FarReach: Write-back Caching in Programmable Switches

Siyuan Sheng and Huancheng Puyang, *The Chinese University of Hong Kong;* Qun Huang, *Peking University;* Lu Tang, *Xiamen University;* Patrick P. C. Lee, *The Chinese University of Hong Kong*

https://www.usenix.org/conference/atc23/presentation/sheng

This paper is included in the Proceedings of the 2023 USENIX Annual Technical Conference.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-35-9

Open access to the Proceedings of the 2023 USENIX Annual Technical Conference is sponsored by

جامعة الملك عبدالله للعلوم والتقنية
King Abdullah University of Science and Technology

FarReach: Write-back Caching in Programmable Switches

Siyuan Sheng[1], Huancheng Puyang[1], Qun Huang[2], Lu Tang[3], and Patrick P. C. Lee[1]
[1]*The Chinese University of Hong Kong* [2]*Peking University* [3]*Xiamen University*

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 direction to absorb frequent writes for high write performance. However, enabling in-switch write-back caching is non-trivial, as it not only is challenged by the strict programming rules and limited stateful memory of programmable switches, but also necessitates reliable protection against data loss due to switch failures. We 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. Experiments 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.

1 Introduction

Key-value stores have been widely deployed in modern data centers to manage structured data (in units of *records*) for data-intensive applications, such as social networking [1, 33, 41], web indexing [7], and e-commerce [11]. Practical key-value storage workloads are traditionally read-intensive (e.g., up to a read-to-write ratio of 30:1 at Facebook's Memcached [1]). Recent field studies of production key-value stores show the dominance of *write-intensive workloads*; for example, more than 20% of Twitter's Twemcache clusters have more writes than reads [41], and the AI/machine-learning services at Facebook's RocksDB production have 92.5% of read-modify-writes [6]. Also, write-intensive workloads are *skewed*; for example, 25% of frequently accessed (i.e., *hot*) records dominate in the write workloads at Twitter's Twemcache clusters [41].

Enabling high write performance for key-value stores in data centers is challenging. Write requests issued 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. In particular, if the server is overloaded, I/O requests will have long queuing delays or even be dropped. Also, in distributed key-value stores that span multiple servers, a small portion of servers may be bottlenecked by substantial requests for hot records under skewed workloads, thereby leading to load imbalance [17, 21].

Programmable switches [5] 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 inherently intercept the I/O requests for all servers within the rack and provide stateful memory that can be programmed to cache frequently accessed records. For each request issued by a client, the switch can read or write any of its cached records and directly respond to the client, thereby eliminating the long RTT to access the server-side record. It is also proven that load balancing is achievable by keeping only $O(N \log N)$ records, where N is the number of servers [14]. Recent studies have demonstrated the effectiveness of load-balanced in-switch caching [20, 27–29] for high throughput and sub-RTT latencies. However, existing in-switch caching approaches [20, 27–29] target read-intensive workloads and implement *write-through* caching (i.e., write requests update records both in the in-switch cache and the server side). Thus, they do not provide write performance gains 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 is subject to several challenges. First, in-switch write-back caching raises an 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 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) [5, 31]. 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 [20, 29], the control-plane interaction between the controller and programmable switches can incur long delays and slow down the data-plane I/O performance. 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 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, such that it offloads 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 that 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. To the best of our knowledge, this is the first work that specifically addresses the availability and fault tolerance issues of in-switch caching.

We implement FarReach and compile the in-switch cache prototype (written in P4 [4]) into the Tofino switch chipset [39]. We evaluate FarReach with YCSB [42] and synthetic workloads. Compared with NetCache [20], a state-of-the-art in-switch caching framework that targets read-intensive workloads and implements 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.

We now release the source code of our FarReach prototype at **http://adslab.cse.cuhk.edu.hk/software/farreach**.

2 Background and Motivation

2.1 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 a stringent timing requirement for line-rate forwarding. It contains multiple *ingress* and *egress* pipelines. When a packet arrives at the switch through an ingress port, the packet first enters the corresponding ingress pipeline, which specifies an egress port. Then the *traffic manager*, which interconnects between the ingress and egress pipelines, 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 [5]. 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

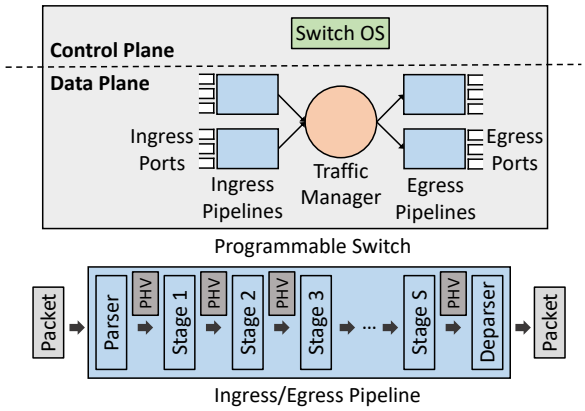


Figure 1: Programmable switch architecture.

stages with multiple *match-action tables* each. Each stage also keeps a limited amount of SRAM, composed of tens of memory blocks, for tracking stateful information that is accessible by the match-action tables. A match-action table can use an ALU to perform arithmetic or logical operations and store the results into 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, while the match-action rules can be configured by the switch OS. A match-action table can also use a special kind of ALU, called stateful ALU, to store the results into on-chip memory. To fulfill the stringent timing requirement, the memory blocks associated with a stage cannot be accessed from other stages, while the processing of a packet within a stage can only access a limited number of memory blocks associated with the stage and each memory block can only be 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.

2.2 Challenges

Write caching policies can be classified into *write-through* and *write-back*. Write-through synchronously updates the records both in the cache and on the server side; in contrast, write-back (a.k.a. delayed-write) updates only the records in the cache, and later reflects the updates on the server side. Existing in-switch caches [20, 27–29] mainly implement write-through caching. In this work, we focus on write-back caching, as it improves the write performance over write-through caching by delaying server-side updates. However, managing write-back caching is non-trivial, and is subject to three unique challenges in programmable switches.

Performance challenge. Since 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 only has a limited number of stages and stateful ALUs) [5], it is necessary to offload switch-level cache management (including cache admission and eviction) to a

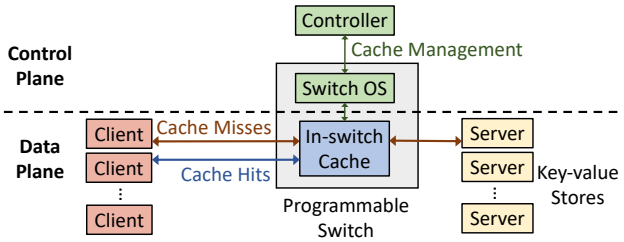


Figure 2: FarReach's architecture.

centralized controller [20, 29], 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 the 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, the outdated records may be returned to the client. Such an issue does not exist in write-through caching [20, 29], 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, but incurs high overhead. Also, the controller is not on the packet forwarding path and has no view about 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 and may have their updates to the server-side storage delayed. If the switch crashes, all 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 [20, 29].

3 FarReach Design

3.1 Design Overview

Architecture. FarReach is a fast, available, and reliable in-switch write-back cache architecture for improving the I/O performance and load balancing of server-side key-value stores. Figure 2 shows FarReach's architecture, 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. Recall that the controller has no view about the data plane (§2.2). Thus, the cache management decisions are triggered by the switches (in the data plane) based on the workload patterns.

Goals. FarReach's core idea is a careful co-design of the control and data planes. Table 1 summarizes our design features. FarReach aims for three design goals:

- *Fast access* (§3.2). FarReach supports non-blocking cache admission for admitting hot records into the in-switch cache, so as to achieve high write performance. It also ensures atomicity in cache admission under the multi-pipeline setting of programmable switches.
- *Availability* (§3.3). FarReach ensures that any latest record that is evicted from the in-switch cache remains available to clients.
- *Reliability* (§3.4). FarReach protects against data loss during switch failures. It uses a crash-consistent snapshot generation algorithm for making snapshots of the in-switch cache state. It also ensures atomicity of snapshot generation in the multi-pipeline setting. It further couples snapshot generation with upstream backup [18] to achieve zero-loss crash recovery.

Table 1: Summary of design features of FarReach.

Design features	Design details
Non-blocking cache admission (§3.2)	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 (§3.3)	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 (§3.4)	FarReach reports original cached records to the controller. It recirculates writes for atomicity, and exploits client-side record preservation for zero-loss recovery.

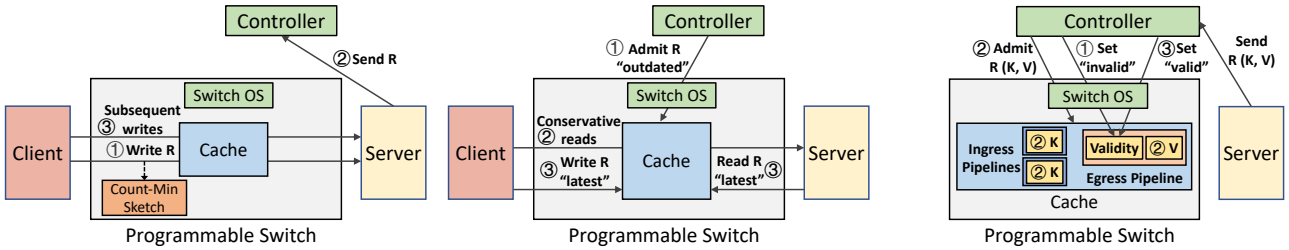
Design assumptions. FarReach currently supports a fixed key length of 16 bytes and a variable value length of up to 128 bytes due to limited switch resources; the same constraint is also assumed in NetCache [20] and DistCache [29]. Thus, FarReach is suitable for workloads dominated by small records (e.g., ZippyDB and UP2X in Facebook's RocksDB production [6]). For large records, FarReach simply relays them between clients and servers without caching.

FarReach currently does not support range queries, since programmable switches cannot feasibly maintain sorted structures with the memory access limitations (§2.1) 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 [25, 30, 37].

3.2 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 (§2.2). 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 (§2.2), the switch may receive subsequent



(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.

writes for the same key before the record from the first write is admitted to the cache. In this case, 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 [20] 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 [10] 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 (§3.3), 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. It triggers the controller to admit the hot record into the in-switch cache, and also tracks 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 of 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. Thus, the server now keeps 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 keeping limited control-plane bandwidth usage (e.g., up to 1.41 MiB/s; see §5.4). This is in contrast to NetCache [20], in which a switch sends records to the controller for cache admission and needs an in-switch Bloom Filter [3] to avoid duplicate submission; FarReach removes the need of maintaining an in-switch Bloom Filter and hence saves 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 the requests for the same key can arrive from different ingress pipelines. Without the atomicity of cache admission, the 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 insight is that although the requests for the same key can enter a switch from different ingress pipelines, FarReach can forward them to the same egress pipeline corresponding to 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. The server-side bottleneck is shown in our evaluation, where the system throughput is up to 12.1 MB/s under 128 simulated servers (§5.2), significantly lower than the maximum throughput 3.2 Tbps of a two-pipeline Tofino switch [39]. 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".

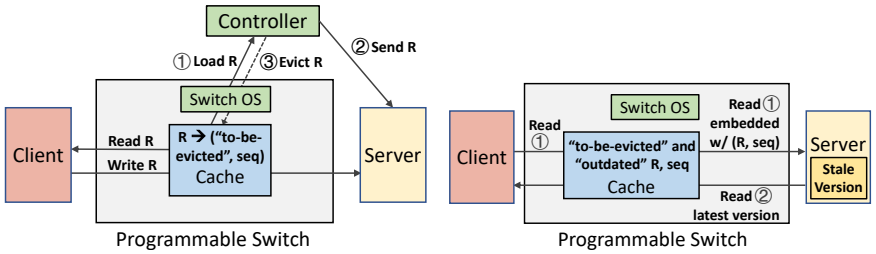
3.3 Available Cache Eviction

Problem of cache eviction. If the in-switch cache is full for cache admission, 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 (§3.2). It then triggers the controller to perform cache eviction on the selected record. Under the write-back policy, the evicted record may also be the latest record and has not yet been updated in the server. It is critical to keep any latest record to be evicted available during cache eviction. To achieve this goal, the controller needs to synchronize the views of both the switch and the server on the evicted record during cache eviction, especially when there also exist read/write requests for the evicted record. However, the controller is constrained 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, whose workflow is shown in Figure 5(a). Our idea is to associate additional metadata with each cached record in the in-switch cache, so as to maintain the availability of any evicted record, while incurring limited 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 (§3.2)), the cache returns R to the client; otherwise, if R is "outdated" (i.e., it has been updated), FarReach forwards the read request of R to the server, which holds the latest record. Thus, we ensure that any evicted cached record that is also the latest record 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 no matter with the view of 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 the request of storing a record from two possible paths: (i) the eviction of a record from the cache and (ii) the write request of the record 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 (§3.2). As programmable switches can provide atomicity and serialize packets in a single pipeline (§3.2), 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 will be discarded.

Handling packet loss. Packet loss in switch-to-server transmissions can break the availability of cache eviction. To elaborate, recall that an in-switch record can be the latest version under the write-back policy. During cache eviction, FarReach



(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.

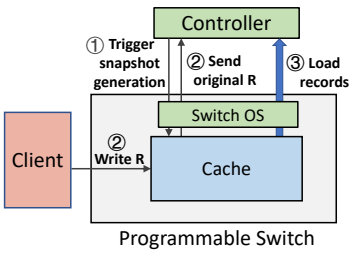


Figure 6: Crash-consistent snapshot generation in FarReach. If the switch receives the first write to a cached record *R* during snapshot generation, it forwards the original *R* to the controller before *R* is updated.

forwards the write request to a "to-be-evicted" record to the server and marks the evicted record as "outdated". If the write request is lost during its transmission (e.g., due to server-side congestion or packet corruption), the server is not updated with the latest version, but still keeps the stale version. As the in-switch cache marks the evicted record as "outdated", it forwards all subsequent reads to the server and receives the stale version. Note that 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 changing 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 that can be used 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.

3.4 Crash-consistent Snapshot Generation

We now address the reliability challenge (§2.2) through a consistency model that incurs zero data loss after switch failures. At a high level, FarReach periodically generates snapshots to protect against data loss of in-switch cached records. It also lets each client preserve the cached records generated after the latest snapshot for recovery. Note that the uncached records are protected by server-side persistent key-value stores (§3.1).

Problem of snapshot generation. Since the in-switch cache keeps the latest records under the write-back policy, we need to protect against data loss in switch failures. 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 switch 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. Note that the snapshot overhead is limited for the controller, as the controller 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. The final snapshot will become inconsistent with the in-switch cache state at the snapshot point. Blocking cache updates during snapshot generation can avoid such inconsistencies, yet it also degrades the 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 the backups for all original cached records that are to be overwritten. At the end of snapshot generation, the controller replaces the overwritten cached records by 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 limited original cached records to the controller (without the need to send the cached record for the subsequent writes after the first write). Thus, the bandwidth overhead in the controller is limited.

Based on the insight, FarReach generates a crash-consistent snapshot in a two-phase manner (i.e., triggering snapshot gen-

eration 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 traversed write request to identify whether the write request 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). Note that 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, which has already been loaded. Once the controller loads all cached records, it notifies the cache about the completion of snapshot generation (i.e., the cache no longer needs to monitor the writes to cached records), 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 (§3.3), 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 the write requests of a record can arrive from multiple ingress pipelines, FarReach needs to trigger snapshot generation in multiple pipelines at the same time; otherwise, the ingress pipelines may set a snapshot point at different times and generate inconsistent snapshots. We propose a coordination mechanism to support simultaneous snapshot generation in multiple ingress pipelines. Specifically, 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, and now the controller can 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 the cached records that are newly added or updated after the latest snapshot point remain unprotected and can be lost during a switch failure. Unfortunately, switches do not have external storage for keeping cached records reliably. To achieve zero-loss crash recovery, we propose *client-side record preservation* based on the idea of upstream backup [18] 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 of a cached key and receives the response from the in-switch cache, it keeps locally the value and sequence number assigned by the in-switch cache (§3.3) for the cached key. After the completion of snapshot generation at each snapshot point, the controller notifies each client with the cached keys and the 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 exploits a *replay-based approach* to achieve zero-loss crash recovery after a switch failure. It first replays the write requests of the latest cached records to update the servers for persistent storage. 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 persistent storage. After all latest records are persisted, the clients can then release their preserved records.

FarReach next recovers the in-switch cache, by replaying the cache admission for each record of the latest in-switch snapshot, and marks 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 (§3.2). Note that we do not simply 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. We can reduce the snapshot period to a smaller window for less vulnerability, at the expense of incurring larger snapshot generation overhead. Nevertheless, the snapshot generation overhead remains still limited (e.g., up to 1.41 MiB/s of control-plane bandwidth when the snapshot period is 2.5 s; see §5.4). We leave how to completely prevent data loss from client crashes as future work.

3.5 Discussion

Novelty. While FarReach borrows ideas from NetCache [20] (e.g., cache admission based on a Count-Min Sketch), it has other novel design elements: (i) non-blocking cache admission for fast access, with atomic validity control to address

the atomicity issue (§3.2); (ii) available cache eviction for the availability of records, with record embedding to handle packet loss (§3.3); and (iii) crash-consistent snapshot generation with zero-loss recovery (§3.4). Note that the last two elements are tailored for write-back caching and are not found in NetCache for 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, yet we show that the extra switch resource usage of FarReach for supporting the write-back policy is similar to that of NetCache (§5.5). Second, FarReach trades extra client-side storage capacity for zero-loss recovery. Nevertheless, since the 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).

Future work. We pose two open issues as future work. First, in addition to reducing vulnerability window from client crashes (§3.4), FarReach should collect the preserved records from all clients during crash recovery, and it may limit scalability as the number of clients increases. One possible solution is to extend FarReach with multiple switches and partition clients among them, such that FarReach can collect the preserved records through multiple switches in parallel. Second, FarReach offloads cache management (including cache admission and eviction) to a centralized controller, which may be overwhelmed by extensive cache admission and eviction decisions. One possible solution is to rate-limit admission and eviction operations to avoid overloading the control plane.

4 Implementation

We prototyped FarReach with both the control and data planes. The control plane includes the switch OS and the controller, while the data plane includes multiple clients and servers as well as the in-switch cache. All communications among different components are based on UDP with a timeout-and-retry mechanism for low-latency yet reliable transmissions.

4.1 Control Plane

We implement both the switch OS and the controller in C++, with 2.2K and 1K LoC, respectively, and compile the programs by `g++` (v5.4.0) with the `-O3` optimization. The switch OS provides interfaces for: (i) cache admission/eviction by configuring match-action tables and setting 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 provided by the switch OS and coordinates snapshot generation by communicating with the switch OS and all key-value storage servers.

4.2 Data Plane

Client implementation. We evaluate our prototype with the YCSB benchmark [42] (§5), which is written in Java. We implement a client application in Java that supports YCSB, with the 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 (§3.4).

Server implementation. We deploy RocksDB (v6.22.1) [37] in each server; RocksDB is a log-structured merge-tree (LSM-tree) persistent key-value store [34] that is suitable for write-intensive workloads. To support multiple servers, we distribute records across servers using consistent hashing [22].

In-switch cache. We implement the in-switch cache in P4 [4] and compile it into the Tofino switch chipset [39]. The cache implementation includes both ingress and egress pipelines. In each ingress/egress pipeline, 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 for egress processing. We implement a match-action table for cache lookup, which matches 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 (§3.4). As the Tofino switch currently does not support cross-pipeline recirculation, we connect the selected ingress pipeline with each of the other ingress pipelines with a physical wire, so as to recirculate the write requests from the other ingress pipelines to the selected ingress pipeline during snapshot generation in the multi-pipeline setting (§3.4). Furthermore, we pre-compute the hash results for the Count-Min Sketch in the ingress pipeline and send them to the egress pipeline by each packet header, so as to save the stages in the egress pipelines.

In each egress pipeline, we store the statistics, metadata, and cached values. In the first stage, we deploy a Count-Min Sketch and configure it with 4 rows as suggested in [20]. 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 (§3.2). 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 need to 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. Second, to assign a sequence number for each write request, we can 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 the write requests of different keys into different registers by hashing, and then increment the hashed register to assign a sequence number for each write request.

5 Evaluation

5.1 Methodology

Testbed. We conduct evaluation on a testbed composed of a 3.2 Tbps two-pipeline Tofino switch [39] and four physical machines. Each machine has four 12-core CPUs (Intel E5-2650 v4), 64 GiB DRAM, and 2 TB hard disk (HGST Ultrastar), and is connected with the switch by a 40 Gbps NIC (Intel XL710). We use two physical machines as clients and another two as key-value storage servers. We connect one client and one server with one pipeline of the switch, and connect the other machines with another pipeline.

Setup. We evaluate FarReach using both YCSB [42] and synthetic workloads (see §5.2 and §5.3, respectively). Since our testbed comprises only two servers, we exploit server rotation [20] to simulate a much larger number of servers. Specifically, let N be the number of simulated servers. Given a workload, we issue the requests to N logical partitions via consistent hashing [22] (§4). We find 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 the bottleneck to 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 add all per-partition performance to obtain the aggregate performance. By default, we simulate 16 servers, and increase the number of simulated servers for scalability evaluation (Exp#3). Note that server rotation is only applied to static workloads without the dynamics in key popularity, and we also study the impact of dynamic workloads (Exp#7).

We compare FarReach against two baselines: NoCache (i.e., no in-switch caching) and NetCache [20] (i.e., the in-switch cache that implements write-through caching). Before each experiment, we pre-load 100M records, each of which contains a 16-byte key and 128-byte value, into each server that is 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 run all experiments with 5 times, and plot the average results with the 95% confidence levels based on the Student's t-distribution.

Summary of results. We summarize the results as follows:

- Under YCSB workloads, FarReach increases the I/O throughput by up to 91% and 84% (for workload A with 50% reads and 50% writes) compared with NoCache and NetCache, respectively (Exp#1). FarReach also achieves sub-RTT latency with up to 72% reduction of average latency (Exp#2) and scales to an increasing number of servers with up to 6.6× throughput gain (Exp#3).
- Under synthetic workloads, FarReach achieves higher throughput gains over NoCache and NetCache for more write-intensive and more skewed workloads (Exp#4 and Exp#5, respectively), while maintaining similar throughput gains for different value sizes and dynamic workloads (Exp#6 and Exp#7, respectively).
- FarReach's snapshot generation incurs limited overhead on throughput and control-plane bandwidth (Exp#8). Its recovery time is within 2.35 s (Exp#9).
- FarReach incurs similar switch resource overhead as NetCache (Exp#10).

5.2 Performance under YCSB Workloads

(Exp#1) Throughput analysis. We first evaluate the end-to-end throughput using YCSB workloads, namely Load (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 (§3.1). For each workload, we generate requests with 16-byte keys and 128-byte values. The Load workload follows the uniform distribution, workload D follows the read-latest distribution, and workloads A, B, C, and F are skewed and follow the Zipf distribution with the Zipfian constant 0.99 (default in YCSB). We verify that under NoCache, the load throughput to a RocksDB instance can reach 0.06 MOPS, which is consistent with prior findings [2, 35].

Figure 7 shows that FarReach increases the throughput of NoCache by 91%, 55%, 85%, and 72% 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 84%, 20%, and 61% in workloads A, B, and F, and achieves similar throughput as 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, and hence limit the throughput of NetCache. NetCache only achieves high throughput in read-intensive workloads B and C. In the non-skewed workloads Load and D, both FarReach and NetCache have similar throughput as NoCache due to limited cache hits.

(Exp#2) Latency analysis. We next evaluate the request latencies. We focus on YCSB workload A, which is skewed and most write-intensive. In particular, we examine the trade-off between the latency and target throughput (i.e., configured by a given sending rate) as in prior studies [8, 12, 20]. We

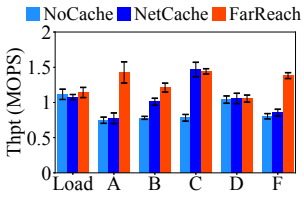


Figure 7: (Exp#1) Throughput analysis.

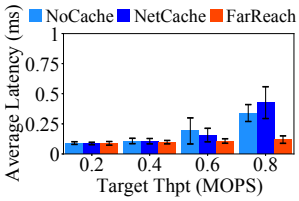


Figure 8: (Exp#2) Latency analysis.

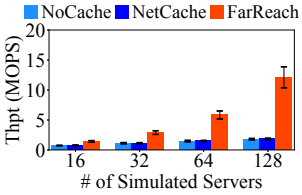


Figure 9: (Exp#3) Scalability analysis.

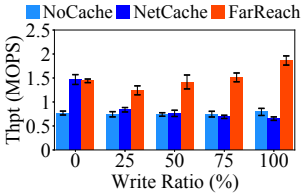


Figure 10: (Exp#4) Impact of write ratio.

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 8 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 65% and 72% 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 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 9 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.

5.3 Performance under Synthetic Workloads

We generate different synthetic workloads with YCSB for varying write ratios (over all reads and writes), key distributions, value sizes, key popularities. By default, we generate requests with 16-byte keys and 128-byte values, where the keys follow the Zipf distribution with the 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 to evaluate the throughput of different schemes. Figure 10 shows that FarReach increases the throughput of NoCache by 67-135% for different write ratios,

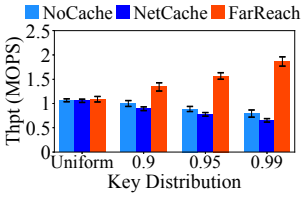


Figure 11: (Exp#5) Impact of key distribution.

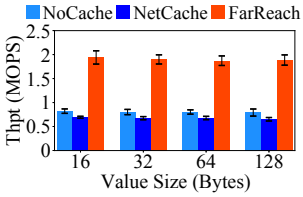


Figure 12: (Exp#6) Impact of value size.

due to load balancing in FarReach. FarReach achieves similar throughput as NetCache when the write ratio is zero (i.e., read-only requests), while increasing the throughput of NetCache by 48-186% 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% through in-switch write-back caching. Note that NetCache has slightly smaller throughput than NoCache, especially under the 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 11 shows that all schemes achieve similar throughput of ≈1 MOPS under the uniform key distribution, as most requests are from the uncached keys and will be 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 34-135% and that of NetCache by 50-186%. The throughput gain of FarReach is higher when the workload is more skewed (i.e., a larger Zipfian constant), as NoCache and NetCache becomes 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 12 shows that the throughput gains of FarReach over NoCache and NetCache remain almost the same at 2.33× across different value sizes, as the caching behavior of FarReach mainly depends on the key distribution.

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 a figure), as both NetCache and

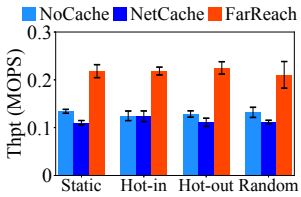


Figure 13: (Exp#7) Impact of key popularity changes.

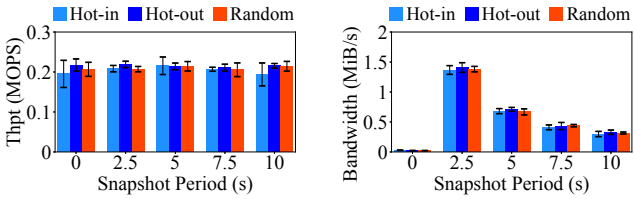


Figure 14: (Exp#8) Performance of snapshot generation, in terms of throughput (left) and control-plane bandwidth (right).

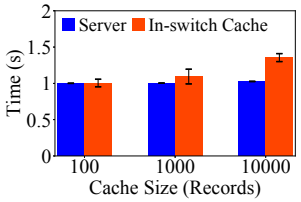


Figure 15: (Exp#9) Crash recovery time.

FarReach directly forward all records to the servers and have similar behavior as 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 [20, 27]: (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 of which runs 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 13 shows that FarReach increases the average throughput of NoCache and NetCache by at least 59% 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 the 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 that in prior experiments as we now use fewer servers, yet our emphasis here is to examine the adaptiveness of FarReach to key popularity changes rather than the absolute performance.

5.4 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, while we observe similar results under the static pattern and they are omitted for brevity.

Figure 14 shows both the throughput and control-plane bandwidth of FarReach versus the snapshot period; note that 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 a 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 (§3.2). 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. Note that the control-plane bandwidth of FarReach 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 servers simulated by server rotations. We manually kill the in-switch cache and the switch OS to mimic a switch failure. We then trigger zero-loss crash recovery (§3.4), 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 15 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 in each server, while taking the majority of time to collect client-side preserved records and 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.

5.5 Switch Deployment

(Exp#10) Switch resource usage. We compile the three schemes into the same Tofino switch chipset [39] to evaluate the switch resource usage. We focus on the following

Table 2: (Exp#10) Switch resource usage (percentages in brackets are fractions of total resource usage).

	SRAM (KiB)	# stages	# actions	# ALUs	PHV size (bytes)
NoCache	320 (2.08%)	4 (33.33%)	6 (nil)	0 (0%)	134 (17.45%)
NetCache	8800 (57.29%)	12 (100%)	69 (nil)	45 (93.75%)	528 (68.75%)
FarReach	8992 (58.54%)	12 (100%)	70 (nil)	45 (93.75%)	499 (64.97%)

metrics: SRAM consumption (with up to 768 KiB for stateful information and 512 KiB for match-action tables per stage), the numbers 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.

Table 2 shows the results. NoCache has the smallest hardware resource usage, as it only needs to 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). Also, both schemes maintain SRAM-based match-action tables, exploit the 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.

6 Related Work

In-switch caching and storage management. Several in-switch caching designs have been proposed for high-performance storage. SwitchKV [27] caches hot keys in a software switch, which forwards the reads of cached keys to the in-memory cache nodes, instead of servers, for accessing the values. IncBricks [28] caches records in general-purpose network accelerators and implements packet parsing in programmable switches to serve the reads of cached keys. NetCache [20] implements a packet processing pipeline for an in-switch read cache based on switch ASICs. DistCache [29] implements distributed in-network caching across multiple racks. The above studies target only read-intensive workloads with write-through caching, which incurs significant overhead under write-intensive workloads (§5). PKache [15] 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 [9] offloads hash-based routing to software switches, and its control plane dynamically updates the routing rules based on server loads for load balancing. NetChain [19] stores records in programmable switches for the coordination of the switch-based chain replication model. TurboKV [13] and Pegasus [26] keep in-switch directory information to speed up the replication protocol of in-memory key-value stores. Concordia [40] tracks the locations of host-side cache copies in programmable switches for efficient cache coherence. Mind [24] maintains in-switch memory management (e.g., address translation and cache coherence) for efficient and transparent rack-scale memory disaggregation. Such systems do not consider in-switch caching for server-side key-value storage.

Write-back caching. Prior studies propose write-back caching policies. DEFER [32] improves the reliability of write-back caching by replication and logging. FlashTier [38] 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 [23] that provide point-in-time consistency, (ii) write-back flush and persist policies [36] that use write barriers for durable and consistent caching, and (iii) client-side buffered write policies [16] that ensure durability by replication with read-after-write consistency guarantees. However, programmable switches have restricted programming requirements and limited hardware resources for implementing such policies. How to enable new write-back caching policies with stronger reliability guarantees is our future work.

7 Conclusion

FarReach is 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. Evaluation under YCSB and synthetic workloads demonstrates the performance benefits of FarReach under skewed write-intensive workloads.

Acknowledgements

We thank our shepherd, Alberto Lerner, and the anonymous reviewers for their comments. This work was supported by Key-Area Research and Development Program of Guangdong Province 2020B0101390001, Joint Funds of the National Natural Science Foundation of China (U20A20179), National Natural Science Foundation of China (62172007), and Natural Science Foundation of Fujian Province of China (2021J05002). Qun Huang is the corresponding author.

References

[1] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proc. of ACM SIGMETRICS*, 2012.

[2] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. TRIAD: Creating synergies between memory, disk and log in log structured key-value stores. In *Proc. of USENIX ATC*, 2017.

[3] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[4] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[5] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: fast programmable match-action processing in hardware for SDN. In *Proc. of ACM SIGCOMM*, 2013.

[6] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. Characterizing, modeling, and benchmarking RocksDB key-value workloads at facebook. In *Proc. of USENIX FAST*, 2020.

[7] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: a distributed storage system for structured data. *ACM Trans. on Computer Systems*, 26(2):1–26, 2008.

[8] Yue Cheng, Aayush Gupta, and Ali R Butt. An in-memory object caching framework with adaptive load balancing. In *Proc. of ACM EuroSys*, 2015.

[9] Eyal Cidon, Sean Choi, Sachin Katti, and Nick McKeown. AppSwitch: Application-layer load balancing within a software switch. In *Proc. of APNet*, 2017.

[10] Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[11] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS operating systems review*, 41(6):205–220, 2007.

[12] Diego Didona and Willy Zwaenepoel. Size-aware sharding for improving tail latencies in in-memory key-value stores. In *Proc. of USENIX NSDI*, pages 79–94, 2019.

[13] Hebatalla Eldakiky, David Hung-Chang Du, and Eman Ramadan. TurboKV: scaling up the performance of distributed key-value stores with in-switch coordination. *CoRR*, abs/2010.14931, 2020.

[14] Bin Fan, Hyeontaek Lim, David G Andersen, and Michael Kaminsky. Small cache, big effect: Provable load balancing for randomly partitioned cluster services. In *Proc. of ACM SOCC*, 2011.

[15] Roy Friedman, Or Goaz, and Dor Hovav. Limited associativity caching in the data plane. *CoRR*, abs/2203.04803, 2022.

[16] Shahram Ghandeharizadeh and Hieu Nguyen. Design, implementation, and evaluation of write-back policy with cache augmented data stores. *Proc. of the VLDB Endowment*, 12(8):836–849, 2019.

[17] Qi Huang, Helga Gudmundsdottir, Ymir Vigfusson, Daniel A Freedman, Ken Birman, and Robbert van Renesse. Characterizing load imbalance in real-world networked caches. In *Proc. of ACM SIGCOMM HotNets Workshop*, 2014.

[18] J-H Hwang, Magdalena Balazinska, Alex Rasin, Ugur Cetintemel, Michael Stonebraker, and Stan Zdonik. High-availability algorithms for distributed stream processing. In *Proc. of IEEE ICDE*, 2005.

[19] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. NetChain: scale-free sub-RTT coordination. In *Proc. of USENIX NSDI*, 2018.

[20] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: balancing key-value stores with fast in-network caching. In *Proc. of ACM SOSP*, 2017.

[21] Jaeyeon Jung, Balachander Krishnamurthy, and Michael Rabinovich. Flash crowds and denial of service attacks: Characterization and implications for cdns and web sites. In *Proc. of WWW*, 2002.

[22] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proc. of ACM STOC*, pages 654–663, 1997.

[23] Ricardo Koller, Leonardo Marmol, Raju Rangaswami, Swaminathan Sundararaman, Nisha Talagala, and Ming Zhao. Write policies for host-side flash caches. In *Proc. of USENIX FAST*, 2013.

[24] Seung-seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. Mind: In-network memory management for disaggregated data centers. In *Proc. of ACM SOSP*, 2021.

[25] LevelDB. https://github.com/google/leveldb/.

[26] Jialin Li, Jacob Nelson, Ellis Michael, Xin Jin, and Dan R. K. Ports. Pegasus: tolerating skewed workloads in distributed storage with in-network coherence directories. In *Proc. of USENIX OSDI*, 2020.

[27] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G Andersen, and Michael J Freedman. Be fast, cheap and in control with SwitchKV. In *Proc. of USENIX NSDI*, 2016.

[28] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. IncBricks: Toward in-network computation with an in-network cache. In *Proc. of ACM ASPLOS*, 2017.

[29] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. DistCache: provable load balancing for large-scale storage systems with distributed caching. In *Proc. of USENIX FAST*, 2019.

[30] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. WiscKey: separating keys from values in SSD-conscious storage. *ACM Trans. on Storage*, 13(1):1–28, 2017.

[31] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. SilkRoad: making stateful layer-4 load balancing fast and cheap using switching ASICs. In *Proc. of ACM SIGCOMM*, 2017.

[32] Srivatsan Narasimhan, Sohum Sohoni, and Yiming Hu. A log-based write-back mechanism for cooperative caching. In *Proc. of IEEE IPDPS*, 2003.

[33] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Harry C. Li Herman Lee, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at Facebook. In *Proc. of USENIX NSDI*, 2013.

[34] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[35] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. Tucana: Design and implementation of a fast and efficient scale-up key-value store. In *Proc. of USENIX ATC*, 2016.

[36] Dai Qin, Angela Demke Brown, and Ashvin Goel. Reliable writeback for client-side flash caches. In *Proc. of USENIX ATC*, 2014.

[37] RocksDB. https://github.com/facebook/rocksdb/.

[38] Mohit Saxena, Michael M Swift, and Yiying Zhang. Flashtier: a lightweight, consistent and durable storage cache. In *Proc. of ACM EuroSys*, 2012.

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

[40] Qing Wang, Youyou Lu, Erci Xu, Junru Li, Youmin Chen, and Jiwu Shu. Concordia: distributed shared memory with in-network cache coherence. In *Proc. of USENIX FAST*, 2021.

[41] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *Proc. of USENIX OSDI*, 2020.

[42] YCSB. https://github.com/brianfrankcooper/YCSB/.