a value V, the controller maintains an egress validity register for atomicity of cache admission.

FarReach does not support range queries, since programmable switches cannot feasibly maintain sorted structures with the memory access limitations (§II-A) and servers are unaware of the latest in-switch records under the write-back policy. In this work, we focus on skewed write-intensive workloads without range queries.

FarReach guarantees reliability for switch failures. We assume that the durability of server-side records is addressed by the persistence feature of key-value stores [23]–[25].

B. Non-blocking Cache Admission

Problem of cache admission. A naïve design of cache admission in programmable switches can introduce *blocking* to write requests. Due to limited switch resources, the controller is responsible for cache management (§II-B). Suppose that the controller is about to admit a new hot record into the in-switch cache. As control-plane processing is slower than data-plane packet forwarding (§II-B), the switch may receive subsequent writes for the same key before the record from the first write is admitted. Such subsequent writes need to be *blocked* until the record is admitted; otherwise, the admitted record may overwrite the newer records from the subsequent writes that arrive earlier at the switch due to the write-back policy.

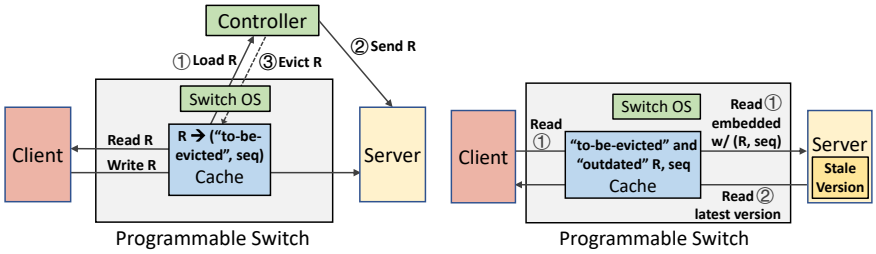
Cache admission policy. Before proposing our cache admission design, we first describe the cache admission policy in FarReach. FarReach currently triggers cache admission for the hot records with high access frequencies. It follows the design of NetCache [14] and deploys space-efficient in-switch data structures for frequency tracking, due to the limited in-switch SRAM. Specifically, FarReach maintains a Count-Min Sketch [26] to track the access frequencies of uncached records for cache admission, as well as a counter array to track the access frequencies of cached records for cache eviction (§III-C), within the switch. A Count-Min Sketch is a fixed-size, error-bounded summary data structure composed of multiple rows with a fixed number of counters each. FarReach samples incoming requests for frequency monitoring to reduce processing overhead. For each sampled request to an uncached key, FarReach updates the Count-Min Sketch and estimates the access frequency. If the frequency exceeds a pre-defined threshold, FarReach identifies the key as hot and triggers the controller to admit the hot record into the in-switch cache, while also tracking the frequency of the cached record in the counter array. To avoid counter overflow, FarReach periodically resets all counters of the Count-Min Sketch and the counter array to zero. Note that we do not claim the novelty of this design.

Our cache admission design. We propose a non-blocking cache admission algorithm for FarReach, as shown in Figure 3. Suppose that a client issues a write request for a record (say, R) to a server. If R is not yet cached and is identified as hot based on the Count-Min Sketch, the switch forwards R to the server (① in Figure 3(a)). The server forwards R to the controller for cache admission (② in Figure 3(a)). Note that a read request issued by a client can also trigger cache admission, except that the server will send the server-side latest record R to the controller (② in Figure 3(a)). Before the controller admits R into the in-switch cache, the switch forwards subsequent writes for the same R's key (i.e., cache misses) to the server without updating the cache (③ in Figure 3(a)). The server directly processes the writes without blocking, thereby keeping the latest record.

After R is admitted, FarReach temporarily marks the admitted R as "outdated" (① in Figure 3(b)). For any read request to R's key (which is "outdated"), FarReach *conservatively* forwards the read request to the server to obtain the latest record (② in Figure 3(b)).

Conservative reads increase read latencies due to server-side processing. To limit conservative reads, our insight is that all requests and responses must traverse the switch, so FarReach can monitor all traversed requests and responses to mark the "outdated" cached record as "latest" as early as possible. Specifically, FarReach marks the "outdated" record as "latest" (③ in Figure 3(b)) if it sees: (i) a write request from a client for the same key (which carries the latest record), or (ii) a read response from the server for the same key (which carries the latest record while the cached record remains outdated). When a cached record is marked as "latest", it can be directly updated by subsequent writes based on the write-back policy. Under skewed write-intensive workloads, an "outdated" cached record can soon be marked as "latest" by a subsequent write for the same key, so conservative reads are limited.

Recall that a server in FarReach is responsible for sending a record to the controller for cache admission (i.e., ② in Figure 3(a)). Thus, it can determine whether any record of the same key has been sent to the controller and avoid sending duplicate records of the same key, thereby limiting control-plane bandwidth usage (e.g., up to 1.41 MiB/s; see §VI-D). This contrasts with NetCache [14], in which a switch sends



(a) Cache eviction workflow (b) Record embedding

Figure 5: Available cache eviction in FarReach. For a record R to be evicted, it is marked as "to-be-evicted" and is made available to the client's read if it is also the latest record. To handle switch-server packet loss, if R is "outdated", the switch embeds the "outdated" evicted record into any read and forwards the read to the server. The server compares the received read with the server-side version and keeps the latest version.

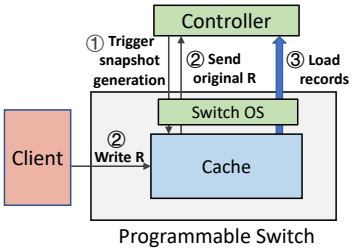


Figure 6: Crash-consistent snapshot generation in FarReach. If the switch receives the first write to a cached record R during snapshot generation, it forwards the original R to the controller before R is updated.

records to the controller for cache admission and needs an in-switch Bloom Filter [27] to avoid duplicate submissions. FarReach removes the need for maintaining an in-switch Bloom Filter, thereby saving switch resource usage for implementing in-switch write-back caching.

Atomic validity control. FarReach stores the keys and values of records in the ingress and egress pipelines, respectively, to accommodate the limited number of stages of a single pipeline. However, it is critical but non-trivial to provide atomicity for cache admission under the multi-pipeline setting. Specifically, a switch can only provide atomicity within a single pipeline rather than multiple pipelines, yet requests for the same key can arrive from different ingress pipelines. Without atomicity of cache admission, write requests to the same key arriving from different ingress pipelines may have inconsistent views on the key: cached or uncached. For the former, the cached record is updated directly by the write-back policy; for the latter, the requests are forwarded to the server based on our non-blocking cache admission design. Thus, the key may be updated with an inconsistent value.

Our observation is that key-value records are often partitioned by keys (e.g., using consistent hashing [28]) among servers connected to different egress pipelines. Even though the requests for the same key can enter a switch from different ingress pipelines, FarReach still forwards them to the same egress pipeline corresponding to the same server. By maintaining a cache instance in each egress pipeline, FarReach can accommodate requests for the same server. Note that such forwarding does not incur cross-pipeline imbalance, as the bottleneck lies in server-side storage (including both CPU processing and disk I/O) instead of line-rate switches. For example, in our evaluation, the system throughput is bottlenecked by server-side storage and is only up to 12.1 MB/s under 128 simulated servers (§VI-B), significantly lower than the maximum throughput of 3.2 Tbps of a two-pipeline Tofino switch [18]. Thus, FarReach can provide atomicity for each record being admitted, with the aid of the single egress pipeline that is connected to the corresponding server, while incurring limited performance degradation.

We propose *atomic validity control* for cache admission in FarReach (Figure 4). Specifically, programmable switches provide atomic primitives for each register within a single pipeline. FarReach introduces a *validity register* for each cached key in an egress pipeline. Before admitting a record R with key K and value V sent by a server, FarReach first sets the validity register for R as "invalid" (① in Figure 4). It then admits, via the switch OS, V into the egress pipeline and K into all ingress pipelines (the latter is to ensure consistency across all ingress pipelines) (② in Figure 4). Finally, it changes the validity register to "valid" (③ in Figure 4). Based on the validity register, FarReach treats a record as a cache hit only if the key is cached in an ingress pipeline and the validity register is "valid" in the single egress pipeline; or as a cache miss otherwise. Thus, if a key has not been admitted into all ingress pipelines, its record is treated as a cache miss as its validity register remains "invalid".

C. Available Cache Eviction

Problem of cache eviction. When the in-switch cache is full, FarReach selects a cached record to evict by sampling multiple cached records and selecting the one with the least access frequency from the counter array (§III-B). The controller then performs cache eviction on the selected record. Under the write-back policy, the evicted record may be the latest version that has not yet been updated in the server. It is critical to ensure the availability of any latest record during cache eviction. To achieve this, the controller needs to synchronize the views of both the switch and the server on the evicted record, especially when there are concurrent read/write requests for the evicted record. However, the controller is limited by slow control-plane processing, which leads to high synchronization overhead.

Our cache eviction design. We propose a cache eviction algorithm for FarReach that ensures availability, as shown in Figure 5(a). Our idea is to associate additional metadata with each cached record in the in-switch cache to maintain the availability of any evicted record, while mitigating synchronization overhead to the controller. Specifically, when a cached record (say, R) is to be evicted, the controller first marks R as "to-be-evicted" and loads R from the in-switch cache (① in Figure 5(a)). It then sends R to a server for storage (② in Figure 5(a)). If there is any write request to the "to-be-evicted" R, FarReach simply forwards the write request to the server (instead of updating the record in the cache under the write-back policy) and marks the evicted record as "outdated". If there is any read request to the "to-be-evicted" R and R is "latest" (marked in cache admission (§III-B)), the cache returns R to the client; otherwise, if R is "outdated" (i.e., it

has been updated), FarReach forwards the read request to the server, which holds the latest record. This ensures that any evicted cached record that is also the latest version remains available. After the server has stored the latest "to-be-evicted" cached record, the controller acknowledges the cache to actually evict the "to-be-evicted" *R* (③ in Figure 5(a)). Note that all writes to the "to-be-evicted" *R* must be forwarded to the server, regardless of whether the record is cached or uncached, so FarReach does not have any atomicity issue when evicting *R* in the multi-pipeline setting.

Identifying latest records. One subtlety is that a server may receive a request of storing a record from two possible paths: (i) the eviction of a record from the cache and (ii) a write request issued by a client. It is critical to differentiate the latest version of a record that is finally stored in the server. To resolve this issue, recall that FarReach forwards the write requests of the same record to the same egress pipeline corresponding to the server (§III-B). As programmable switches can provide atomicity and serialize packets in a single pipeline (§III-B), FarReach associates a *sequence number* with each cached record atomically. It increments the sequence number for each write request of the key in the egress pipeline based on the serialized order of accessing the cache, and embeds the sequence number into the write request. When the server receives a request of storing a record, it overwrites the existing record only if the received record has a higher sequence number than the existing record; otherwise, the received record is discarded.

Handling packet loss. Packet loss in switch-to-server transmissions can compromise the availability of cache eviction. During cache eviction, FarReach forwards the write request for a "to-be-evicted" record to the server and marks the evicted record as "outdated". If the write request is lost during transmission (e.g., due to server-side congestion or packet corruption), the server is not updated with the latest version and retains the stale version. As the in-switch cache marks the evicted record as "outdated", it forwards all subsequent reads to the server, which returns the stale version. Such an issue does not exist in cache admission, as a write request updates either the server (before the record is admitted to the in-switch cache) or the in-switch cache (after the record is admitted to the in-switch cache), instead of both of them.

To maintain availability under packet loss, FarReach employs *record embedding* during cache eviction, as shown in Figure 5(b). Our insight is that even though an evicted record is marked as "outdated" during cache eviction, it can still be the latest version for serving read requests. Specifically, before forwarding a read to the server, the in-switch cache embeds the "outdated" evicted record (if such a record exists) into the read; the embedded record includes the value and sequence number assigned by the switch (① in Figure 5(b)). FarReach ensures that the latest version is available to any client-issued read by comparing the embedded record with the server-side version (② in Figure 5(b)). If the sequence number embedded into a read request is larger than that stored in the server (i.e., the embedded record is the latest version), FarReach directly returns the embedded record to the client; otherwise, FarReach returns the record stored in the server (which is the latest version) to the client.

D. Crash-consistent Snapshot Generation

To protect against data loss during switch failures (§II-B), FarReach periodically generates snapshots of in-switch cached records. It also lets each client preserve the cached records generated after the latest snapshot for zero-loss recovery. Note that uncached records are protected by server-side persistent key-value stores (§III-A).

Problem of snapshot generation. We propose to generate a snapshot for all cached records in the in-switch cache at regular time points (called *snapshot points*), so that the switch can restore from the latest snapshot when recovering from a failure. However, the design of such snapshot generation is non-trivial. Since programmable switches have limited stages for cache backup and limited on-chip memory for snapshot storage, they need to offload all cached records to the controller. The snapshot overhead is limited for the controller, as it only needs to store the latest snapshot for crash recovery (e.g., 1.5 MB for 10K records with 16-byte keys and 128-byte values). When the cached records are loaded to the controller during snapshot generation, some cached records may be updated under the write-back policy, leading to inconsistencies between the final snapshot and the in-switch cache state at the snapshot point. Blocking cache updates during snapshot generation can avoid such inconsistencies, but degrades I/O performance.

Our snapshot generation algorithm. We propose a two-phase snapshot generation algorithm for FarReach to maintain crash consistency in snapshot generation without blocking cache updates. Our insight is that whenever FarReach receives the first write request to a cached record during snapshot generation, it can send the original cached record (i.e., after the snapshot point but before the first write) to the controller. This allows the controller to keep backups of all original cached records that are to be overwritten. At the end of snapshot generation, the controller replaces the overwritten cached records with their backups of the original cached records, so that the snapshot is crash-consistent with the in-switch cache state at the snapshot point. Under the skewed write-intensive workloads where most writes are issued to a small fraction of hot records, FarReach only needs to send a limited number of original cached records to the controller (for the first writes only), thereby limiting the bandwidth overhead.

Based on the insight, FarReach generates a crash-consistent snapshot in a two-phase manner (i.e., triggering snapshot generation and making a consistent snapshot) at each snapshot point, as shown in Figure 6. In the first phase, the controller notifies the in-switch cache to trigger snapshot generation (① in Figure 6). The cache monitors each write request to identify whether it is the first write to a cached record during snapshot generation. If so, the cache sends the original cached record to the controller (② in Figure 6). In the second phase, the controller loads all cached records from the cache for snapshot generation (③ in Figure 6). If a cached record has been loaded to the controller and later receives the first write, the cache no longer needs to send the original cached record. Once the controller loads all cached records, it notifies the cache about the completion of snapshot generation, reverts any overwritten cached record with the original one, and finally obtains a crash-

consistent snapshot.

FarReach carefully updates the snapshot to address two corner cases. If a new record is first admitted to the cache after the snapshot point, the controller will not include the record into the snapshot. If a cached record is evicted after the snapshot point, the controller saves the evicted record during cache eviction (§III-C), and replaces the updated record with the evicted record in the snapshot after the second phase of snapshot generation.

Atomic triggering of snapshot generation. As write requests for a record can arrive from multiple ingress pipelines, FarReach needs to trigger snapshot generation in multiple pipelines simultaneously; otherwise, the ingress pipelines may set snapshot points at different times and generate inconsistent snapshots. We propose a coordination mechanism to support simultaneous snapshot generation in multiple ingress pipelines. Specifically, at the beginning of a snapshot period, FarReach selects one of the ingress pipelines and recirculates all write requests from other ingress pipelines to the selected ingress pipeline; in other words, all write requests are processed as if they arrive at a single ingress pipeline. The controller first notifies the selected ingress pipeline to trigger snapshot generation, such that the selected ingress pipeline notifies the egress pipelines to send any original cached record that receives the first write to the controller. It then notifies the remaining ingress pipelines to trigger snapshot generation. After all ingress pipelines trigger snapshot generation, FarReach disables the recirculation, allowing the controller to perform snapshot generation with all ingress pipelines in parallel. Thus, we ensure that snapshot generation is applied to all ingress pipelines at the same snapshot point. Note that the recirculation overhead is limited, due to the short duration for notifying all ingress pipelines to trigger snapshot generation (e.g., $\approx$6 ms from our evaluation).

Zero-loss crash recovery. Our snapshot generation only guarantees crash consistency for switch failures, but cached records that are newly added or updated after the latest snapshot point remain unprotected and can be lost during a switch failure. Since switches do not have external storage for reliably keeping cached records, we propose *client-side record preservation* based on the idea of upstream backup [22] in stream processing, by keeping the copies of cached records after the latest snapshot point on the client side. Specifically, after a client sends a write request for a cached key and receives the response from the in-switch cache, it locally keeps the value and sequence number assigned by the in-switch cache (§III-C) for the cached key. After the completion of snapshot generation at each snapshot point, the controller notifies each client with the cached keys and corresponding sequence numbers at the snapshot point. Each client then releases its preserved records whose sequence numbers are no larger than those notified by the controller. Since the in-switch cache only keeps a limited number of hot records, FarReach incurs low client-side overhead for record preservation.

FarReach uses a *replay-based approach* for zero-loss crash recovery after a switch failure. It first replays the write requests of the latest cached records to update the servers for persistence. Specifically, FarReach collects both the latest in-switch snapshot (from the controller) and the client-side preserved records (from all clients), and selects the record with the largest sequence number for each cached key. If the sequence number of each selected record is larger than that stored in a server, FarReach replays the write request to store the selected record in the server for persistence. After all the latest records are persisted, clients can release their preserved records.

FarReach then recovers the in-switch cache by replaying the cache admission for each record of the latest in-switch snapshot and marking each cached record as "outdated". The "outdated" records of the in-switch cache are expected to be quickly marked as "latest" under skewed write-intensive workloads (§III-B). Note that we do not start with an empty in-switch cache from scratch, as it incurs large overhead to admit all records through the controller.

Client crashes. One limitation of FarReach is that data loss can occur if both a client and the switch crash simultaneously. If any client crashes before replay-based recovery, the cached records preserved by the client, which are not yet protected by the latest in-switch snapshot, will be lost after a switch failure. One solution is to reduce the snapshot period to a smaller window for less vulnerability, at the expense of larger snapshot generation overhead. Our evaluation shows that the snapshot generation overhead remains limited (e.g., up to 1.41 MiB/s of control-plane bandwidth when the snapshot period is 2.5 s; see §VI-D). Another solution is to exploit multiple switches, which we consider in §IV, and our evaluation shows that the control-plane overhead can be further mitigated (§VI-E).

E. Discussion

Novelty. While FarReach borrows ideas from NetCache [14] (e.g., cache admission based on a Count-Min Sketch), it introduces several novel design elements: (i) non-blocking cache admission for fast access, with atomic validity control to address atomicity issues (§III-B); (ii) available cache eviction for ensuring record availability, with record embedding to handle packet loss (§III-C); and (iii) crash-consistent snapshot generation with zero-loss recovery (§III-D). Note that the last two elements are tailored for write-back caching and are not found in NetCache, which uses write-through caching.

Trade-offs. FarReach makes two trade-offs in its design. First, FarReach trades extra switch resources for in-switch caching for higher key-value storage performance under skewed write-intensive workloads. Nevertheless, the extra switch resource usage is limited (§VI-F). Second, FarReach trades extra client-side storage and computation capacity for zero-loss recovery. Nevertheless, since clients only keep the copies of cached records after the latest in-switch snapshot, the client-side storage overhead is limited (e.g., 1.5 MB for 10K cached records with 16-byte keys and 128-byte values). The client-side computation overhead for record preservation is also limited. It takes only $\approx$0.7 μs on average to preserve a record, which is significantly smaller than FarReach's average request latency of $\approx$100 μs in our evaluation.

IV. DistReach Design

A. Design Overview

Motivation. FarReach proposes snapshot generation and client-side record preservation to ensure reliability against a switch

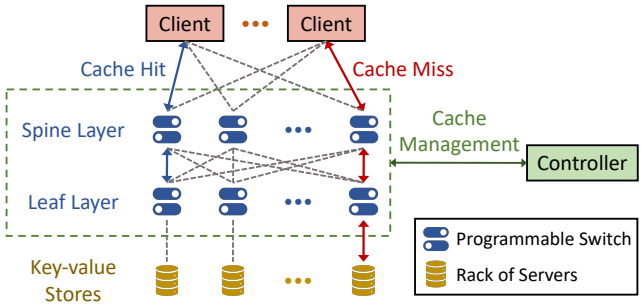


Figure 7: DistReach's architecture.

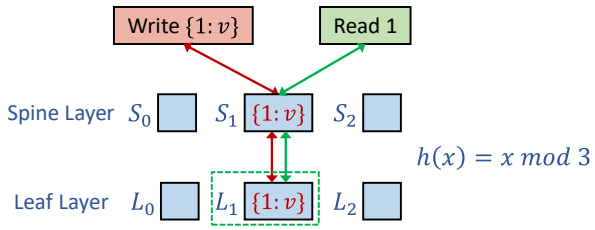


Figure 8: Example of switch-based replication.

failure (§III-D). However, this approach has two main issues. First, FarReach incurs extra control-plane overhead, as the controller periodically loads snapshots from the switch for reliable storage. Second, FarReach requires client-side collaboration to preserve records for zero-loss recovery, thereby introducing management complexities for individual clients. As mentioned in §III-D, even with client-side record preservation, data loss remains possible (albeit unlikely) during client crashes.

Since modern data centers typically use multiple switches to distribute traffic loads [29], [30], we can exploit this distributed switch deployment to replicate multiple copies of each cached record across different switches. As long as any one switch remains reliable, we can ensure the reliability of all cached records. This eliminates the need for maintaining snapshots in the controller and preserving records on the client side.

To this end, we design DistReach, a distributed in-switch write-back caching framework. In the following, we show how DistReach extends FarReach to manage multiple copies of each cached record during read/write processing and cache admission/eviction.

Architecture. Figure 7 shows the architecture of DistReach. We use the multi-layer networking infrastructure in modern data centers (e.g., spine-leaf and fat-tree topologies) to organize multiple switches for distributed in-switch caching. For simplicity, we focus on the two-layer topology (i.e., the spine and leaf layers) to describe DistReach's design. We also discuss how DistReach is deployed with more switch layers.

DistReach follows FarReach's admission and eviction designs (§III-B and §III-C, respectively) to determine which records are hot and will be cached in all layers. In each layer, DistReach partitions hot records across switches using consistent hashing [28] (as in DistCache [15]). For each record request, DistReach deterministically identifies a switch in each layer via consistent hashing and forwards the request through the identified switches in all layers. If the record is cached by switches along the I/O path, the request is served by in-switch caching (cache hit); otherwise, the request is forwarded to the server (cache miss). DistReach adds new features to manage multiple cache copies of each record for reliability.

We assume that each request issued by a client is forwarded through all layers (i.e., from the spine to the leaf layer). This assumption holds if clients are outside of the data center that hosts key-value storage. For clients inside the data center, the client-side leaf switch forwards each request to the spine layer, which processes the request as described above. Since DistReach forwards each request through specifically identified switches in all layers for cache access, we do not use conventional load balancing protocols (e.g., ECMP). Nevertheless, the switch layers are not the system bottleneck in practice for two reasons. First, we partition network traffic across multiple switches in each layer by consistent hashing to balance the load. Second, the major overhead lies in server-side I/O operations, making the performance impact of packet forwarding less significant.

B. Switch-based Replication

Each record request traverses switches in different layers based on consistent hashing, and DistReach organizes the switches based on *chain replication* [31]. Specifically, when a record request traverses the two-layer topology, DistReach forms a *chain* on the two switches (i.e., one in the spine layer and one in the leaf layer) identified by consistent hashing. DistReach replicates the record over the two switches and chains them from the spine to the leaf layer. To ensure that the switches along the chain cache the same records, DistReach deploys the same number of switches in each layer and uses the same hash function for consistent hashing.

DistReach forwards each write request along the chain of switches from the spine to the leaf layer. The write request first updates the spine switch with the latest record under the write-back policy. It then replicates the latest record to the leaf switch. After successfully updating all switches in the chain and marking the cached record as "latest" (§III-B), DistReach acknowledges the latest record to the client.

DistReach also forwards each read request across all switches in the chain and serves the request by the leaf switch. If the record cached in the leaf switch is outdated, DistReach follows FarReach's design for availability (§III-B). Specifically, DistReach conservatively forwards the read request to the server to retrieve the latest record. The server then issues a read response to the client along the chain in reverse (i.e., from the leaf to the spine layer). The read response updates all switches in the chain if the cached copies remain outdated. As the write requests quickly update the leaf switch with the latest record under skewed write-intensive workloads, the number of conservative reads is limited. On the other hand, if the record in the leaf switch is the latest (e.g., updated by the last successful write), the leaf switch answers the read request without accessing the server.

Upon a switch failure, DistReach contacts the available switch in the chain to recover the lost records of the failed switch without data loss. Specifically, suppose that DistReach now adds a newly recovered switch for recovering the lost data of a failed switch. The controller first identifies the chain

that spans the failed switch and selects the available switch in the chain (e.g., the spine switch if the leaf switch fails) to load its cached records. The controller also notifies the servers to load the cached records from the available switch. After the servers write the loaded records into persistent storage, the controller admits them to the recovered switch in the chain. Each admitted record is marked with the same state (e.g., "outdated" or "latest") as in the available switch. As the available switch caches the same records as the failed switch, the recovered switch caches all lost records of the failed switch after DistReach's recovery and achieves zero data loss.

Figure 8 gives an example of switch-based replication. Suppose that there are three spine switches, S_0, S_1, and S_2, and three leaf switches, L_0, L_1, and L_2. We use the hash function $h(x) = x \bmod 3$ to map each cached record x into the corresponding spine switch $S_{h(x)}$ and leaf switch $L_{h(x)}$. Consider a write request for a cached record with the key 1 to update its value as v. DistReach calculates $h(1) = 1 \bmod 3 = 1$ to identify the spine switch S_1 and the leaf switch L_1 as the switches caching the record. Then, it forwards the write request from S_1 to L_1 and replicates the latest value v in both S_1 and L_1. For a read request for the key 1, DistReach also forwards the request from S_1 to L_1, with L_1 serving the request.

C. Consistent Cache Management Across Switches

In our switch-based chain replication, if a cached record updated by a write request becomes available to clients (i.e., updated in the leaf switch), the record must also be updated in the spine switch for reliability. This implies that the controller needs to apply admission or eviction consistently to both the spine and leaf switches to ensure that they cache the same record. However, the RMT model (§II-A) only ensures atomicity for single-switch cache management (§III-B and §III-C), but not for multiple-switch management. Managing each switch independently for read/write requests would introduce inconsistent cache copies across switches, rendering switch failure recovery infeasible.

To ensure consistent caching in multiple-switch management, DistReach carefully arranges the order of admission and eviction of a record. For cache admission, DistReach first atomically admits the record into the spine switch in the chain following FarReach's admission design, which marks the record as "valid" (§III-B). Then, DistReach atomically admits the record into the leaf switch in the chain. For cache eviction, DistReach first atomically evicts the record from the leaf switch in the chain as in FarReach, which marks the record as "to-be-evicted" (§III-C). It then atomically evicts the record from the spine switch. As the leaf switch is the last (first) to admit (evict) the record, DistReach determines whether the record is cached by all switches in the chain via the cache status of the leaf switch. If the record is cached in the leaf switch, it implies that all switches in the chain cache the record, and DistReach serves read/write requests by the switches as described before (i.e., any acknowledged record must be updated to all switches in the chain); otherwise, DistReach forwards the record requests to the server. As the latest record is either consistently replicated to all switches in the chain or persistently stored by the server, DistReach maintains reliability in multiple-switch cache management.

D. Design Considerations

Deployment in more layers. DistReach can be extended to support more than two layers and tolerate more switch crashes. Suppose that the network has $n \geq 2$ layers. For each hot record, DistReach identifies n switches in all layers using consistent hashing to replicate n copies of the record across the chain of n switches. DistReach forwards the record requests along the chain from the first to the last layer. Each write updates n copies with the latest record from the first-layer switch to the last-layer switch for reliability, and each read retrieves the latest record from the last-layer switch for availability. To keep consistent cache copies during multiple-switch management for reliability, DistReach applies the admission (eviction) decision to the last-layer switch after (before) the other $n-1$ switches in the chain, in which each switch is managed atomically.

DistReach achieves zero-loss recovery after the failures of up to $n-1$ switches in a chain. Specifically, it selects any available switch in the chain to load the cached records, which are updated to the servers and admitted to the recovered switches. Since some write requests may not update all available switches at the time of switch failures, DistReach also updates other available switches with the records loaded from the available switch to keep consistent cache copies after recovery. For the write requests that have not successfully updated the available switch when switch failures occur, their records will not be cached by any switch after recovery. However, it does not violate availability and reliability requirements, as DistReach does not acknowledge the records to any client, and the records need not be recovered.

Performance consideration. Forwarding each request across multiple switches does not degrade I/O performance. First, the system bottleneck is the I/O overhead of server-side key-value stores instead of packet processing in switches. Traversing multiple switches incurs negligible extra latency for each request. Second, as long as a few hottest records are cached (i.e., $O(m \log m)$ records, where m is the number of servers [11]), server-side load balancing is achieved.

Routing consideration. Note that routing changes after switch failures do not affect the correctness of DistReach. When a switch fails and DistReach cannot forward requests across all switches in the chain for in-switch caching, we find an alternative path to forward the requests to the corresponding server. We delay processing these requests in the server until the lost records in the failed switch are consistently restored in all switches and the server, ensuring that the latest records are available to clients. This delay has limited performance impact. First, only requests for cached records mapped to the failed switch are delayed, while all other requests are processed by the switches and servers as usual. Second, the crash recovery time of DistReach is limited (§VI-E), so the delay is short.

V. Implementation

We implement FarReach and DistReach with both control and data planes. The control plane includes the switch OSes of

all switches and the controller, while the data plane includes multiple clients, servers, and in-switch caches. All communications among different components are based on UDP with a timeout-and-retry mechanism for low-latency yet reliable transmissions.

A. FarReach Implementation

We first describe the implementation of FarReach for single-switch deployment.

Control plane. We implement both the switch OS and the controller in C++, with 2.2K and 1K LoC, respectively, and compile the programs using `g++` (v5.4.0) with the `-O3` optimization. The switch OS provides interfaces for: (i) cache admission/eviction by configuring match-action tables and registers, and (ii) snapshot generation by loading in-switch records and sending original cached records. The controller manages the in-switch cache through the interfaces from the switch OS and coordinates snapshot generation by communicating with the switch OS and all key-value storage servers.

Clients. Our prototype is evaluated with the YCSB benchmark [20] (§VI), written in Java. We implement a client application in Java that supports YCSB, with common key-value storage interfaces including `get`, `put`, and `delete` to access records stored in both the in-switch cache and key-value storage servers. The client application also provides a shim layer to manage client-side record preservation for zero-loss recovery under switch failures (§III-D).

Servers. We deploy RocksDB (v6.22.1) [24] in each server; RocksDB is a log-structured merge-tree (LSM-tree) persistent key-value store [32] suitable for write-intensive workloads. To support multiple servers, we distribute records across servers using consistent hashing [28].

In-switch cache. We implement the in-switch cache in P4 [17] and compile it into the Tofino switch chipset [18]. The cache implementation includes both ingress and egress pipelines. Each ingress/egress pipeline in the Tofino switch provides 12 stages for pipeline programming. Each stage has 4 stateful ALUs to support at most 4 register arrays, and each register can store 4 bytes of data.

In each ingress pipeline, we deploy multiple match-action tables to prepare for egress processing. We implement a match-action table for cache lookups, which match the key (currently of size 16 bytes) in the packet header to obtain the record location in the egress pipeline. We also deploy a match-action table to trigger snapshot generation, such that each egress pipeline can send the original cached records to the controller (§III-D). Our current Tofino switch model does not support cross-pipeline recirculation, although the cross-pipeline recirculation feature is documented in the Tofino programming manual. For the evaluation purpose, we connect the selected ingress pipeline with each of the other ingress pipelines with a physical wire, so as to recirculate write requests from the other ingress pipelines to the selected ingress pipeline during snapshot generation in the multi-pipeline setting (§III-D). This physical wiring incurs extra device management overhead. Furthermore, we pre-compute the hash results for the Count-Min Sketch in the ingress pipeline and send them to the egress pipeline via each packet header to save stages in the egress pipelines.

In each egress pipeline, we store statistics, metadata, and cached values. In the first stage, we deploy a Count-Min Sketch configured with 4 rows (as in [14]). Each row corresponds to a register array with 64K registers. We use part of the second stage to maintain a counter array (as a register array) to track the access frequencies of cached records. To support write-back caching and snapshot generation, we use the remaining part of the second stage and the third and fourth stages to maintain the required metadata. We use the remaining 8 (out of 12) stages to provide 32 register arrays of 4-byte registers in total for supporting a value size of up to 128 bytes.

We address two subtle issues in the egress pipeline implementation. First, to support write-back caching, the in-switch cache needs to directly respond to a write request with a cache hit. However, the Tofino switch cannot directly change the egress port in the egress pipeline. Thus, we drop the original write request and send a response to the client by cloning. This degrades the packet processing capability of the Tofino switch, but does not undermine system performance, as the bottleneck lies on the server side. Second, to assign a sequence number for each write request, we may maintain a global counter to track the latest sequence number, but this easily leads to overflow. Instead, we use multiple global counters to reduce the likelihood of overflow. Specifically, we maintain a register array with 32K registers. We map write requests of different keys into different registers by hashing, and increment the hashed register to assign a sequence number for each write.

B. DistReach Implementation

DistReach follows the implementation of FarReach for the cache admission and eviction in each switch, with the following differences. In the control plane, DistReach no longer needs snapshot generation. Instead, it follows a specific order to admit/evict records for consistency across multiple switches during cache management (§IV). Also, the controller loads cached records from an available switch for zero-loss recovery after switch failures in a chain. In the data plane, each client does not need record preservation, but issues requests to access the switches by consistent hashing [28]. Since the switches in the last layer across the chains process all reads and writes, DistReach only deploys the Count-Min Sketch in each of the switches in the last layer to track the access frequencies of read/write requests, so as to identify hot records for triggering cache admission in the controller.

VI. Evaluation

A. Methodology

Evaluation environments. We conduct hardware evaluation and software simulation. We configure a hardware testbed that consists of a 3.2 Tbps two-pipeline Tofino switch [18] and four physical machines. Each machine has two 12-core CPUs (Intel Xeon E5-2650 v4), 64 GiB DRAM, and 2 TiB hard disk (HGST Ultrastar), and is connected to the switch via a 40 Gbps NIC (Intel XL710). We use two physical machines as clients and the other two as key-value storage servers. We connect one client and one server to one pipeline of the switch, and connect the other machines to the other pipeline. We use the

hardware testbed to evaluate the throughput and crash recovery performance of FarReach, the crash recovery performance of DistReach, and the switch resource usage of both FarReach and DistReach. We measure throughput in terms of million operations per second (MOPS) processed by FarReach.

Due to hardware limitations, we lack sufficient switches for comprehensive large-scale hardware evaluation of DistReach. Instead, we leverage software simulation based on Mininet [21] to evaluate the throughput of DistReach in distributed switch deployment. We run Mininet on a physical machine with a 12-core CPU (Intel Xeon Silver 4214), 64 GiB DRAM, and a 2.4 TiB hard disk (Seagate ST1200MM0099). We simulate four programmable switches using bmv2 [19] and organize them in spine and leaf layers, each with two switches. We connect each leaf switch to all spine switches in a spine-leaf topology. We also simulate a client that issues sufficient requests and 16 servers that are connected to the two leaf switches (each connected to eight servers). We forward requests across switches in different layers using consistent hashing, and the requests are served by either leaf switches or servers (§IV-C). We run DistReach under various workloads, especially as the number of switches per layer increases (§VI-E). Due to the limited processing capability in software simulation, we measure throughput in terms of operations per second (OPS) processed by DistReach.

Setup. We evaluate FarReach using both YCSB [20] and synthetic workloads (§VI-B and §VI-C, respectively) using hardware evaluation. Since our hardware testbed comprises only two servers, we use server rotation [14] to simulate a larger number of servers. Specifically, let N be the number of simulated servers. Given a workload, we issue requests to N logical partitions via consistent hashing [28]. We identify the partition (called the *bottleneck partition*) that receives the most requests among all N partitions. We run each experiment over N iterations. In the first iteration, we deploy the bottleneck partition in a physical server and send sufficient requests to saturate it and measure its performance. In the subsequent $N-1$ iterations, we deploy the bottleneck partition in a physical server and each of the $N-1$ non-bottleneck partitions in another physical server, and measure the performance of the non-bottleneck partition. After N iterations, we aggregate the performance of all partitions. By default, we simulate 16 servers and increase the number of simulated servers for scalability evaluation (§VI-B). Note that server rotation is only applied to static workloads without dynamic key popularity, and we also study the impact of dynamic workloads (§VI-C).

We compare FarReach against two baselines: NoCache (i.e., no in-switch caching) and NetCache [14] (i.e., the in-switch cache with write-through caching). Before each experiment, we pre-load 100M records, each with a 16-byte key and a 128-byte value, into servers that are initially empty. For FarReach and NetCache, we fix the in-switch cache size as 10,000 records and pre-load the hottest records into the cache. We also set the sampling rate as 0.5 and the pre-defined threshold as 20 requests for the Count-Min Sketch. For FarReach, we set the snapshot period as 10 s by default.

We evaluate the throughput and crash recovery performance of DistReach using software simulation and hardware evaluation, respectively (§VI-E). In software simulation, we directly run all simulated servers without server rotation. We compare DistReach with two distributed baselines: DistNoCache (i.e., no caching in all switches) and DistCache [15] (i.e., distributed write-through caching). For DistReach and DistCache, we set the cache size of each switch as 10,000 records (i.e., caching 20,000 records in two switches per layer). Since DistReach does not require snapshot generation, we do not set a snapshot period as in FarReach. In hardware evaluation, we measure the crash recovery time of DistReach from a switch failure.

One important issue is to mitigate the influence of simulation overhead in software simulation. In our hardware testbed, processing a request in the Tofino switch only takes hundreds of nanoseconds and the switch is not the bottleneck (instead, server-side storage is the bottleneck). However, in our software simulation, the simulated switches become the bottleneck, in which processing a request ranges from $\approx$1 ms for DistNoCache to $\approx$6 ms for DistCache and DistReach due to in-switch value processing. Thus, our software simulation disables in-switch value processing for cache hits in DistCache and DistReach to ensure similar in-switch processing overhead across all schemes. In addition, we inject a delay of 100 ms for each request that accesses server-side key-value stores in our software simulation, so that the system bottleneck lies in server-side storage. We run each experiment for 30 minutes to ensure sufficient client requests for stable results.

We run all experiments five times and plot the average results with 95% confidence levels based on the Student's t-distribution (some error bars may be invisible due to limited deviations).

B. Performance of FarReach under YCSB Workloads

(Exp#1) Throughput analysis. We first evaluate the end-to-end throughput using YCSB workloads: Insert (inserting records), A (50% reads, 50% writes), B (95% reads, 5% writes), C (100% reads), D (95% reads, 5% writes), and F (50% reads, 50% read-modify-writes); we do not consider range queries (i.e., Workload E) due to switch limitations (§III-A). For each workload, we generate requests with 16-byte keys and 128-byte values. The Insert workload follows a uniform distribution, workload D follows a read-latest distribution, and workloads A, B, C, and F are skewed and follow the Zipf distribution with a Zipfian constant 0.99 (default in YCSB). We verify that under NoCache, the Insert throughput to a RocksDB instance reaches 0.06 MOPS, which is consistent with prior findings [33], [34].

Figure 9 shows that FarReach increases the throughput of NoCache by 90.8%, 55.0%, 84.3%, and 71.7% in the four skewed workloads A, B, C, and F, respectively, by reducing and balancing the server-side load with in-switch write-back caching. FarReach also increases the throughput of NetCache by 83.9%, 20.0%, and 61.4% in workloads A, B, and F, respectively, and achieves similar throughput to NetCache in workload C (which is read-intensive). In NetCache, the writes of the cached keys keep invalidating the in-switch write-through cache, especially in write-intensive workloads A and F, thereby limiting the throughput of NetCache. NetCache only achieves high throughput in read-intensive workloads B and C. In the non-skewed Insert workload, both FarReach and NetCache

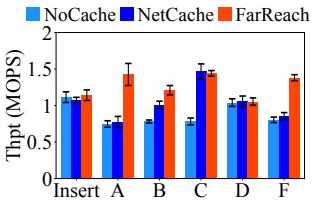


Figure 9: (Exp#1) Throughput analysis.

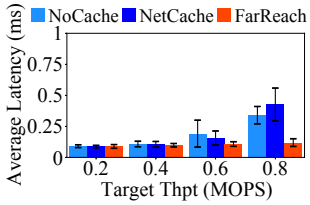


Figure 10: (Exp#2) Latency analysis.

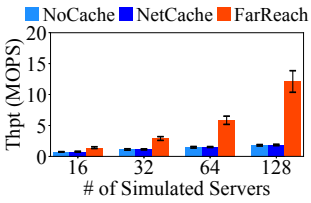


Figure 11: (Exp#3) Scalability analysis.

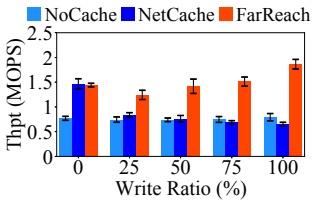


Figure 12: (Exp#4) Impact of write ratio.

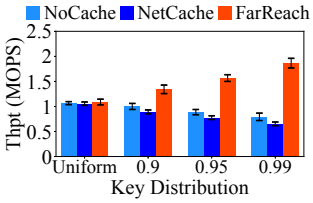


Figure 13: (Exp#5) Impact of key distribution.

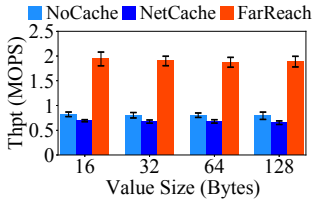


Figure 14: (Exp#6) Impact of value size.

have similar throughput to NoCache due to limited cache hits. Although the read-latest workload D prefers to request the most recently accessed records, all three schemes still have similar throughput. The reason is that the recently accessed records are frequently changed and only a small portion of requests belong to the 10,000 hottest records, leading to limited cache hits in FarReach and NetCache.

(Exp#2) Latency analysis. We next evaluate the request latencies. We focus on YCSB workload A, which is skewed and most write-intensive. We examine the trade-off between the latency and target throughput (i.e., configured by a given sending rate) as in prior studies [14], [35], [36]. We only show the average latency results, while the results of other latency statistics (e.g., medium and 95th-percentile) are similar and hence omitted for brevity.

Figure 10 shows that all schemes have small average latencies under low target throughput, as the servers do not have heavy loads and can quickly process requests. FarReach reduces the average latencies of NoCache and NetCache by 64.9% and 72.0% when the target throughput is 0.8 MOPS, respectively. For high target throughput, both NoCache and NetCache are bottlenecked by an overloaded server and hence incur large queuing delays. NetCache has a larger latency than NoCache, as NetCache needs extra server-side overhead to update the in-switch write-through cache for the write requests. FarReach effectively reduces and balances the server-side load and hence achieves a small latency. Note that NoCache and NetCache show larger confidence intervals than FarReach, especially for high target throughput. The reason is that the server-side queuing latency can vary significantly for a highly overloaded bottleneck server across different runs, while FarReach maintains a low latency due to load balancing.

(Exp#3) Scalability analysis. We evaluate the scalability of different schemes by varying the number of simulated servers. We focus on YCSB workload A. Figure 11 shows that the throughput gains of FarReach are 1.9×, 2.5×, 3.9×, and 6.6× those of NoCache and NetCache (both of which have very similar throughput) under 16, 32, 64, and 128 servers, respectively. As the number of simulated servers increases, the throughput of FarReach also increases due to load balancing across all servers, while the throughput of both NoCache and NetCache is limited by the overloaded servers due to load imbalance. Our results show that FarReach scales to a large number of servers under skewed write-intensive workloads.

C. Performance of FarReach under Synthetic Workloads

We generate different synthetic workloads with YCSB for varying write ratios (over all reads and writes), key distributions, value sizes, and key popularities. By default, we generate requests with 16-byte keys and 128-byte values, where the keys follow the Zipf distribution with a Zipfian constant 0.99, and set the write ratio as 100% (i.e., write-only requests).

(Exp#4) Impact of write ratio. We first vary the write ratio of the synthetic workload. Figure 12 shows that FarReach increases the throughput of NoCache by 67.1-135.3% for different write ratios, due to load balancing in FarReach. FarReach achieves similar throughput to NetCache when the write ratio is zero (i.e., read-only requests), while increasing the throughput of NetCache by 48.3-185.7% as the write ratio ranges from 25% to 100%. The throughput gain of FarReach over NoCache and NetCache is the highest when the write ratio is 100% due to in-switch write-back caching. Note that NetCache has slightly lower throughput than NoCache, especially under a write ratio of 100%, due to the extra server-side overhead to maintain cache coherence for write requests.

(Exp#5) Impact of key distribution. We next consider synthetic workloads under the uniform key distribution as well as the Zipfian key distributions with different Zipf constants. Figure 13 shows that all schemes achieve similar throughput of ≈1 MOPS under the uniform key distribution, as most requests are from uncached keys and are processed by the servers, so FarReach cannot benefit from in-switch write-back caching. For the skewed workloads, FarReach increases the throughput of NoCache by 34.2-135.3% and that of NetCache by 50.4-185.7%. The throughput gain of FarReach is higher when the workload is more skewed (i.e., a larger Zipfian constant), as NoCache and NetCache become more imbalanced.

(Exp#6) Impact of value size. We further vary the value size of the synthetic workload from 16 bytes to 128 bytes (while the key size remains 16 bytes); note that the number of records that can be cached in both NetCache and FarReach (i.e., 10,000 records) remains unchanged. Figure 14 shows that the throughput gains of FarReach over NoCache and NetCache remain almost the same at 2.3× across different value sizes, as the caching behavior of FarReach mainly depends on the key distribution.

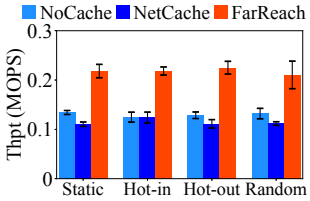


Figure 15: (Exp#7) Impact of key popularity changes.

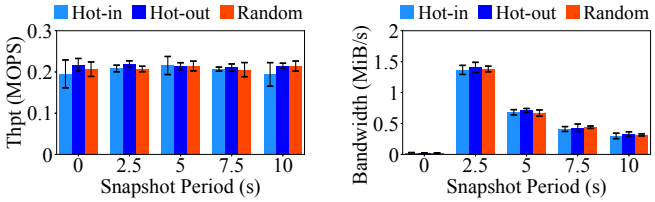


Figure 16: (Exp#8) Performance of snapshot generation, in terms of throughput (left) and control-plane bandwidth (right).

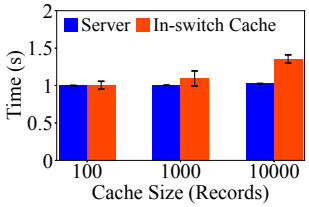


Figure 17: (Exp#9) Crash recovery time.

We also evaluate all schemes when the value size increases to 256 bytes (i.e., exceeding the maximum value size of 128 bytes). All schemes achieve similar throughput of ≈0.7 MOPS (not shown in figures), as both NetCache and FarReach directly forward all records to the servers as in NoCache.

(Exp#7) Impact of key popularity changes. Finally, we consider dynamic key popularity patterns, in which the access frequency of a specific key may change over time, while the previous experiments thus far focus on a static key popularity pattern. We consider three dynamic patterns as used in prior work [12], [14]: (i) *hot-in*, which periodically moves the 200 coldest keys to the highest key popularity ranks and decreases the ranks of other keys accordingly; (ii) *hot-out*, which periodically moves the 200 hottest keys to the lowest key popularity ranks and increases the ranks of other keys accordingly; and (iii) *random*, which randomly replaces 200 keys of the top 10,000 hottest keys with coldest keys. As the dynamic patterns will trigger cache management decisions and hence change the system state, we cannot simulate multiple servers by server rotations as in prior experiments. Instead, we evaluate the performance on the two physical servers, each running a RocksDB instance. For each dynamic pattern, we run each scheme for 70 s, and change the key popularity ranks based on each dynamic pattern every 10 s. We measure the instantaneous throughput every 1 s, and evaluate the average throughput over the entire 70 s.

Figure 15 shows that FarReach increases the average throughput of NoCache and NetCache by at least 59.4% under different dynamic patterns. We also run each scheme for 70 s without any key popularity change (i.e., static), and FarReach has similar throughput gains as in the dynamic patterns. The reason is that FarReach quickly reacts to key popularity changes (typically within 1 s from our measurement), so it maintains the cache hit rate and hence the average throughput. Note that the throughput is smaller than in prior experiments as we use fewer servers, yet our emphasis here is to examine the adaptiveness of FarReach to key popularity changes rather than absolute performance.

D. Snapshot Generation and Crash Recovery

(Exp#8) Performance of snapshot generation. We vary the period of snapshot generation to evaluate the throughput and control-plane bandwidth of FarReach on synthetic workloads. We focus on the results under dynamic patterns, in which the bandwidth costs of both snapshot generation and cache management are included. We observe similar results under the static pattern and omit them for brevity.

Figure 16 shows both the throughput and control-plane bandwidth of FarReach versus the snapshot period; if the snapshot period is zero, it means that snapshot generation is disabled. FarReach keeps its throughput at about 0.2 MOPS for various snapshot periods under different dynamic patterns, implying that snapshot generation has limited impact on throughput. When snapshot generation is disabled (i.e., the snapshot period is zero), FarReach only incurs about 0.03 MiB/s of control-plane bandwidth, since it only triggers cache management decisions for new hot records and avoids sending duplicate records to the controller (§III-B). When snapshot generation is enabled and as the snapshot period increases from 2.5 s to 10 s, the control-plane bandwidth of FarReach decreases from 1.41 MiB/s to 0.33 MiB/s, which is far smaller than the maximum bandwidth of the controller (i.e., 40 Gbps).

(Exp#9) Crash recovery time. We evaluate the crash recovery time of FarReach under a switch failure for various in-switch cache sizes. Specifically, for a given in-switch cache size, we first run the synthetic workload under the static pattern with 16 simulated servers. We manually kill the in-switch cache and the switch OS to mimic a switch failure. We then trigger zero-loss crash recovery (§III-D), which applies a replay-based approach to update the servers and recover the in-switch cache. For multiple servers, we take the average time of updating a server as the server-side recovery time.

Figure 17 shows that the time of updating a server in FarReach stays at about 1 s as the cache size increases, as FarReach only replays a limited number of writes partitioned into each server, while the majority of time is spent on collecting client-side preserved records and the control-plane in-switch snapshot. The time to recover the in-switch cache in FarReach increases from 1 s to 1.35 s as the cache size increases from 100 records to 10,000 records, as FarReach needs to admit more records from the latest snapshot under a larger cache size. Overall, the crash recovery time is within 2.35 s for various in-switch cache sizes.

E. Performance of DistReach

We evaluate DistReach based on software simulation (Exp#10-#12) and hardware evaluation (Exp#13).

(Exp#10) Throughput analysis of distributed in-switch caching. We first evaluate the throughput of distributed in-switch caching using YCSB workloads. Figure 18 shows that in the Insert workload (which is non-skewed), all schemes have similar throughput. In workloads A, B, and F, all of which are skewed and mixed with reads and writes, DistReach increases the throughput of DistNoCache by 133.1%, 126.7%, and 126.7%, respectively, and that of DistCache by 90.0%, 41.8%, and 78.0%, respectively. In the read-only workload C,

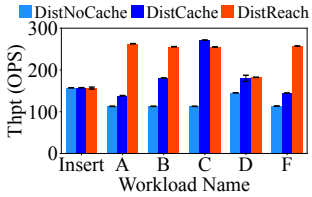


Figure 18: (Exp#10) Throughput analysis of distributed in-switch caching.

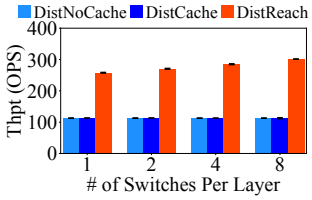


Figure 19: (Exp#11) Impact of number of switches per layer.

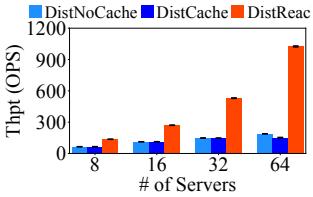


Figure 20: (Exp#12) Impact of number of servers on distributed in-switch caching.

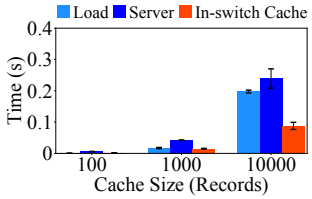


Figure 21: (Exp#13) Crash recovery time of DistReach versus cache size.

DistReach has a throughput gain of 2.3× over DistNoCache, but has slightly lower throughput than DistCache. The reason is that both DistReach and DistCache absorb reads by in-switch caching, while DistReach accesses the servers with conservative reads until the newly admitted records are updated as "latest" by read responses. Note that conservative reads have limited impact in our hardware testbed (§VI-B), as the Tofino switch can quickly update the cached records without the simulation overhead of software switches. In the read-latest workload D with 95% reads and 5% writes, DistReach has similar throughput as DistCache; while DistReach absorbs writes, it also incurs slight overhead in conservative reads. Both DistReach and DistCache increase the throughput of DistNoCache by 23.8% due to cache hits.

(Exp#11) Impact of number of switches per layer. We vary the number of switches in each layer from one to eight. We fix the number of servers as 16 and uniformly connect them to different leaf switches (e.g., each leaf switch is connected with two servers if the leaf layer has eight switches). We consider the default write-only workloads in §VI-C.

Figure 19 shows that the throughput of DistReach increases by 16.9% when the number of switches per layer increases from one to eight, since more switches provide larger cache space. The increase is marginal (instead of being linear with the number of switches), as in-switch caching is not the bottleneck and a small cache can sufficiently balance the server-side load [11]. Nevertheless, DistReach still greatly outperforms the baselines. Under eight switches per layer, the throughput gain of DistReach is 2.7× those of DistNoCache and DistCache. We emphasize that the primary objective of DistReach is to reduce the reliability maintenance overhead of FarReach instead of further improving the performance of FarReach.

(Exp#12) Impact of number of servers on distributed in-switch caching. We additionally consider different numbers of servers under the default write-only workload. We vary the number of servers from 8 to 64. The servers are uniformly connected to leaf switches as in Exp#11.

Figure 20 shows that the throughput of all schemes increases with the number of servers as more servers process requests. The throughput gain of DistReach is 2.1-5.4× and 2.1-6.8× that of DistNoCache and DistCache, respectively.

(Exp#13) Crash recovery time of DistReach versus cache size. We evaluate the crash recovery time of DistReach on our Tofino-switch testbed by varying the in-switch cache size. Due to the lack of Tofino switches, we use our only Tofino switch to simulate DistReach's recovery of a switch failure under a spine-leaf topology. Specifically, after pre-loading the 10,000 hottest records into the switch, the controller first treats the switch as the available switch to load the cached records. It also notifies the servers to load the cached records from the available switch. We then clear the in-switch cache by evicting all records from the switch to simulate a switch failure. Finally, after the servers persist the loaded records, the controller treats the switch as the recovered switch to admit the loaded records. We measure the time of loading records, updating servers, and admitting records as the crash recovery time.

Figure 21 shows that DistReach has very low crash recovery time. Under a cache size of 10,000 records, DistReach takes 0.20 s to load cached records, 0.24 s to write loaded records to servers, and 0.09 s to admit them to the switch. Thus, the total crash recovery time is 0.53 s. Compared with FarReach's crash recovery time (§VI-D), DistReach reduces the total crash recovery time of FarReach by 77.4% due to switch-based replication, without collecting client-side preserved records and the latest control-plane in-switch snapshot.

F. Switch Deployment

(Exp#14) Switch resource usage. We compile all schemes into the same Tofino switch chipset [18] to evaluate switch resource usage. For DistNoCache and DistCache, we evaluate per-switch resource usage. For DistReach, we focus on the resource usage of each leaf switch, which has higher resource demands than each spine switch as it also maintains a Count-Min sketch. We focus on the following metrics: SRAM consumption (with up to 768 KiB for stateful information and 512 KiB for match-action tables per stage), the number of stages (12 stages in total), actions, and ALUs (at most 4 stateful ALUs per stage) for in-switch computation, and the packet header vector (PHV) size (768 bytes in total) for cross-stage communication.

Tables II and III show the results of the ingress and egress pipelines, respectively. The PHV size of each scheme remains the same for both ingress and egress pipelines, as the Tofino switch chipset processes the same PHV parsed from packet headers across stages and pipelines. NoCache and DistNoCache have the smallest hardware resource usage, as they only support basic network functions (e.g., L2/L3 forwarding). NetCache and FarReach have similar switch resource usage, as both of them deploy an in-switch cache that consumes SRAM to track stateful information (e.g., key-value records and cache metadata). Both schemes also maintain SRAM-based match-action tables, utilize stages, actions, and ALUs to perform in-switch computations (e.g., cache lookups and updates), and use the PHV size to transmit each request across different stages. DistCache and DistReach have similar switch resource usage for the same reasons. Compared with NoCache and DistNoCache, which

Table II: (Exp#14) Ingress switch resource usage (percentages in brackets are fractions of total resource usage).

	SRAM (KiB)	# stages	# actions	# ALUs	PHV size (bytes)
NoCache	112 (0.73%)	3 (25%)	4 (nil)	0 (0%)	134 (17.45%)
NetCache	1248 (8.13%)	8 (66.67%)	20 (nil)	0 (0%)	528 (68.75%)
FarReach	1280 (8.3%)	6 (50%)	27 (nil)	0 (0%)	499 (64.97%)
DistNoCache	112 (0.73%)	3 (25%)	4 (nil)	0 (0%)	134 (17.45%)
DistCache	1280 (8.3%)	6 (50%)	22 (nil)	0 (0%)	523 (68.10%)
DistReach	1264 (8.2%)	10 (83.33%)	30 (nil)	0 (0%)	501 (65.23%)

Table III: (Exp#14) Egress switch resource usage (percentages in brackets are fractions of total resource usage).

	SRAM (KiB)	# stages	# actions	# ALUs	PHV size (bytes)
NoCache	256 (1.67%)	2 (16.67%)	3 (nil)	0 (0%)	134 (17.45%)
NetCache	7856 (51.15%)	12 (100%)	184 (nil)	45 (93.75%)	528 (68.75%)
FarReach	8080 (52.6%)	12 (100%)	201 (nil)	45 (93.75%)	499 (64.97%)
DistNoCache	256 (1.67%)	2 (16.67%)	3 (nil)	0 (0%)	134 (17.45%)
DistCache	7824 (50.97%)	12 (100%)	184 (nil)	45 (93.75%)	523 (68.10%)
DistReach	8032 (52.3%)	12 (100%)	182 (nil)	44 (91.67%)	501 (65.23%)

use resources for basic network functions, the in-switch caches consume most resources for caching records. Although the in-switch caches use up the available stages, they do not exhaust all resources in each stage. Thus, other networking tasks (e.g., congestion control) can still leverage the remaining resources to share the stages with in-switch caching.

G. Additional Experiments

We conduct additional experiments on distributed in-switch caching. We summarize the results (see the supplementary file) as follows:

- We vary the write ratio and key distribution of the synthetic workload (§VI-C). The throughput gain of DistReach (up to 2.4×) increases with the write ratio and Zipf constant.
- We introduce key popularity changes as in Exp#7. The throughput gain of DistReach holds 2.4× under both static and dynamic patterns.
- We vary the number of switch layers to introduce multiple switch failures, while the crash recovery time of DistReach keeps limited (e.g., 0.96 s under eight switch layers).

VII. Related Work

In-switch caching and storage management. Several in-switch caching designs have been proposed. SwitchKV [12] caches hot keys in a software switch, which forwards reads of cached keys to in-memory cache nodes for value access. IncBricks [13] caches records in general-purpose network accelerators and implements packet parsing in programmable switches to serve reads of cached keys. NetCache [14] implements a packet processing pipeline for an in-switch read cache based on switch ASICs. DistCache [15] implements distributed in-network caching across multiple racks. The above studies primarily target read-intensive workloads with write-through caching, which incurs significant overhead under write-intensive workloads (§VI). PKache [37] implements in-switch caching with limited associativity and provides a general framework with different cache management policies, yet it does not address write-back caching.

Aside from caching, some studies use programmable switches for efficient storage management. AppSwitch [38] offloads hash-based routing to software switches, and its control plane dynamically updates routing rules based on server loads for load balancing. NetChain [39] stores records in programmable switches for coordinating switch-based chain replication, yet it does not address distributed in-switch caching. TurboKV [40] and Pegasus [41] keep in-switch directory information to speed up replication of in-memory key-value stores. Concordia [42] tracks the locations of host-side cache copies in switches for efficient cache coherence. Mind [43] maintains in-switch memory management (e.g., address translation and cache coherence) for efficient and transparent rack-scale memory disaggregation. RedPlane [44] tracks networking flow states in switches and makes periodic snapshots for write-intensive workloads with relaxed consistency (allowing the loss of states). Switcharoo [45] implements lookups and insertions of cuckoo hash tables in switches to reduce control plane overhead, yet it does not address in-switch cache management issues. P4LRU [46] implements an LRU cache in switches to track recently-accessed networking flow states and indexes of key-value storage, yet it cannot track the values of key-value storage to absorb requests due to limited switch stages. Such systems do not consider in-switch caching for server-side key-value storage.

Write-back caching. Prior studies propose various write-back caching policies. DEFER [47] improves the reliability of write-back caching by replication and logging. FlashTier [48] deploys a write-back flash cache and ensures consistency by storing both cached data and mapping details durably in flash. Some studies propose write-back caching policies with different reliability guarantees. Examples include: (i) ordered and journaled policies [49] that provide point-in-time consistency, (ii) write-back flush and persist policies [50] that use write barriers for durable and consistent caching, and (iii) client-side buffered write policies [51] that ensure durability by replication with read-after-write consistency. However, programmable switches have restricted programming requirements and limited hardware resources for implementing such policies. Enabling new write-back caching policies with stronger reliability guarantees is our future work.

VIII. Conclusion

We study the in-switch write-back caching problem. We first propose FarReach, which targets single-switch deployment and forms a fast, available, and reliable in-switch write-back caching framework for load-balanced key-value stores in modern data centers under skewed write-intensive workloads. It incorporates new co-designs of control and data planes for cache admission and eviction under a write-back policy. In particular, FarReach pays special attention to crash-consistent snapshot generation and zero-loss crash recovery, so as to protect against data loss under switch failures. We further propose DistReach, which extends FarReach to support distributed in-switch caching deployment. The novelty of DistReach is to

leverage switch-based replication to significantly reduce the reliability maintenance overhead of FarReach. Evaluation under YCSB and synthetic workloads demonstrates the performance benefits of both FarReach and DistReach under skewed write-intensive workloads.

REFERENCES

[1] S. Sheng, H. Puyang, Q. Huang, L. Tang, and P. P. Lee, "FarReach: Write-back caching in programmable switches," in *Proc. of USENIX ATC*, 2023.

[2] J. Yang, Y. Yue, and K. V. Rashmi, "A large scale analysis of hundreds of in-memory cache clusters at Twitter," in *Proc. of USENIX OSDI*, 2020.

[3] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling memcache at Facebook," in *Proc. of USENIX NSDI*, 2013.

[4] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *Proc. of ACM SIGMETRICS*, 2012.

[5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: a distributed storage system for structured data," *ACM Trans. on Computer Systems*, vol. 26, no. 2, pp. 1–26, 2008.

[6] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *ACM SIGOPS operating systems review*, vol. 41, no. 6, pp. 205–220, 2007.

[7] Z. Cao, S. Dong, S. Vemuri, and D. H. Du, "Characterizing, modeling, and benchmarking RocksDB key-value workloads at facebook," in *Proc. of USENIX FAST*, 2020.

[8] Q. Huang, H. Gudmundsdottir, Y. Vigfusson, D. A. Freedman, K. Birman, and R. van Renesse, "Characterizing load imbalance in real-world networked caches," in *Proc. of ACM SIGCOMM HotNets Workshop*, 2014.

[9] J. Jung, B. Krishnamurthy, and M. Rabinovich, "Flash crowds and denial of service attacks: Characterization and implications for cdns and web sites," in *Proc. of WWW*, 2002.

[10] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: fast programmable match-action processing in hardware for SDN," in *Proc. of ACM SIGCOMM*, 2013.

[11] B. Fan, H. Lim, D. G. Andersen, and M. Kaminsky, "Small cache, big effect: Provable load balancing for randomly partitioned cluster services," in *Proc. of ACM SOCC*, 2011.

[12] X. Li, R. Sethi, M. Kaminsky, D. G. Andersen, and M. J. Freedman, "Be fast, cheap and in control with SwitchKV," in *Proc. of USENIX NSDI*, 2016.

[13] M. Liu, L. Luo, J. Nelson, L. Ceze, A. Krishnamurthy, and K. Atreya, "IncBricks: Toward in-network computation with an in-network cache," in *Proc. of ACM ASPLOS*, 2017.

[14] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "NetCache: balancing key-value stores with fast in-network caching," in *Proc. of ACM SOSP*, 2017.

[15] Z. Liu, Z. Bai, Z. Liu, X. Li, C. Kim, V. Braverman, X. Jin, and I. Stoica, "DistCache: provable load balancing for large-scale storage systems with distributed caching," in *Proc. of USENIX FAST*, 2019.

[16] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "SilkRoad: making stateful layer-4 load balancing fast and cheap using switching ASICs," in *Proc. of ACM SIGCOMM*, 2017.

[17] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[18] Tofino, https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series/tofino.html.

[19] P4 switch behavioral model, https://github.com/p4lang/behavioral-model.

[20] YCSB, https://github.com/brianfrankcooper/YCSB/.

[21] Mininet, https://mininet.org/.

[22] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik, "High-availability algorithms for distributed stream processing," in *Proc. of IEEE ICDE*, 2005.

[23] LevelDB, https://github.com/google/leveldb/.

[24] RocksDB, https://github.com/facebook/rocksdb/.

[25] L. Lu, T. S. Pillai, H. Gopalakrishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "WiscKey: separating keys from values in SSD-conscious storage," *ACM Trans. on Storage*, vol. 13, no. 1, pp. 1–28, 2017.

[26] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[27] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[28] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *Proc. of ACM STOC*, 1997, pp. 654–663.

[29] J. Tate, P. Beck, P. Clemens, S. Freitas, J. Gatz, M. Girola, J. Gmitter, H. Mueller, R. O'Hanlon, V. Para *et al.*, *IBM and Cisco: together for a world class data center*. IBM Redbooks, 2013.

[30] N. Kang, M. Ghobadi, J. Reumann, A. Shraer, and J. Rexford, "Efficient traffic splitting on commodity switches," in *Proc. of ACM CoNEXT*, 2015.

[31] R. Van Renesse and F. B. Schneider, "Chain replication for supporting high throughput and availability." in *Proc. of USENIX OSDI*, 2004.

[32] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (LSM-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[33] O. Balmau, D. Didona, R. Guerraoui, W. Zwaenepoel, H. Yuan, A. Arora, K. Gupta, and P. Konka, "TRIAD: Creating synergies between memory, disk and log in log structured key-value stores," in *Proc. of USENIX ATC*, 2017.

[34] A. Papagiannis, G. Saloustros, P. González-Férez, and A. Bilas, "Tucana: Design and implementation of a fast and efficient scale-up key-value store," in *Proc. of USENIX ATC*, 2016.

[35] D. Didona and W. Zwaenepoel, "Size-aware sharding for improving tail latencies in in-memory key-value stores," in *Proc. of USENIX NSDI*, 2019, pp. 79–94.

[36] Y. Cheng, A. Gupta, and A. R. Butt, "An in-memory object caching framework with adaptive load balancing," in *Proc. of ACM EuroSys*, 2015.

[37] R. Friedman, O. Goaz, and D. Hovav, "Limited associativity caching in the data plane," *CoRR*, vol. abs/2203.04803, 2022.

[38] E. Cidon, S. Choi, S. Katti, and N. McKeown, "AppSwitch: Application-layer load balancing within a software switch," in *Proc. of APNet*, 2017.

[39] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica, "NetChain: scale-free sub-RTT coordination," in *Proc. of USENIX NSDI*, 2018.

[40] H. Eldakiky, D. H. Du, and E. Ramadan, "TurboKV: scaling up the performance of distributed key-value stores with in-switch coordination," *CoRR*, vol. abs/2010.14931, 2020.

[41] J. Li, J. Nelson, E. Michael, X. Jin, and D. R. K. Ports, "Pegasus: tolerating skewed workloads in distributed storage with in-network coherence directories," in *Proc. of USENIX OSDI*, 2020.

[42] Q. Wang, Y. Lu, E. Xu, J. Li, Y. Chen, and J. Shu, "Concordia: distributed shared memory with in-network cache coherence," in *Proc. of USENIX FAST*, 2021.

[43] S.-s. Lee, Y. Yu, Y. Tang, A. Khandelwal, L. Zhong, and A. Bhattacharjee, "Mind: In-network memory management for disaggregated data centers," in *Proc. of ACM SOSP*, 2021.

[44] D. Kim, J. Nelson, D. R. Ports, V. Sekar, and S. Seshan, "RedPlane: Enabling fault-tolerant stateful in-switch applications," in *Proc. of ACM SIGCOMM*, 2021.

[45] T. Caiazzi, M. Scazzariello, and M. Chiesa, "Millions of low-latency state insertions on ASIC switches," *Proceedings of the ACM on Networking*, vol. 1, no. CoNEXT3, pp. 1–23, 2023.

[46] Y. Zhao, W. Liu, F. Dong, T. Yang, Y. Li, K. Yang, Z. Liu, Z. Jia, and Y. Yang, "P4LRU: towards an LRU cache entirely in programmable data plane," in *Proc. of ACM SIGCOMM*, 2023.

[47] S. Narasimhan, S. Sohoni, and Y. Hu, "A log-based write-back mechanism for cooperative caching," in *Proc. of IEEE IPDPS*, 2003.

[48] M. Saxena, M. M. Swift, and Y. Zhang, "Flashtier: a lightweight, consistent and durable storage cache," in *Proc. of ACM EuroSys*, 2012.

[49] R. Koller, L. Marmol, R. Rangaswami, S. Sundararaman, N. Talagala, and M. Zhao, "Write policies for host-side flash caches," in *Proc. of USENIX FAST*, 2013.

[50] D. Qin, A. D. Brown, and A. Goel, "Reliable writeback for client-side flash caches," in *Proc. of USENIX ATC*, 2014.

[51] S. Ghandeharizadeh and H. Nguyen, "Design, implementation, and evaluation of write-back policy with cache augmented data stores," *Proc. of the VLDB Endowment*, vol. 12, no. 8, pp. 836–849, 2019.