Be Fast, Cheap and in Control with SwitchKV

Xiaozhou Li[1], Raghav Sethi[1], Michael Kaminsky[2], David G. Andersen[3], Michael J. Freedman[1]

[1]*Princeton University,* [2]*Intel Labs,* [3]*Carnegie Mellon University*

Abstract

SwitchKV is a new key-value store system design that combines high-performance cache nodes with resource-constrained backend nodes to provide load balancing in the face of unpredictable workload skew. The cache nodes absorb the hottest queries so that no individual backend node is over-burdened. Compared with previous designs, SwitchKV exploits SDN techniques and deeply optimized switch hardware to enable efficient content-based routing. Programmable network switches keep track of cached keys and route requests to the appropriate nodes at line speed, based on keys encoded in packet headers. A new hybrid caching strategy keeps cache and switch forwarding rules updated with low overhead and ensures that system load is always well-balanced under rapidly changing workloads. Our evaluation results demonstrate that SwitchKV can achieve up to 5× throughput and 3× latency improvements over traditional system designs.

1 Introduction

In pursuit of meeting aggressive latency and throughput service level objectives (SLOs) for Internet services, providers have increasingly turned to in-memory [32, 35] or flash-based [2] key-value storage systems as caches or primary data stores. These systems can offer microseconds of latency and provide throughput hundreds to thousands of times that of the hard-disk-based approaches of yesteryear. The choice of flash vs DRAM comes with important differences in throughput, latency, persistence, and cost-per-gigabyte. Recent advances in SSD performance, including new hardware technologies such as NVMe [34], are opening up new points in the design space of storage systems that were formerly the exclusive domain of DRAM-based systems. However, no single SSD is fast enough, and scale-out designs are necessary both for capacity and throughput.

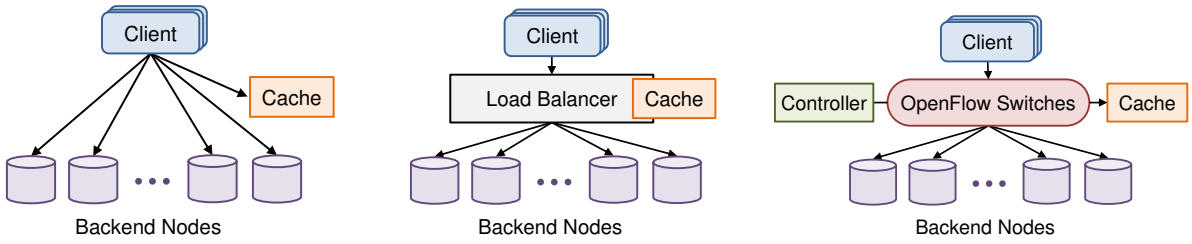
Dynamic load balancing is a key challenge to scaling out storage systems under skewed real-world workloads [3, 5, 7]. The system performance must not become bottlenecked due to unevenly partitioned load across cluster nodes. Conventional static data partitioning techniques such as consistent hashing [23] do not help with dynamic load imbalance caused by skewed and rapidly-changing key popularity. Load balancing techniques that reactively replicate or transfer hot data across nodes often introduce performance and complexity overheads [24].

Prior research shows that a small, fast frontend cache can provide effective dynamic load-balancing by directly serving the most popular items without querying the backend nodes, making the load across the backends much more uniform [13]. That work proves that the cache needs to store only the $O(n \log n)$ hottest items to guarantee good load balance, where n is the total number of *backend nodes* (independent of the number of keys).

Unfortunately, traditional caching architectures such as the look-aside Memcached [15] and an on-path look-through small cache [13] suffer a major drawback when using a frontend cache for load balancing, as shown in Table 1. In these architectures, clients must send all read requests to the cache first. This approach imposes high overhead when the hit ratio is low, which is the case when the cache is small and used primarily for load balancing. Some look-aside systems (Fig. 1a) make the clients responsible for handling cache misses [15], which further increases the system overhead and tail latency. Other designs that place the cache in the frontend load-balancers (Fig. 1b) [13] are vulnerable to the frontend crashes.

SwitchKV is a new cluster-level key-value store architecture that can achieve high efficiency under widely varying and rapidly changing workloads. As shown in Fig. 1c, SwitchKV uses a mix of server classes, where specially-configured high-performance nodes serve as fast, small caches for data that is hash partitioned across resource-constrained backend nodes. At the heart of SwitchKV's design is an efficient content-based routing mechanism that uses software-defined networking (SDN) techniques to serve requests with minimal overhead. Clients encode keys into packet headers and send the requests to OpenFlow switches. These switches maintain forwarding rules for all cached items, and route requests directly to the cache or backend nodes as appropriate based on the embedded keys.

SwitchKV achieves high performance by *moving the cache out of the data path* and by exploiting switch hardware that has already been optimized to match (on query keys) and forward traffic to the right node at line rate with



(a) Look-aside **(b)** On-path look-through **(c)** SwitchKV

Figure 1: Different cache architectures.

	Look-aside	**On-path look-through**	**SwitchKV**
Clients' responsibilities	handle cache misses	nothing (transparent)	encode keys in packet headers
Cache load	*100% queries*	*100% queries*	cache hits (likely <40% queries)
Latency with cache miss	*three* machine transits	*two* machine transits	one machine transit
Failure points	switches	*load balancer*, switches	switches
Cache update involves	cache, backends	cache, backends	cache, backends, *switches*
Cache update rate limit	high	high	*low* (<10K/s in switch hardware)

Table 1: Comparison of different cache architectures.

low latency. All responses return within one round-trip, and there is no overhead for the significant volume of queries for keys that are not in the cache. SwitchKV can scale-out by adding more cache nodes and switches, and is resilient to cache crashes.

The benefits of using OpenFlow switches come at a price: the update rate of forwarding rules in hardware is much lower than that of in-memory caches. Our solution includes an efficient hybrid cache update mechanism that minimizes the cache churn, while still reacting quickly to rapid changes in key popularity. Backends send periodic reports to the cache nodes about their recent hot keys as well as instant reports about keys that suddenly become very popular. Cache nodes maintain query statistics for the cached keys, add or evict the appropriate keys when they receive reports, and instruct SDN controllers to update switch forwarding rules accordingly.

Our SwitchKV prototype uses low-power backend nodes. The same design principles and evaluation results also apply to clusters with more powerful backends, by using high-end cache servers [26] that can keep the same order of performance gap between cache and backends.

The main contributions of this paper are as follows:

- The design of a new cost-effective, large-scale, persistent key-value store architecture that exploits SDN and switch hardware capabilities to enable efficient cache-based dynamic load balancing with content routing.
- An efficient cache update mechanism to meet the challenges imposed by switch hardware and small cache size, and to react quickly to rapid workload changes.
- Evaluation and analysis that shows SwitchKV can handle the traffic characteristics of modern cloud applications more efficiently than traditional systems.

2 Background and Related Work

Clustered Key-Value Stores. Their simple APIs (e.g., `get`, `put`, `delete`) form a fundamental building block of modern cloud services. Given the performance requirements, some systems keep data entirely in memory [35, 38], with disks used only for failure recovery; others put a significant fraction of data in cache to achieve high hit ratio [32]. Systems that aggressively use DRAM are often more expensive and power-hungry than those that use flash storage.

Meanwhile, SSDs are becoming faster, as hardware [34] and software stacks [27, 39] for flash storage become optimized. With a proper design, SSD-based key-value store clusters can be a cost-effective way to meet the SLOs of many cloud services.

Load Balancing. Key-value workloads for cloud applications are often skewed [3, 7]. Many cloud services further experience unpredictable flash crowds or adversarial access patterns [22]. These all pose challenges to scaling out SSD-based key-value clusters, because the service quality is often bottlenecked by the performance of overloaded nodes. For example, a web server may need to contact 10s to 100s of storage nodes with many sequential requests when responding to a page request [32], and the tail latency can significantly degrade the service performance at large scale [10].

Good load balancing is necessary to ensure that the cluster can meet its performance goals without substantial over-provisioning. Consistent hashing [23] and virtual nodes [9] help balance the static load and space utilization, but are unable to balance the dynamic load with skewed query distributions. Traditional dynamic

load balancing methods either use the "power of two choices" [31] or migrate data between nodes [6, 24, 40]. Both are limited in their ability to deal with large skew, are usually too slow to handle rapid workload changes, and often introduce consistency challenges and system overhead for migration or replication.

Caching can be an effective dynamic load balancing technique for hash partitioned key-value clusters [4, 13]. A frontend cache can absorb the hottest queries and make the load across the backends less skewed. Fan et al. [13] prove that the size of the cache required to provide good load balance is only $O(n \log n)$, where n is the total number of backend nodes. This theoretical result inspired the design of SwitchKV.

Caching Architectures. Look-aside [15] and on-path look-through [13] are the two typical caching architectures, shown in Fig. 1 and compared in Table 1. When the cache is small, the hit ratio is usually low (e.g., <40%). This is enough to ensure good load balance, but creates serious overhead in both traditional architectures. The cache is required to process all queries, including those for keys that are not cached, wasting substantial system I/O and network bandwidth in the process.

A cache miss in a look-aside architecture results in an additional round-trip of latency, as the query must be sent back to the client with a cache miss notification, and then resent to the backend. Look-through architectures reduce this latency by placing the cache in the on-path load balancer, however, the cache still must process each incoming request to determine whether to forward or serve it. Additionally, the load balancers become new critical failure points, which are far less reliable and durable than network switches [16].

3 SwitchKV Design

The primary design goal of SwitchKV is to remove redundant components on the query path such that latency can be minimized for all queries, throughput can scale out with the number of backend nodes, and availability is not affected by cache node failures.

The key to achieving this goal is the observation that specialized programmable network switches can play a key role in the caching system. Switch hardware has been optimized for decades to perform basic lookups at high speed and low cost. This simple but efficient function is a perfect match to the first step of a query processing: determine whether the key is cached or not.

The core of our new architectural design is an effective content-based routing mechanism. All clients, cache nodes, and backend nodes are connected with OpenFlow switches, as shown in Fig. 1c. Clients encode keys in query packet headers, and send packets to the cluster switch. Switches have forwarding rules installed, including exact match rules for each cached key and wildcard rules for each backend, to route queries to the right node at line rate. Table 1 summarizes the significant benefits of this new architecture over traditional ones.

Exploiting SDN and fast switch hardware benefits system performance, efficiency and robustness. However, it also adds complexity and limitations. The switches have limited rule space and a relatively slow rule update rate. Therefore, cached keys and switch forwarding rules must be managed carefully to realize the full benefits of this new architecture. The rest of this section describes SwitchKV's query-processing flow and mechanisms to keep the cache up-to-date.

3.1 Content Routing for Queries

We first describe how SwitchKV handles client queries, assuming both cache and switch forwarding rules are installed and up-to-date. The process of updating cache and switch rules will be discussed in Section 3.2.

Query operations are performed over UDP, which has been widely used in large-scale, high-performance in-memory key-value systems for low latency and low overhead [28, 32]. Because UDP is connectionless, queries can be directed to different servers by switches without worrying about connection states. With a well-provisioned network, packet loss is rare [32], and simple application-driven loss recovery is sufficient to ensure both reliability and high throughput [28].

3.1.1 Key Encoding and Switch Forwarding

An essential system component to enable content-based routing is the programmable network switches that can install new *per cached key* forwarding rules on the fly. These switches can use both TCAM and L2/L3 tables for packet processing. The TCAM is able to perform flexible wildcard matches, but it is expensive and power hungry to include on switches. Thus, the size of the TCAM table is usually limited to a few thousand entries [25, 37].[1] The L2 table, however, matches only on destination MAC addresses; it can be more cost-effectively implemented in switches and is more power-efficient. Modern commodity switches support 128K [37] or more L2 entries. These sizes may be insufficient for environments where a large percentage of data must be cached, but is a large enough cache size to ensure good load balancing in SwitchKV.

Key Encoding in Packet Headers. Because MAC addresses have more bits for key encoding and switches usually have large enough L2 tables to store forwarding rules for all cached keys, clients encode query keys in the destination MAC addresses of UDP packets. The MAC

[1] Some high-end switches advertise larger TCAM table (e.g., 125K to 1 million entries [33]), albeit at higher cost and power consumption. Such capabilities would not meaningfully change our design, as our design primarily relies on exact-match rules.

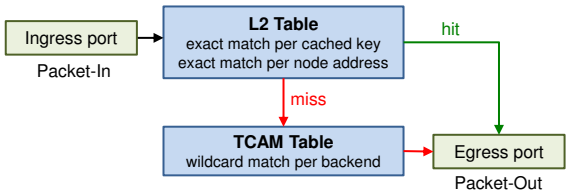


Figure 2: Packet flow through a switch.

consists of a small *prefix* and a *hash* of the key, computed by the same consistent hashing used to partition the keyspace across the backends.

The prefix is used to identify the packet as being a request destined for SwitchKV, and to let the switches distinguish different types of queries. Only `get` queries coming directly from the clients may need to be forwarded to the cache nodes. Other types of queries should be forwarded to the backends, including `put` queries, `delete` queries, and `get` queries from a cache node due to cache misses. Therefore, get queries from the clients use one prefix, and all other queries use a different one.

In order to forward queries to the appropriate backend nodes, each client tracks the mapping between keyspace partitions and the backend nodes, and encodes *identifiers of backends* for the query keys into the destination IP addresses. This mapping changes only when backend nodes are added or removed, so client state synchronization has very low overhead.

Finally, the client's address and identity information is stored in the packet payload so that the node that serves the request knows where to send responses.

Switch Forwarding. There are three classes of rules in switches, which are used to forward `get` queries for the cached keys to the cache nodes, other queries to the backends, and non-query packets (e.g., query responses, cache updates) to the destination node respectively. Fig. 2 shows the packet flow through a switch. The L2 table stores exact match rules on destination MAC addresses for each cached key and each cache and backend node. The TCAM table stores wildcard match rules on destination IP addresses for each backend node.

The L2 table is set to have a higher priority. A switch will first look for an exact match in the L2 table and will forward the packet to an egress port if either the packet was addressed directly to a node or it is a `get` query for a cached key. If there is no match in the L2 table, the switch will then look for a wildcard match in the TCAM and forward the packet to the appropriate backend node.

Below are the detailed switch forwarding rules:

- Exact match rules in L2 table for all cached keys. We use `pre1` to denote the prefix for `get` queries from clients. For each cached key in cache node:
  ```
  match:<mac_dst = pre1-keyhash>
  action:<port_out = port_cache_node>
  ```

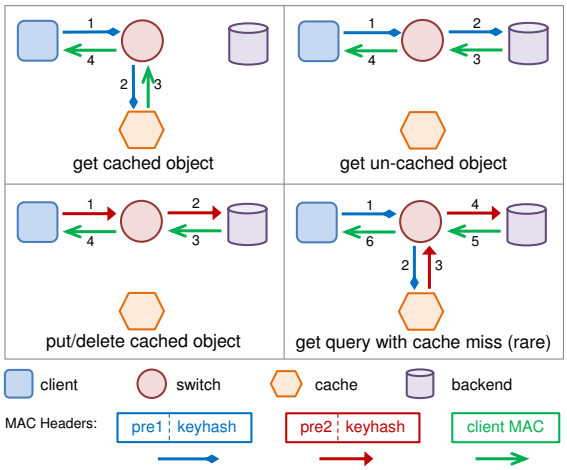


Figure 3: Query packets flows and destination MAC addresses. Internal messages for cache consistency during `put` or `delete` operations are not included. A cache miss only occurs due to key hash collision or temporarily outdated switch rules.

- Exact match rules in L2 table for all clients, cache nodes, and backends. For each node:
  ```
  match:<mac_dst = mac_node>
  action:<port_out = port_node>
  ```
- Wildcard match rules in TCAM table for all backend nodes. For each backend node:
  ```
  match:<ip_dst/mask = id_node>
  action:<port_out = port_backend_node>
  ```

3.1.2 Query Flow Through the System

A main benefit of SwitchKV is that it can send queries to the appropriate nodes with minimal overhead, especially for queries on uncached keys which make up most of the traffic. Fig. 3 shows the possible packet flows of queries.

Handle Read Requests. SwitchKV targets read-heavy workloads, so the efficiency of handling read requests is critical to the system performance. Switches route `get` queries to the cache or backends based on match results in the forwarding tables. When it receives a `get` query, the cache or backend node will look for the key in its local store, either in memory or SSD. The backend will send a reply message with the destination MAC set as the client address. The cache node will also reply if the key is found. This reply will be forwarded back to the client.

In most cases, queries sent to the cache node will hit the cache, because queries for keys not in the cache were filtered out by the switches. However, it is possible for a cache node to receive a `get` query but not find the key in its local in-memory store. This may occur due to a small delay in rule removal from the switch, or a rare hash collision with another key. When this happens, the cache node must forward the packet to the backends. To do so, the cache will send the query packet back to the switch, with the appropriate destination MAC address

prefix (e.g., from `pre1` to `pre2` in Fig. 3). This prevents the packet from matching the same L2 rule in switches again, so that the query can be forwarded to the appropriate backend node via a wildcard match in TCAM.

Handle Write Requests. Clients send `put` and `delete` queries with a MAC prefix that is different from the prefix of `get` queries (as shown in Fig. 3), so that the packets will not trigger a rule in the L2 table of switches, and will be forwarded directly to the backends. When a backend node receives a `put` or `delete` query for a key, it will update its local data store and reply to the client.

Each backend node keeps track of which keys in its local store are also being cached. If a `put` or `delete` request for a cached key arrives, the backend will send messages to update the cache node before replying to the clients. The cache node is then responsible for communicating the update to the network controller for switch rule updates. This policy ensures that data items in the cache and backends are consistent to the client, but allows temporary inconsistency between cached keys and switch forwarding rules. The next section describes the detailed mechanism of cache update and consistency.

3.2 Hybrid Cache Update

As our goal is to build a system that is robust for (nearly) arbitrary workloads, the limited forwarding rule update rate poses challenges for the caching mechanism. Since each cache addition or eviction requires a switch rule installation or removal, the rule update rate in switches directly limits the cache update rate, which affects how quickly SwitchKV can react to workload distribution changes. Though switches are continuously being optimized to speed up their rule update and some switches can now achieve 12K updates per second [33], they are still too slow to support traditional caching strategies that insert each recently-visited key to the cache.

To meet this new challenge, we designed new hybrid cache update algorithms and protocols to minimize unnecessary cache churn. The cache update mechanism consists of three components: 1) Backends *periodically* report recent hot keys to the cache nodes. 2) Backends *immediately* report keys that suddenly become very hot to the cache nodes. 3) Cache nodes add selected keys from reports and evict appropriate keys when necessary, and they instruct the network controller to make corresponding switch rule updates through REST APIs. Cache addition is prioritized over eviction in order to react quickly to sudden workload distribution changes at the cost of some additional buffer switch rule space. Fig. 4 shows our cache update mechanism at a high level.

3.2.1 Update with Periodic Hot Key Report

In most caching systems, a query for a key that is not in the cache would bring that key into cache and evict another key if the cache is full. However, many recently visited keys are not hot and will not be accessed again in the near future. This would result in unnecessary cache churn, which would harm the performance of SwitchKV because its cache update rate is limited.

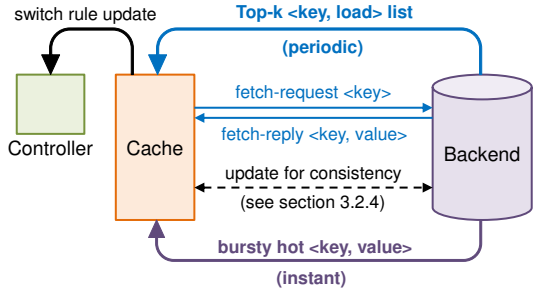


Figure 4: Cache update overview.

Instead, we use a different approach to add objects to the cache less aggressively. Each backend node maintains an efficient top-k load tracker to track recent popular keys. Backend nodes periodically (e.g., every second) report their recent hot keys and loads to the cache nodes. Each cache node maintains an in-memory data store and frequency counter for the cached items with the same load metric. The cache node keeps a load threshold based on the load statistics of cached keys. Upon receiving the reports, the cache node selects keys whose loads are above the threshold to add to the cache. It sends `fetch` requests for the selected keys to the corresponding backend nodes to get the values. It then updates the cache and instructs the network controller to update switch rules based on the received `fetch` responses.

Time-segmented Top-K Load Tracker. Each backend node maintains a key-load list with k entries to store its approximated hottest k keys and their loads. It also keeps a local frequency counter for the recently visited keys, so that it can know what are the most popular keys and how frequently they are queried. A backend node cannot afford to keep counters for all keys in memory. Instead, since only information about hot keys is needed, we can use memory-efficient top-k algorithms to find frequent items in the query stream [8].

To keep track of *recent* hot keys, we segment the query stream into separate intervals. At the end of each interval, the frequency counter extracts the top-k list of its current segment, then clears itself for the next segment. The key-load list is updated by the top-k list of the new segment using weighted average. Suppose the frequency of key x in the new segment is f_x, and the current load of x is L'_x, then the new load of x is

$$L_x = \alpha \cdot f_x + (1 - \alpha) \cdot L'_x, \quad (1)$$

where α represents the degree of weighting decrease. A higher α discounts previous load faster. Only keys in the new top-k list will be kept in the new updated key-load list. L'_x is zero for keys not in the previous key-load list.

Algorithm 1 Update Frequency Counter

```
1:  function SeeQuery(x)
2:      if x is not tracked in the counter then
3:          if the counter is not full then
4:              create a bucket with f = 1 if not exists
5:              add x to the first bucket; return
6:          y ← first key of first bucket, the least visited key
7:          replace y with x and keep the same frequency
8:      Update(x)
9:  function Update(x) // key x is tracked in the counter
10:     ⟨b, f⟩ ← current ⟨bucket, frequency⟩ of x
11:     if next bucket of b has frequency f + 1 then
12:         move x to the next bucket
13:     else if x is the only key of b then
14:         increase frequency of b to f + 1
15:     else move x to a new bucket with frequency f + 1
16:     if b is empty then delete b
```

The frequency counter uses a "space-saving algorithm" [30] to track the heavy hitters of the query stream in each time segment and approximate the frequencies of these keys. Fig. 5 shows the data structure of the frequency counter.

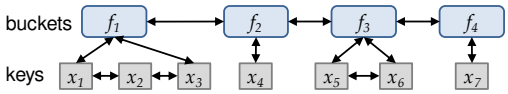


Figure 5: Structure of top-k frequency counter.

The counter consists of a linked list of buckets, each with a unique frequency number f. The buckets are sorted by their frequency in increasing order (e.g., $f_1 < f_2$). Each bucket has a linked list of keys that have been visited for the same number of times, f. Keys in the same bucket are sorted by their most recent visited time, with newest key at the tail of the list. With this structure, getting a list of top-k hot keys and their load is straightforward. For example, the top-5 list in Fig. 5 is $[\langle x_7, f_4\rangle \langle x_6, f_3\rangle \langle x_5, f_3\rangle \langle x_4, f_2\rangle \langle x_3, f_1\rangle]$.

The counter has a configurable size limit N, which is the maximum number of keys it can track. Algorithm 1 describes how to update the counter. When processing (e.g., create, delete, move) buckets and keys, the orders described above are always maintained. The counter requires O(N) memory, and has O(1) running time for each query. To reduce the computational overhead, we can randomly sample the query packets, and only update the counter for a small fraction of the queries. Sampling can provide a good approximation of the ranking of heavy hitters in highly skewed workloads.

Cache Adds Selected Keys from Reports. The cache also tracks the load for all cached keys. In order to be comparable with the load of reported keys, it must keep the same parameters (e.g., time segment interval, average weights, sampling rate) with the tracker in the backends. Cache nodes update a load threshold periodically based on the loads of cached keys, and send `fetch` queries for the reported keys with load higher than the threshold.

Too big of a threshold would prevent caching hot keys, while a too small of one would cause frequent unnecessary cache churn. To compute a proper load threshold in practice, the cache samples a certain number of key loads and uses the load at a certain rank (e.g., 10^{th} percentile from the lowest) as the threshold value. This process runs in the background periodically, so it does not introduce overhead to serving queries or updating cached data.

3.2.2 Update for Bursty Hot Keys

Periodic reports can update the cache effectively with low communication and memory overhead, but cannot react quickly when some keys suddenly become popular. In addition to periodic reports, the backends also send instant reports to the cache to report bursty queries, so that those queries can be offloaded to the cache immediately.

Each backend maintains a circular log to track the recently visited keys, and a hash table that keeps only entries for keys currently in the log and tracks the number of occurrence of these keys. As shown in Fig. 6, when a key is queried, it is inserted into the circular log, with the existing key at that position evicted. The hash table updates the count of the keys accordingly and adds or deletes related entries when necessary. If the count of a key exceeds a threshold and the node's overall load is also above a certain threshold, the key and its value are *immediately sent and added to the cache*. The size of the circular log and hash table could be small (e.g., a few hundreds of entries), which introduces little overhead to query processing.

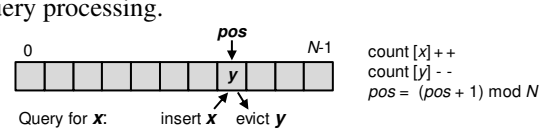


Figure 6: Circular log and counter.

3.2.3 Handle Burst Change with Rule Buffer

The distribution changes in real-world workloads are not constant. Sudden changes in the key popularity may lead a large number of cache updates in a short period of time. In traditional caching algorithms, a cache addition when the cache is full would also trigger a cache eviction, which in SwitchKV would mean that each addition involves two forwarding rule updates in the switch. As a result, the cache would only able to add keys at half of the switch update rate on average.

In order to react quickly to sudden workload changes, we prioritize cache addition over eviction. Cache evictions and switch rule deletion requests are queued and executed after cache additions and rule installations until a maximum delay is reached. In this way, we can

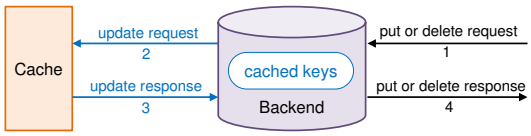


Figure 7: Updates to keep cache consistency.

reduce the required peak switch update rate for bursty cache updates to half, so that new hot keys can be added to cache more quickly. For example, if the switch update rate limit is 2000 rules per second, and the maximum delay for rule deletions is one second, then the cache can update at 1000 keys/second on average, and a maximum of 2000 keys/second for a short period (one second).

To allow delay in switch rule deletions, a rule buffer must be reserved in the L2 table. The size of this buffer is the maximum switch update rate times the duration of maximum delay. In the example above, the switch should reserve space for at least 2000 rules, which is small compared to the available L2 table size in switches.

Delaying rule deletion may result in stale forwarding rules in the L2 table. The stale rules will produce a temporary cache miss for some queries, as shown in the lower right block of Fig. 3. The miss overhead is small, however, because the evicted or deleted keys are (by definition) less likely to be frequently visited.

3.2.4 Cache Consistency

SwitchKV always guarantees consistent responses to clients. As a performance optimization, it allows temporary inconsistency between switch forwarding rules and cached keys, which (as described above) can introduce temporary overheads for a small number of queries, but never causes inconsistent data access.

In traditional cache systems such as Memcached [15], when a client sends a `put` or `delete` request, it will also send a request to the cache to either update or invalidate the item if it is in cache. The cache in SwitchKV is small and it is possible that most requests are for uncached keys, so forwarding each `put` or `delete` request to the cache introduces unnecessary overhead.

The backends avoid this overhead by tracking, in-memory, which keys in its local store are currently cached. The backend only updates the cache when it receives requests for one of these cached keys. Keys are added to the set whenever the backend receives a `fetch` request, or sends an instant hot object detected by the circular-log counter. When the cache evicts a key, or decides not to add the item from a `fetch` response or instant report, it sends a message to the backend so that the backend can remove this key from its cached key set.

We use standard leasing mechanisms to ensure consistency when there are cache or backend failures or network partitions [17]. Backends grant the cache a short-term lease on each cached key. The cache periodically renews its leases and only return a cached value while the lease is still valid. When a backend receives a `put` or `delete` request for a cached key, it will send an `update` request to the cache, as shown in Fig. 7, and will wait for the response or until the lease expires before it replies to the client. We choose to update the cached data rather than invalidate it for a `put` request to reduce the cache churn and rule update burden on switches.

3.3 Local Storage and Networking

Optimizing the single-node local performance of cache and backends is not our primary goal, and has been extensively researched [27, 28, 39]. Nevertheless, we made several design choices on local storage and networking to maximize the potential performance of each server, which we discuss here.

3.3.1 Parallel Data Access

Exploiting the parallelism of multi-core systems is critical for high performance. Many key-value systems use various concurrent data structures so that all cores can access the shared data in parallel [11, 14, 32]. However, they usually scale poorly with writes and can introduce significant overhead and complexity to our cache update algorithms that require query statistics tracking.

Instead, SwitchKV partitions the data in each cache and backend node based on key hash. Each core has *exclusive access* to its own partition, and runs its own load trackers. This greatly improves both the concurrency and simplicity of the local stores. Prior work [14, 29, 41] observed that partitioning may lower the performance when the load across partitions is imbalanced. In SwitchKV, however, backend nodes do not face high skew in key popularity. By exploiting CPU caches and packet burst I/O, a cache node that serves a small number of keys can handle different workload distributions [28].

3.3.2 Network Stack

SwitchKV uses Intel® DPDK [21] instead of standard socket I/O, which allows our user-level libraries to control NICs, modify packet headers, and transfer packet data with minimal overhead [28].

Since each core in the cache and backend nodes has exclusive access to its own partition of data, we can have the NIC deliver each query packet to the appropriate RX queue based on the key. SwitchKV can achieve this by using Receive Side Scaling (RSS) [12, 20] or Flow Director (FDir) [28, 36].[2] Both methods require information about the key in packet headers for the NIC to identify which RX queue should the packet be sent to. This requirement is automatic in SwitchKV where key hashes are already part of the packet header.

[2]Our prototype uses RSS. FDir enables more flexible control of the network stack, but it is not supported in the Mellanox NICs that we use.

3.4 Cluster Scaling

To scale system performance, the cluster will require multiple caches and OpenFlow switches. This section briefly sketches a design (not yet implemented) for a scale-out version of SwitchKV.

Multiple Caches. We can increase SwitchKV's total system throughput by deploying additional cache nodes. As each individual node can deliver high throughput because of its small dataset size (especially when keys fit within its L3 cache), we do not replicate keys across nodes and instead simply partition the cache across the set of participating nodes.[3] Each cache node is responsible for multiple backends, and each backend reports only to its dedicated cache node. As such, we do not require any cache coherency protocols between the cache nodes.

If the mapping between backends and cache nodes changes, the relevant backends will delete their cached items from their old cache nodes, and then report to the new ones. If the change is due to a cache crash, the network controller will detect the failed node and delete all forwarding rules to it.

Network Scaling. To scale network throughput, we can use the well-studied multi-rooted fat-tree [1, 19]. Such an architecture may require exact match rules for cached keys to be replicated at multiple switches. This approach may sacrifice performance until the rule updates complete, but does not compromise correctness (the backends may need to serve the keys temporarily).

On the other hand, if the switching bottleneck is in terms of rule space (as opposed to bandwidth), then each switch must be configured to store only rules for a subset of the backend nodes, i.e., we partition the backends, and thus the rule space, across our switches. In this case, queries for keys in a backend node must be sent through a switch associated with that key's backend (i.e., that has the appropriate rules); that switch can be identified easily by the query packets' destination IP addresses.

4 Evaluation

In this section, we demonstrate how our new architecture and algorithms significantly improve the overall performance of a key-value storage cluster under various workloads. Our experiments answer three questions:

- How well does a fast small cache improve the cluster load balance and overall throughput? (§4.2)
- Does SwitchKV improve system throughput and latency compared to traditional architectures? (§4.3)
- Can SwitchKV's new cache update mechanism react quickly to workload changes? (§4.4)

[3]Note that while we *are* very concerned about load amongst our backend nodes, our cache nodes have orders-of-magnitude higher performance, and thus the same load-balancing concerns do not arise.

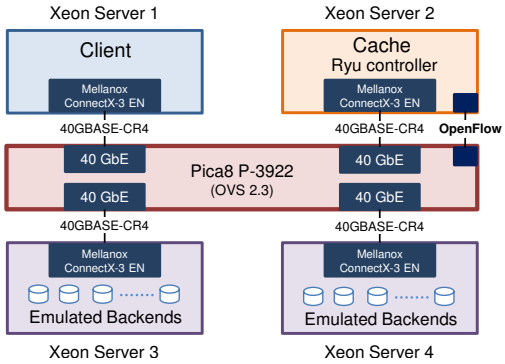


Figure 8: Evaluation platform.

Our SwitchKV prototype is written in C/C++ and runs on x86-64 Linux. Packet I/O uses DPDK 2.0 [21]. In order to minimize the effects of implementation (rather than architectural) differences, we implemented the look-aside and look-through caches used in our evaluation simply by changing the query data path in SwitchKV.

4.1 Evaluation Setup

Platform. Our testbed consists of four server machines and one OpenFlow switch. Each machine is equipped with dual 8-core CPUs (Intel® Xeon® E5-2660 processors @ 2.20 GHz), 32 GB of total system memory, and one 40Gb Ethernet port (Mellanox ConnectX-3 EN) that is connected to one of the four 40GbE ports on a Pica8 P-3922 switch. Fig. 8 diagrams our evaluation platform. One machine serves as the client, one machine as the cache, and two machines emulate many backends nodes.

We derived our emulated performance from experimental measurements on a backend node that fits our target configuration: an Intel® Atom™ C2750 processor paired with an Intel® DC P3600 PCIe-based SSD. On this SSD-based target backend, we ran RocksDB [39] with 120 million 1KB key-value pairs, and measured its performance against a client over a 1Gb link. The backend could serve 99.4K queries per second on average.

Each emulated backend node in our experiments runs its own isolated in-memory data structures to serve queries, track workloads, and update the cache. It has a configurable maximum throughput enforced by a fine-grained rate limiter.[4] The emulated backends do not store the actual key-value pairs due to limited memory space. Instead, they reply to the client or update the cache with a fake random value for each key. In most experiments (except Fig. 14), we emulate a total of 128 backend nodes in the two server machines, and limit each node to serve at most 100K queries per second. Table 2 summarize the default experiment settings unless otherwise specified.

[4]Since it is hard to predict the performance bottleneck at a backend node if its load is skewed, we assume backends have a fixed throughput limit as measured under uniform workloads.

Number of backend nodes	128
Max throughput of each backend	100 KQPS
Workload distribution	Zipf (0.99)
Number of items in cache	10000

Table 2: Default experiment settings unless otherwise specified

Workloads and Method. We evaluate both skewed and uniform workloads in our experiments, and focus mainly on skewed workloads. Most skewed workloads use a non-uniform key popularity that follows a Zipf distribution of skewness 0.99, which is the same that used by YCSB [7]. The request generator uses approximation techniques to quickly generate workloads with a Zipf distribution [18, 28]. The keyspace size is 10 billion, so each of the 128 backend nodes is responsible for serving approximately 78 million unique keys. The mapping of a given key to a backend is decided by the key hash. We use fixed 16-byte keys and 128-byte values.

Most experiments (except Fig. 12) use read-only workloads, since SwitchKV aims to load balance read requests. All write requests have to be processed by the backends, so they cannot be load balanced by the cache.

To find the maximum effective system throughput, the client tracks the packet loss rate, and adjusts its sending rate every 10 milliseconds to keep the loss rate between 0.5% to 1%. This self-adjusted rate control enables us to evaluate the real-time system performance.

Our server machines can send packets at 28 Mpps, but receive at only 15 Mpps. To avoid the system being bottlenecked by the client's receiving rate, the backends and cache node fully process all incoming queries, but send only half of the responses back to the client. The client doubles its receiving rate before computing the loss rate.

4.2 Load Balancing with a Small Cache

We first evaluate the effectiveness of introducing a small cache for reducing load imbalances.

Fig. 9 shows a snapshot of the individual backend node throughput with caching disabled under workloads of varying skewness. We observe that the load across the backend nodes is highly imbalanced.

Fig. 10 shows how caching affects the system throughput. Under uniform random workload, the backends total throughput can reach near the maximum capacity (128 backends $\times$ 100 KQPS). However, when the workload is skewed, the system throughput without the cache is bottlenecked by the overloaded node and significantly reduced. Adding a small cache can help the system achieve good load balance across all of nodes: A cache with only 10,000 items can improve the system's overall throughput by 7$\times$ for workloads with Zipf skewness of 0.99.

Fig. 11 investigates how different numbers of cached items affect the system throughput. The backends' load quickly becomes well balanced as the number of cached

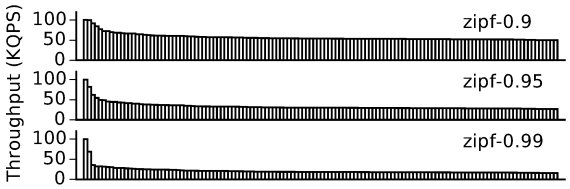


Figure 9: Throughput of each backend node without cache under workloads with different Zipf skewness. Node IDs (x-axis) are sorted according to their throughput.

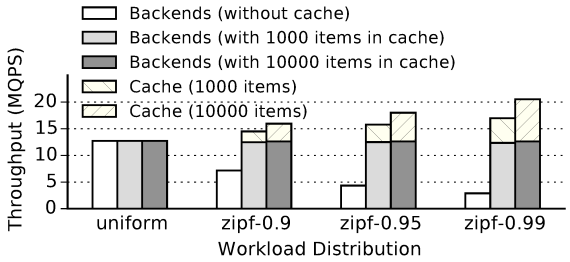


Figure 10: System throughput with and without the use of a cache. Figure illustrates the portion of total throughput handled by the cache and that by backend nodes.

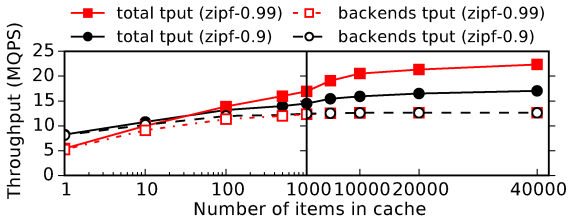


Figure 11: System throughput as cache size increases. Even a modest-sized cache of 10,000 items achieves significant gains.

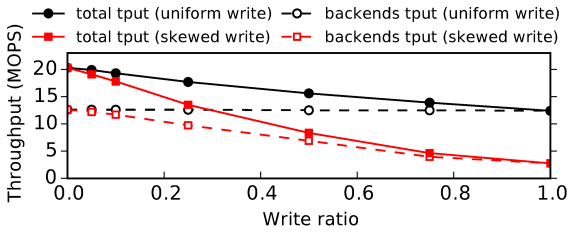
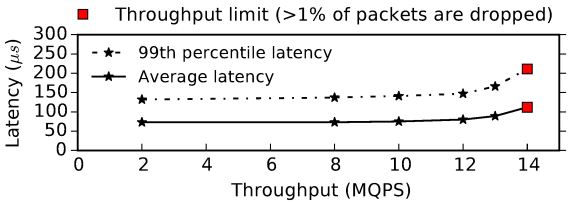


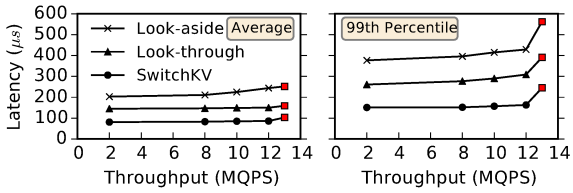
Figure 12: System throughput with different write ratio.

items grows to 1000. Then, the system throughput continues to grow as more items are cached, but the benefits from increased cache size diminish (as one expects given a Zipf workload). The system would require significantly more memory at the cache node or many more cache nodes to further increase the hit ratio. We choose to cache 10,000 items for the rest of the experiments.

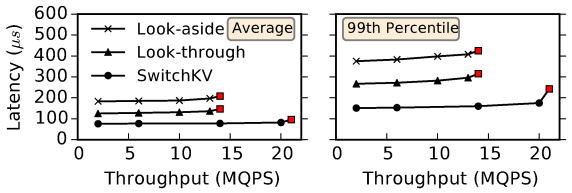
Fig. 12 plots the systems throughput with different write ratios and write workloads. We assume the backend nodes have the same performance for read and write operations, and use two types of write workload: write queries uniformly distributed across all keys and write queries according to the same Zipf 0.99 distribution as read queries. Write workloads cannot be balanced by



(a) Queries for cached keys with Zipf 0.99 workloads.



(b) Queries for uncached keys with uniform workloads.



(c) Zipf 0.99 workloads with 10000 items in cache.

Figure 13: End-to-end latency as a function of throughput.

the cache, so the system throughput with skewed write workload quickly decreases as the write ratio increases. With the uniform write workload, load across the backends is always uniform, so increasing the write ratio only decreases the effective throughput of the cache.

4.3 Benefits of the New Architecture

This section compares the system performance between SwitchKV and traditional look-aside and on-path look-through architectures. As summarized in Table 1, compared to traditional architectures in which the cache handles all queries first, the cache in SwitchKV is only involved when the requested key is already cached (with high likelihood), and thus uncached items are served with only a single machine transit. As a result, we expect SwitchKV to have both lower latency and higher throughput than traditional architectures, which is strongly supported by our experimental results.

Latency. We first compare the average and 99^{th} percentile latency of different architectures, as shown in Fig. 13. To measure the end-to-end latency, the client tags each query packet with the current timestamp. When receiving responses, the client compares the current timestamp and the previous timestamp echoed back in the responses. To measure latency under different throughputs, we disable the client's self rate adjustment, and manually set different send rates.

Fig. 13a shows the latency when the client only sends queries for keys in the cache. In all three architectures,

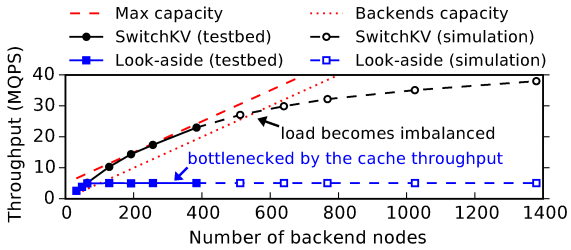


Figure 14: System throughput scalability as the number of backend nodes increases, for SwitchKV and look-aside architecture with Zipf 0.99 workload and at most 10000 items in cache. On-path look-through has the same throughput as look-aside. Each backend node is rate limited at 50K queries per second, cache is rate limited at 5 million queries per second. Look-through has similar performance to look-aside.

the queries will be forwarded to the cache by the switch and the cache reply directly to the client. Accordingly, they have the same latency for cache hits.

Fig. 13b shows the latency when the client generates uniform workloads and the cache is empty, which results in all queries missing the cache. Look-aside has the highest latency because it takes three machine transits (cache→client→backend) to handle a cache miss. Look-through also has high latency because it takes two machine transits (cache→backend) to handle a cache miss. In comparison, queries for uncached keys in SwitchKV cache will directly go to the backend nodes.

Fig. 13c shows the overall latency for a Zipf 0.99 workload and 10000-item cache. As shown in Fig. 10, about 38% of queries will hit the cache under these settings. The average latency is within the range of cache hits and cache misses. The 99^{th} percentile latency is about the same as cache miss latency. As all queries must go through the cache in look-aside and look-through architectures, we cannot collect latency measurements beyond the 14 million QPS mark for them, as the cache is unable to handle more traffic. This result illustrates one of the major benefits of the SwitchKV design: requests for uncached keys are simply not sent to the cache, allowing a single cache node to support more backends (higher aggregate system throughput).

Throughput. We then compare the full system throughput under a Zipf 0.99 workload as the number of backend nodes increase, for different architectures. For each architecture, the cache node stores at most 10000 items.

In order to emulate more backend nodes in this experiment, we scale down the rate capacity of each backend node to at most 50K queries per second, and limit the cache to serve at most 5 million queries per second. The performance improvement ratio of SwitchKV to other architectures will be the same as long as the performance ratio of the cache to a backend node is 100:1. To achieve the maximum system throughput, the cache may store

fewer items when it becomes the performance bottleneck as the backend cluster size increases.

Fig. 14 shows the experiment results. The throughput of the look-aside architecture is bottlenecked quickly by the cache capacity when the number of backend nodes increases to 64, while the throughput of SwitchKV can scale out to much larger cluster sizes. When the number of backend nodes goes beyond 400, the throughput begins to drop below the maximum system capacity, because the cache is insufficient for providing good load balance for such a cluster. To retain linear scalability as the cluster grows, we would need to have a more powerful cache node or increase the number of cache nodes.

Less skewed workloads will yield better scalability for SwitchKV, but will hit the same performance bottleneck for both look-aside and look-through architectures. Due to space constraints, we omit these results.

4.4 Cache Updates

This section evaluates the effectiveness of SwitchKV's hybrid cache-update mechanisms. In these experiments, we keep the workload distribution (Zipf 0.99) the same, and change only the popularity of each key. The workload generator in the client actually generates key indices with fixed popularity ranks. We change the query workloads by changing the mapping between indices and key strings. We use three different workload change patterns:

1. ***Hot-in***: Move N cold keys to the top of the popularity ranks, and decrease the ranks of other keys accordingly. This change is radical, as cold keys suddenly become the hottest ones in the cluster.
2. ***Hot-out***: Move N hottest keys to the bottom of the popularity ranks, and increase the ranks of other keys accordingly. This change is more moderate, since the new hottest keys are most likely already in the cache if N is smaller than the cache size.
3. ***Random***: Replace N random keys in the top K hottest keys with cold keys. We typically set K to the cache size. This change is typically moderate when N is not large, since the probability that most of the hottest keys are changed at once is low.

A note about our experimental infrastructure, which affects SwitchKV's performance under rapid workload changes: The Pica8 P-3922 switch's L2 rule update is poorly implemented. The switch performs an unnecessary linear scan of all existing rules before each rule installation, which makes the updates very slow as the L2 table grows. We benchmark the switch and find it can only update about 400 rules/second when the there are about 10K existing rules, which means the cache can only update 200 items/second on average. Some other switches can update their rules much faster (e.g., 12K updates/second [33]). Though still too slow to support the update rate needed by traditional caching algorithms, these switches would provide much higher performance with SwitchKV under rapidly changing workloads.

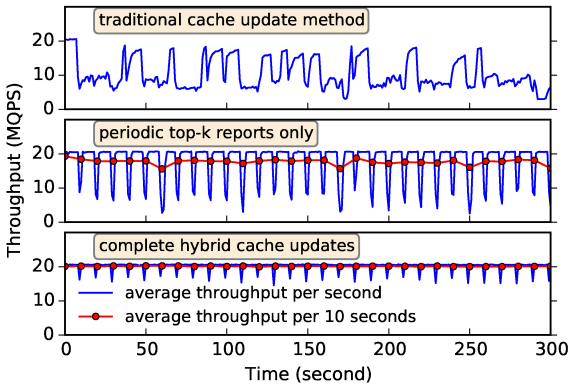


Figure 15: Throughput with *hot-in* workload changes, i.e., change 200 cold keys into the hottest keys every 10 seconds.

All experiments use Zipf 0.99 workloads and a 10000-item-sized cache. Each experiment begins with a pre-populated cache containing the top 10,000 hot items. Each backend node sends reports to the cache as follows: its top five hot keys every second, and keys that were visited more than eight times within the last two hundred queries instantly. The choice of parameters for periodic and instant updates is flexible, determined by the performance goals, cache size, and update rate limit. For example, the size and threshold of the ring counter for instant reports determines when a key is hot enough to be immediately added to the cache. A threshold that is too low may cause unnecessary cache churn, while a threshold that is too high may make the cache slow to respond to bursty workload changes. We omit a sensitivity analysis of these parameters due to space limits. We also compare SwitchKV with a traditional update method, in which backends try to add every queried key to the cache.

We first evaluate system throughput under the *hot-in* change pattern. Since this is a radical change, we do not expect it to happen frequently. Thus, we move 200 cold keys to the top of the popularity ranks every ten seconds. Fig. 15 shows the system throughput over time. A traditional cache update method has very poor performance, as it performs many cache updates for recently-visited yet non-hot keys. With periodic top-k reports alone, a backend's hot keys are not added to the cache until its next report (once per second). The throughput is reduced to less than half after the workload changes, and recovers in 1-2 seconds. The bottom subfigure shows SwitchKV's throughput using its complete cache update mechanism, which includes the instant hot key reports. The new hot keys are immediately added to the cache, resulting in a lower performance drop and a much faster recovery after a sudden workload change. This demonstrates that SwitchKV is robust enough to meet the SLOs even with certain adversarial changes in key popularity.

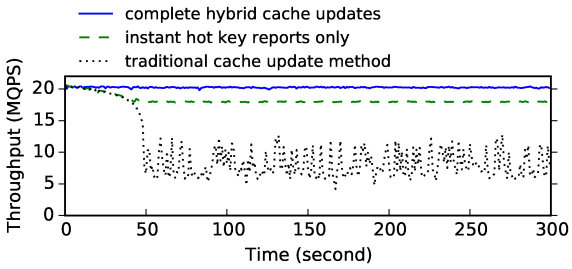


Figure 16: Throughput with *hot-out* workload changes, i.e., move out 200 hottest keys every second.

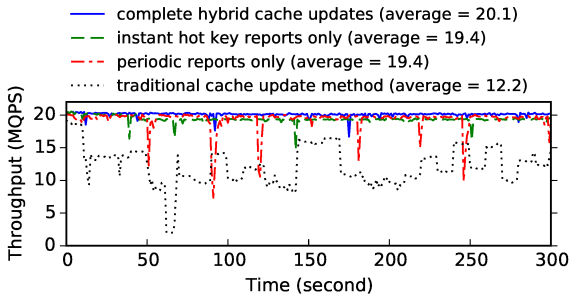


Figure 17: Throughput with *random* workload changes, i.e., replace 200 out of the top 10000 keys every second.

Our next experiment evaluates SwitchKV's throughput under a *hot-out* change pattern. Every second, the 200 hottest keys suddenly go cold, and we thus increase the popularity ranks of all other keys accordingly. As shown in Fig. 16, the complete update mechanism can handle this change well. With instant reports only and no periodic reports, the system cannot achieve its maximum throughput: the circular log counter can detect only very hot keys, not the keys just entering the bottom of the top-10000 hot-key list. These keys are only added to cache as they further increase in their popularity when more of the hottest keys move out. Note that this gap becomes particularly apparent as the system reaches its steady state 50 seconds into the experiment; at this point, none of the pre-populated cached keys remain in the cache.

Fig. 17 shows the throughput with a *random* change pattern, in which we randomly replace 200 keys in the top 10000 popular keys every second. The complete update mechanism is able to handle the workload changes. There are occasionally short-term small performance drops, which occur when the hottest keys are replaced. The throughput would be lower, however, if SwitchKV were to omit either its instant or periodic reports.

Fig 18 shows the effectiveness of SwitchKV's rule buffer in handling bursty workload changes (see §3.2.3). The maximum delay for cache eviction and rule deletion is set to 2 seconds. With a switch rule buffer and prioritizing rule installation, the 600 new keys can be added to the cache within 1.5 seconds. Without the rule buffer, this installation time would double. The rule buffer thus

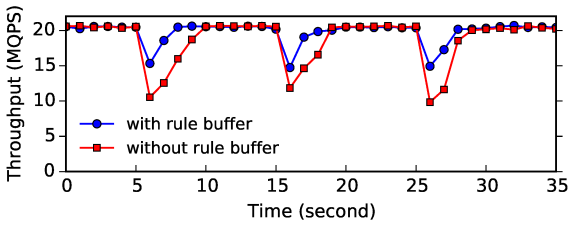


Figure 18: Throughput with *hot-in* workload changes with 600 new hottest keys every time, which requires 1200 rule updates and will take the switch at least three seconds to finish them.

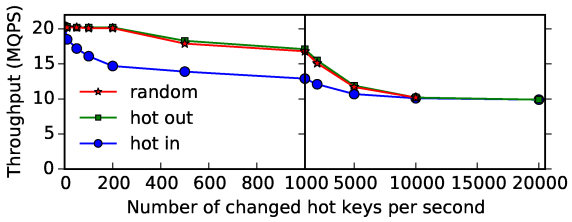


Figure 19: Throughput with different workload change patterns as a function of change rate.

reduces any throughput impact and allows faster recovery during bursty workload changes.

Fig. 19 shows the average throughput with different change patterns and rates. The switch can update 400 rules per second, which can support 200 cache updates per second. The system throughput is near maximum for random and hot-out change patterns when the change rate is within 200 keys per second, and then goes down as the change rate increases. Throughput drops quickly under increasing hot-in changes, as the cache is less effective when more of the hottest keys change every second. Once all patterns change more than 10000 of the hottest keys per second, all three patterns yield similar throughput, as all patterns replace the entire cache every second. Still, even at this point the cache can still keep up to 200 of current hot keys, and most of the hottest keys are likely to added to the cache from the instant reports, so throughput is still much higher (by 3×) than that of the system lacking a cache. The performance under fast changing workloads would be higher with switches that can update their rules faster.

5 Conclusion

SwitchKV is a scalable key-value storage system that can maintain efficient load balancing under widely varying and rapidly changing real-world workloads. It achieves high performance in a cost effective manner, both by combining fast small caches with new algorithm design, and by exploiting SDN techniques and switch hardware. We demonstrate SwitchKV can meet the service-level objectives for throughput and latency more efficiently than traditional systems.

Acknowledgments

The authors are grateful to Hyeontaek Lim, Anuj Kalia, Sol Boucher, Conglong Li, Nanxi Kang, Xin Jin, Linpeng Tang, Aaron Blankstein, Haoyu Zhang, our shepherd Jinyang Li and the anonymous NSDI reviewers for their constructive feedback. This work was supported by National Science Foundation Awards CSR-0953197 (CAREER) and CCF-0964474, and by Intel via the Intel Science and Technology Center for Cloud Computing (ISTC-CC).

References

[1] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM Conference on Data Communication*, 2008.

[2] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. Fawn: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, 2009.

[3] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, 2012.

[4] A. Bestavros. Www traffic reduction and load balancing through server-based caching. *IEEE Parallel Distrib. Technol.*, 5(1), Jan. 1997.

[5] P. Bodik, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson. Characterizing, Modeling, and Generating Workload Spikes for Stateful Services. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SOCC)*, 2010.

[6] Y. Cheng, A. Gupta, and A. R. Butt. An in-memory object caching framework with adaptive load balancing. In *Proceedings of the Tenth European Conference on Computer Systems*, 2015.

[7] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, 2010.

[8] G. Cormode and M. Hadjieleftheriou. Finding frequent items in data streams. *Proc. VLDB Endow.*, 1(2), Aug. 2008.

[9] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with cfs. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[10] J. Dean and L. A. Barroso. The tail at scale. *Commun. ACM*, 56(2), Feb. 2013.

[11] B. Debnath, S. Sengupta, and J. Li. Flashstore: High throughput persistent key-value store. *Proc. VLDB Endow.*, 3(1-2), Sept. 2010.

[12] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. Routebricks: Exploiting parallelism to scale software routers. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP)*, 2009.

[13] B. Fan, H. Lim, D. G. Andersen, and M. Kaminsky. Small Cache, Big Effect: Provable Load Balancing for Randomly Partitioned Cluster Services. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC)*, 2011.

[14] B. Fan, D. G. Andersen, and M. Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2013.

[15] B. Fitzpatrick. Distributed Caching with Memcached. *Linux J.*, 2004(124):5–, Aug. 2004. ISSN 1075-3583.

[16] P. Gill, N. Jain, and N. Nagappan. Understanding network failures in data centers: Measurement, analysis, and implications. In *Proceedings of the ACM SIGCOMM Conference on Data Communication*, 2011.

[17] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles (SOSP)*, 1989.

[18] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billion-record synthetic databases. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, 1994.

[19] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Vl2: A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM Conference on Data Communication*, 2009.

[20] S. Han, K. Jang, K. Park, and S. Moon. Packetshader: A gpu-accelerated software router. In *Proceedings of the ACM SIGCOMM Conference*, 2010.

[21] Intel Data Plane Development Kit (DPDK). `http://dpdk.org/`.

[22] J. Jung, B. Krishnamurthy, and M. Rabinovich. Flash crowds and denial of service attacks: Characterization and implications for cdns and web sites. In *Proceedings of the 11th International Conference on World Wide Web (WWW)*, 2002.

[23] 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 *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing (STOC)*, 1997.

[24] M. Klems, A. Silberstein, J. Chen, M. Mortazavi, S. A. Albert, P. Narayan, A. Tumbde, and B. Cooper. The yahoo!: Cloud datastore load balancer. In *Proceedings of the Fourth International Workshop on Cloud Data Management (CloudDB)*, 2012.

[25] D. Kreutz, F. Ramos, P. Esteves Verissimo,

C. Ẽsteve Rothenberg, S. Azodolmolky, and S. Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1), Jan. 2015.

[26] S. Li, H. Lim, V. W. Lee, J. H. Ahn, A. Kalia, M. Kaminsky, D. G. Andersen, O. Seongil, S. Lee, and P. Dubey. Architecting to achieve a billion requests per second throughput on a single key-value store server platform. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, 2015.

[27] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A Memory-efficient, High-performance Key-value Store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[28] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. Mica: A holistic approach to fast in-memory key-value storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2014.

[29] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys)*, 2012.

[30] A. Metwally, D. Agrawal, and A. El Abbadi. Efficient Computation of Frequent and Top-k Elements in Data Streams. In *Proceedings of the 10th International Conference on Database Theory (ICDT)*, 2005.

[31] M. Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Trans. Parallel Distrib. Syst.*, 12(10), Oct. 2001.

[32] 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 *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2013.

[33] NoviSwitch. `http://noviflow.com/products/noviswitch/`.

[34] NVM Express. `http://www.nvmexpress.org/`.

[35] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The Case for RAMClouds: Scalable High-performance Storage Entirely in DRAM. *SIGOPS Oper. Syst. Rev.*, 43(4):92–105, Jan. 2010.

[36] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris. Improving network connection locality on multicore systems. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys)*, 2012.

[37] Pica8. `http://www.pica8.com/`.

[38] Redis. `http://redis.io/`.

[39] RocksDB. `http://rocksdb.org/`.

[40] R. Taft, E. Mansour, M. Serafini, J. Duggan, A. J. Elmore, A. Aboulnaga, A. Pavlo, and M. Stonebraker. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proc. VLDB Endow.*, 8(3), Nov. 2014.

[41] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*, 2013.