NetCache: Balancing Key-Value Stores with Fast In-Network Caching

Xin Jin[1], Xiaozhou Li[2], Haoyu Zhang[3], Robert Soulé[2,4], Jeongkeun Lee[2], Nate Foster[2,5], Changhoon Kim[2], Ion Stoica[6]

[1]Johns Hopkins University, [2]Barefoot Networks, [3]Princeton University, [4]Università della Svizzera italiana, [5]Cornell University, [6] UC Berkeley

ABSTRACT

We present NetCache, a new key-value store architecture that leverages the power and flexibility of new-generation programmable switches to handle queries on hot items and balance the load across storage nodes. NetCache provides high aggregate throughput and low latency even under highly-skewed and rapidly-changing workloads. The core of NetCache is a packet-processing pipeline that exploits the capabilities of modern programmable switch ASICs to efficiently detect, index, cache and serve hot key-value items in the switch data plane. Additionally, our solution guarantees cache coherence with minimal overhead. We implement a NetCache prototype on Barefoot Tofino switches and commodity servers and demonstrate that a single switch can process 2+ billion queries per second for 64K items with 16-byte keys and 128-byte values, while only consuming a small portion of its hardware resources. To the best of our knowledge, this is the first time that a sophisticated application-level functionality, such as in-network caching, has been shown to run at line rate on programmable switches. Furthermore, we show that NetCache improves the throughput by 3-10× and reduces the latency of up to 40% of queries by 50%, for high-performance, in-memory key-value stores.

CCS CONCEPTS

• **Information systems → Key-value stores**; • **Networks → Programmable networks**; **In-network processing**; • **Computer systems organization → Cloud computing**;

KEYWORDS

Key-value stores; Programmable switches; Caching

ACM Reference Format:
Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of SOSP '17, Shanghai, China, October 28, 2017,* 17 pages.
https://doi.org/10.1145/3132747.3132764

SOSP '17, October 28, 2017, Shanghai, China

ACM ISBN 978-1-4503-5085-3/17/10…$15.00
https://doi.org/10.1145/3132747.3132764

1 INTRODUCTION

Modern Internet services, such as search, social networking and e-commerce, critically depend on high-performance key-value stores. Rendering even a single web page often requires hundreds or even thousands of storage accesses [34]. So, as these services scale to billions of users, system operators increasingly rely on *in-memory* key-value stores to meet the necessary throughput and latency demands [32, 36, 38].

One major challenge in scaling a key-value store—whether in memory or not—is coping with skewed, dynamic workloads. Popular items receive far more queries than others, and the set of "hot items" changes rapidly due to popular posts, limited-time offers, and trending events [2, 11, 19, 21]. For example, prior studies have shown that 10% of items account for 60-90% of queries in the Memcached deployment at Facebook [2]. This skew can lead to severe load imbalance, which results in significant performance degradations: servers are either over- or under-utilized, throughput is reduced, and response times suffer from long tail latencies [14]. This degradation can be further amplified when storage servers use per-core sharding to handle high concurrency [5].

The problem of load imbalance is particularly acute for high-performance, in-memory key-value stores. While traditional flash-based and disk-based key-value stores can be balanced using a fast in-memory caching layer (such as SwitchKV [28]), server-based caching does not work for in-memory stores, because there is little difference in performance between the caching and storage layers (Figure 1). Alternatively, one could use selective replication—i.e., replicating hot items to additional storage nodes. However, in addition to consuming more hardware resources, selective

replication requires sophisticated mechanisms for data movement, data consistency, and query routing, which complicates the system design and introduces overheads [10, 25, 40].

This paper presents NetCache, a new key-value store architecture that leverages the power and flexibility of new-generation programmable switches to cache data in the network. Switches are optimized for data input-output and offer orders of magnitude better performance over traditional caches and storage servers (Figure 1), making them an ideal place to build an on-path caching layer for high-performance, in-memory key-value stores. Whereas traditional caches often require a high cache hit ratio (>90%) to absorb most queries, NetCache uses switches as a *load-balancing cache* with medium cache hit ratio (<50%) [28]. A load-balancing cache services accesses to hot items and makes the remaining load on servers more uniform. Although switches have limited on-chip memory, NetCache exploits the theoretical result that caching $O(N \log N)$ items is sufficient to balance the load for N storage servers (or CPU cores) across the entire hash-partitioned key-value storage cluster, regardless of the number of items stored in the system [17]. Overall, the entire system is able to guarantee high aggregate throughput and low latency, despite workload skew.

The core of NetCache is a packet-processing pipeline that detects, indexes, stores, and serves key-value items. We use match-action tables to classify keys, which are carried in packet headers, and register arrays implemented as on-chip memory in programmable switches to store values. We exploit the multi-pipeline, multi-stage structure of modern switch ASICs to efficiently index and pack variable-length values into limited switch table and memory resources.

To identify hot items in the switch data plane, NetCache maintains counters for each cached key, and a heavy-hitter detector for uncached keys. This leverages the fact that switches are naturally placed on the data path and interpose on all queries through the system. The heavy-hitter detector includes a Count-Min sketch [12] to report hot uncached keys, and a Bloom filter [8] to remove duplicate reports. Both data structures can be implemented in the switch data plane at line rate using minimal resources. The heavy-hitter detector obviates the need for building, deploying, and managing a separate monitoring component in the servers to count and aggregate key access statistics [28].

Another advantage of our architecture is that it automatically guarantees cache coherence with low overhead. Read queries for cached items are handled directly by switches without having to visit a storage server. Write queries for cached keys invalidate any copies stored in the switches on the routes to storage servers, and the servers atomically update the switches with new values. Hence, the control plane is only responsible for inserting and evicting appropriate keys based on cache counters and heavy-hitter reports.

NetCache is incrementally deployable. It is particularly suitable for modern rack-scale storage systems that contain thousands of cores per rack and use per-core sharding for high performance. We only need to program the ToR switch to add the NetCache functionality; other parts of the network are unmodified. NetCache is fully compatible with existing routing protocols and network functions. We provide a NetCache library for clients to access the key-value store, which exposes an API similar to existing key-value stores. We also provide a simple shim layer for storage servers, which makes it easy to integrate NetCache with existing key-value stores.

In summary, we make the following contributions.

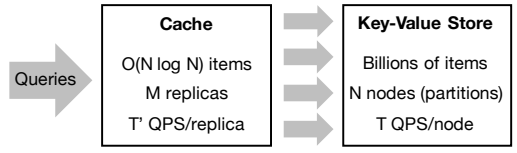
- We design NetCache, a new key-value store architecture that leverages new-generation programmable switches to cache data in the network for dynamic load balancing (§3).
- We design a packet-processing pipeline that efficiently detects, indexes, stores, and serves hot key-value items in the switch data plane, as well as a mechanism to guarantee cache coherence (§4).
- We implement a prototype of NetCache on Barefoot Tofino switches and commodity servers (§6). NetCache only uses a small portion of Tofino on-chip resources, leaving enough space for traditional packet processing.
- We perform an evaluation on the NetCache prototype (§7) and demonstrate that NetCache is able to run on programmable switches at line rate—i.e., processing 2+ billion queries per second (QPS) for 64K items with 16-byte keys and 128-byte values on a single switch. Overall, the system improves the throughput by 3-10×, and reduces the latency of up to 40% of queries by 50%, for high-performance, in-memory key-value stores.

We discuss system limitations and possible solutions in §5. In particular, NetCache focuses on a single key-value storage rack; provides a restricted key-value interface; and cannot handle highly-skewed, write-intensive workloads. The performance and capability of new-generation programmable switches make them appealing to be used beyond traditional network functionalities. We hope that our first-hand experiences with programmable switches will be valuable to inform a new generation of distributed systems that deeply integrate switches and servers.

2 MOTIVATION

NetCache is motivated by the recent trend towards high-performance, in-memory key-value stores. We argue that caching hot items in the network like NetCache is a natural solution to provide performance and strong consistency guarantees for in-memory key-value stores under highly skewed and dynamic real-world workloads.

Load balancing for performance guarantees. As a critical building block for large-scale Internet services that serves



Traditional	In-memory: O(10) MQPS	Flash/Disk: O(100) KQPS
NetCache	**Switch: O(1) BQPS**	**In-memory: O(10) MQPS**

Figure 1: Motivation. To provide effective load balancing, a cache node only needs to cache $O(N \log N)$ items, but needs to be orders of magnitude faster than a storage node ($T' \gg T$). NetCache caches data in the network when key-value stores move to the memory.

billions of users, key-value stores are expected to provide high performance guarantees to meet strict service level agreements (SLAs). Ideally, if the throughput of each storage node is T, a key-value store with N nodes should be able to guarantee an aggregate throughput of $N \cdot T$. Given that each node handles no more load than T, the query latency is also bounded. These performance guarantees make it easy for service providers to scale out a storage system to meet particular SLAs.

Unfortunately, the skewed, dynamic nature of real-world workloads make it difficult to provide such guarantees [2, 11, 19, 21]. Popular items are queried far more often than other items, leading to severe imbalance on the storage servers. The system is bottlenecked by the overloaded nodes, leaving many other nodes not fully utilized, so that the entire system cannot achieve the desired aggregate throughput. Moreover, overloaded nodes receive more queries than they can handle, leading to long queues for outstanding queries. This causes the system to have high tail latencies, which violate the latency requirements.

Fast, small cache for load balancing. Caching is an effective technique for alleviating load imbalance (Figure 1). Theoretical analysis proves that a cache only need to store $O(N \log N)$ items to balance the load for a hash-partitioned key-value cluster with N storage nodes, *regardless of the number of key-value items* [17]. Specifically, given N nodes with total system load of $N \cdot T$, if the cache is able to absorb all queries to the hottest $O(N \log N)$ items, then no node would experience more than T load with high probability, regardless of the query distribution. Since $O(N \log N)$ is relatively small, the hot items can be replicated to all cache nodes in order to avoid circular load balancing issues in the caching layer. While $O(N \log N)$ items seem small enough to be put into each client, it is difficult to ensure cache coherence, and client caching would not have the caching benefits if there are many clients accessing a common set of hot items but this set is not hot to each client. Therefore, it is more effective to build a caching layer in front of the storage servers.

To handle arbitrarily skewed workloads, the caching layer must provide an aggregate throughput comparable to the storage layer. Given M caching nodes with per-node throughput T', we need

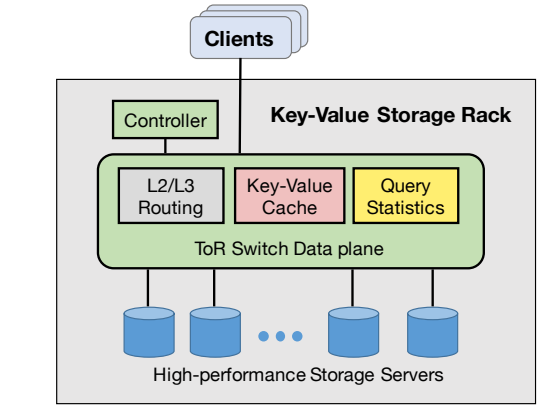
$$M \simeq N \cdot \frac{T}{T'}.$$

If $T' \simeq T$, then $M \simeq N$, which implies that we would need to build a caching layer with a similar number of nodes as the storage layer. This has (i) high cost, as it uses too many caching nodes, and (ii) high overhead, as M nodes must be modified for each cache update. Therefore, it requires $T' \gg T$ (i.e., orders of magnitude difference) to build a cost-effective, low-overhead caching layer.

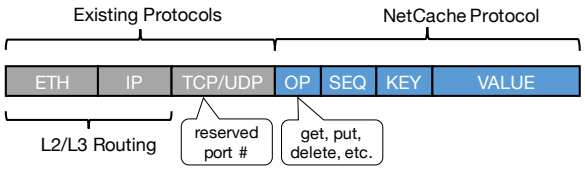
In-network caching for in-memory key-value stores. In-memory caches are effective for flash-based and disk-based key-value stores since DRAMs are orders of magnitude faster than SSDs and HDDs. However, as key-value stores themselves are being moved to the main memory, in-memory caches lose their performance advantage and are no longer effective. Consider an in-memory key-value store with 128 servers: if each server provides 10 million QPS, the entire store can achieve 1.28 billion QPS throughput, and the caching layer needs to provide comparable throughput.

Building the caching layer into the network is a natural solution for balancing in-memory key-value stores. Switches are optimized for I/O—e.g., current ASICs such as Barefoot Tofino [4] and Broadcom Tomahawk II [7] are able to process several billion packets per second. This means that we can build the caching layer with a single box for high-performance, in-memory key-value stores. Furthermore, if we use the ToR switch as the cache for a key-value storage rack, it incurs *no latency penalties and no additional hardware cost.* Other possible alternatives like FPGAs or NPUs either do not provide enough throughput with a single box or are not immediately deployable (e.g., are not yet available or are too expensive).

Programmable switch ASICs for in-network caching. Modern commodity switches have tens of megabytes on-chip memory. Since many in-memory key-value stores target at small values (e.g., <100-byte values for 76% read queries in the Memcached deployment at Facebook [34]), the on-chip memory is big enough to store the $O(N \log N)$ items for load balancing, while can still leave enough switch resources for traditional network functionalities. We show in Figure 10(e) in §7 that empirically caching a few thousand items is enough for a rack with 128 servers (or partitions). For large items that do not fit in one packet, one can always divide an item into smaller chunks and retrieve them with multiple packets. Note that multiple packets would always be necessary when a large item is accessed from a storage server.



(a) NetCache architecture.



(b) NetCache packet format.

Figure 2: NetCache overview.

Traditional switch ASICs are fixed function, so adding a new feature requires designing and manufacturing a new ASIC, which incurs huge capital and engineering costs. However, new-generation programmable switch ASICs like Barefoot Tofino [4] and Cavium XPliant [9] make in-network caching viable and immediately deployable. They allow users to program the switch packet-processing pipeline without replacing the switch ASICs. Specifically, we are able to (*i*) program the switch parser to identify custom packet formats (e.g., containing custom fields for keys and values), (*ii*) program the on-chip memory to store custom state (e.g., store hot items and query statistics), and (*iii*) program the switch tables to perform custom actions (e.g., copy values from on-chip memory to packets and detect heavy hitters). The goal of this paper is to leverage programmable switches to make in-network caching a reality.

3 NETCACHE OVERVIEW

NetCache is a new rack-scale key-value store architecture that leverages in-network caching to provide dynamic load balancing across all storage servers. We assume the rack is dedicated for key-value storage and the key-value items are hash-partitioned to the storage servers. We use the ToR switch that is directly connected to the servers as a load-balancing cache. Figure 2(a) shows the architecture overview of a NetCache storage rack, which consists of a ToR switch, a controller, and storage servers.

Switch. The switch is the core component of NetCache. It is responsible for implementing on-path caching for key-value items and routing packets using standard L2/L3 protocols. We reserve an L4 port to distinguish NetCache packets (Figure 2(b)) from other packets (§4.1). Only NetCache packets are processed by NetCache modules in the switch. This makes NetCache fully compatible with other network protocols and functions.

The key-value cache module stores the hottest items. Read queries are handled directly by the switch while write queries are forwarded to the storage servers (§4.2). Cache coherence is guaranteed with a light-weight *write-through* mechanism (§4.3). We leverage match-action tables and register arrays to index, store, and serve key-value items (§4.4.2).

The query statistics module provides key-access statistics to the controller for cache updates (§4.4.3). This is critical for enabling NetCache to handle dynamic workloads where the popularity of each key changes over time. It contains (*i*) per-key counters for the cached items and (*ii*) a heavy hitter (HH) detector to identify hot keys not present in the cache. The HH detector uses a Count-Min sketch to report HHs and a Bloom filter to remove duplicate HH reports, both of which are space-efficient data structures and can be implemented in programmable switches with minimal hardware resources.

Since the switch is a read cache, if the switch fails, operators can simply reboot the switch with an empty cache or use a backup ToR switch. The switch only caches hot items and computes key access statistics; it does not maintain any critical system state. Because NetCache caches are small, they will refill rapidly after a reboot.

Controller. The controller is primarily responsible for updating the cache with hot items (§4.3). It receives HH reports from the switch data plane, and compares them with per-key counters of the items already in the cache. It then decides which items to insert into the cache and which ones to evict. Note that the NetCache controller is different from the network controller in Software-Defined Networking (SDN): the NetCache controller does not manage network protocols or distributed routing state. The operator uses existing systems (which may be an SDN controller or not) to manage routing tables and other network functions. The NetCache controller does not interfere with these existing systems and is only responsible for managing its own state—i.e., the key-value cache and the query statistics in the switch data plane. It can reside as a process in the switch OS or on a remote server. It communicates with the switch ASIC through a switch driver in the switch OS. As all queries are handled by the switch and storage servers, the controller only handles cache updates and thus is not the bottleneck.

Storage servers. NetCache servers run a simple shim that provides two important pieces of functionality: (*i*) it maps

NetCache query packets to API calls for the key-value store; (*ii*) it implements a cache coherence mechanism that guarantees consistency between the caching and storage layers, hiding this complexity from the key-value stores. This shim layer makes NetCache easy to integrate with existing in-memory key-value stores.

Clients. NetCache provides a client library that applications can use to access the key-value store. The library provides an interface similar to existing key-value stores such as Memcached [32] and Redis [38]—i.e., `Get`, `Put`, and `Delete`. It translates API calls to NetCache query packets and also generates replies for applications.

4 NETCACHE DESIGN

4.1 Network Protocol

Packet format. Figure 2(b) shows the packet format of NetCache. NetCache is an application-layer protocol embedded inside the L4 payload. Similar to many other key-value stores, NetCache uses UDP for read queries (to achieve low latency) and TCP for write queries (to achieve reliability) [34]. The NetCache fields are inside the TCP/UDP payload and a special TCP/UDP port is reserved for NetCache. NetCache switches use this port to invoke the custom packet processing logic for NetCache queries; other switches do not need to understand the NetCache format and treat NetCache queries as normal packets. The major header fields for NetCache are `OP`, `SEQ`, `KEY` and `VALUE`. `OP` stands for *operator* and denotes whether it is a `Get`, `Put`, `Delete` or any other type of query. `SEQ` can be used as a sequence number for reliable transmissions by UDP `Get` queries, and as a value version number by TCP `Put` and `Delete` queries. `KEY` and `VALUE` store the key and value of an item respectively. `VALUE` is empty for `Get` and `Delete`. `Get` queries and replies have the same packet format, except that switches and storage servers add the `VALUE` field in the reply packets.

Network routing. NetCache leverages existing routing protocols to forward packets. For a NetCache query, based on the data partition, the client appropriately sets the Ethernet and IP headers and sends the query to the storage server that owns the queried item, without any knowledge of NetCache. NetCache switches are placed on the path from the clients to the storage clusters. They process NetCache queries based on Algorithm 1; other switches simply forward packets based on the destination MAC/IP address according to the L2/L3 routing protocol. In this way, NetCache can coexist with other network protocols and functions.

4.2 Query Handling

The key advantage of NetCache is that it provides an on-path in-network cache to serve key-value queries at line rate.

Algorithm 1 ProcessQuery(pkt)

```
   – cache: on-chip key-value cache
   – stats: on-chip query statistics
 1: if pkt.op == Get then
 2:     if cache.hit(pkt.key) and cache[pkt.key].valid() then
 3:         add pkt.value header
 4:         pkt.value ← cache[pkt.key]
 5:         stats.cache_count(pkt.key)
 6:     else
 7:         stats.heavy_hitter_count(pkt.key)
 8:         if stats.is_hot_key(pkt.key) then
 9:             inform controller for potential cache updates
10: else if pkt.op == Put or pkt.op == Delete then
11:     if cache.hit(pkt.key) then
12:         cache.invalidate(pkt.key)
13: Update packet header and forward
```

Read queries (`Get`) on cached items are directly returned by switches without traversing any storage server, providing very low latency; write queries (`Put` and `Delete`) are passed to storage servers with no performance overhead. Figure 3 illustrates how NetCache handles different types of queries, and Algorithm 1 gives the pseudo code.

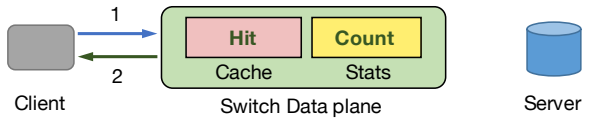
Handling read queries. The switch checks whether the cache contains the item or not. If it is a cache hit and the value is valid (Figure 3(a)), the switch inserts a *pkt.value* field to the packet header, copies the value from the cache to *pkt.value*, and increases the counter of the item (line 2-5). Except for the inserted *pkt.value* field, other fields of the packet are retained. Then the switch updates the packet header by swapping the source and destination addresses and ports in L2-L4 header fields, and returns the packet back to the client as the reply.

If it is a cache miss (Figure 3(b)), the switch counts the key access with its HH detector, and informs the controller if the key is hot (line 7-9). By counting queries with uncached keys, the HH detector only reports new popular keys not in the cache, which saves both the memory consumption in the switch for HH detection and the computation in the controller for cache updates. The query is then forwarded to the corresponding storage server, which processes the query and sends the reply back to the client.

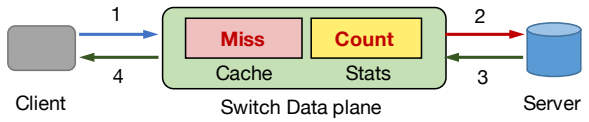
Handling write queries. The switch checks whether the item is in the cache (Figure 3(c)). If so, the switch invalidates the cached value, stopping subsequent read queries from fetching the old value from the cache (line 10-12). The query is then forwarded to the corresponding storage server, which updates the value atomically and replies to the client.

4.3 Cache Coherence and Cache Update

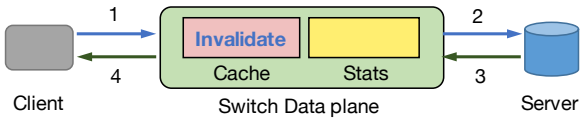
Cache coherence. NetCache uses *write-through* to guarantee cache coherence. The switch invalidates the cached value



(a) Get query with cache hit. Switch directly replies query.



(b) Get query with cache miss. Switch does statistics only.



(c) Put/Delete query. Switch invalidates key if in cache.

Figure 3: Handling different types of queries.

when it processes a write query for a cached key, and modifies the operation field in the packet header to special values to inform the server that the queried key is in the cache.

Cache invalidation ensures that while a write query is in progress, all subsequent read and write queries are sent to the server, which updates the value atomically and serializes the queries for consistency. The server recognizes from the packet header that the key is cached, then sends another packet to update the switch cache with the new value, and retries the update upon packet loss for reliability. The server replies to the client as soon as it completes the write query, and does not need to wait for the switch cache to be updated. The server blocks all subsequent write queries until it confirms the switch cache is updated with the new value, in order to ensure the consistency between the server and the switch cache.

This design provides lower latency for write queries than a standard *write-through* cache which updates both the cache and the storage server before replying to the client. After the update, the item becomes valid again and begins to serve following read queries. The updates are purely in the data plane at line rate. We do not use *write-back* to avoid possible data loss caused by switch failures, and do not use *write-around* because data plane updates incur little overhead and are much faster than control plane updates. Note that our data plane design (§4.4.2) only allows updates for new values that are no larger than the old ones. Otherwise, the new values must be updated by the control plane.

Cache Update. To cope with dynamic workloads, the controller frequently updates the cache with the hottest keys. The primary challenge is the limited switch table and register update rate. Commodity switches are able to update more

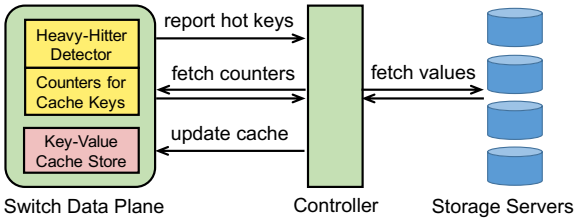


Figure 4: Cache update.

than 10K table entries per second [35]. While this number is continuously being improved over time, the update rate is not sufficient to support traditional cache update mechanisms like LRU and LFU. These mechanisms update the cache for every query, which may cause unnecessary cache churns and hurt overall cache performance. Compared to a traditional cache with high cache hit ratio (>90%), NetCache provides a load balancing cache with medium cache hit ratio (<50%). Therefore we insert an item to the cache only when it becomes hot enough (based on HH detection in the data plane), rather than for each item access.

Specifically, the controller receives HH reports from the data plane via the switch driver in the switch OS (Figure 4). It compares the hits of the HHs and the counters of the cached items, evicts less popular keys, and inserts more popular keys. As the cache may contain tens of thousands of items, it is expensive to fetch all counters and to compare them with HHs. To reduce this overhead, we use a sampling technique similar to Redis [39], i.e., the controller samples a few keys from the cache and compare their counters with the HHs. The values of the keys to insert are fetched from the storage servers. To guarantee cache coherence during cache updates, when the controller is inserting a key to the cache, write queries to this key are blocked at the storage servers until the insertion is finished, which is the same as handling write queries to cached items. Future write queries to the key are handled by data plane updates as discussed previously.

4.4 Switch Data Plane Design

The core of NetCache is a packet-processing pipeline that realizes (*i*) a variable-length key-value store to serve cached items with guaranteed cache coherence, and (*ii*) a query statistics engine to provide essential key-access information for cache updates, as introduced in Figure 2(a) and Algorithm 1. We first give a brief background on the emerging programmable switch data plane and then present our design for the NetCache data plane.

4.4.1 A Primer on Switch Data Plane

Figure 5 illustrates the basic data plane structure of a modern switch ASIC. The data plane contains three main components: ingress pipeline, traffic manager, and egress pipeline

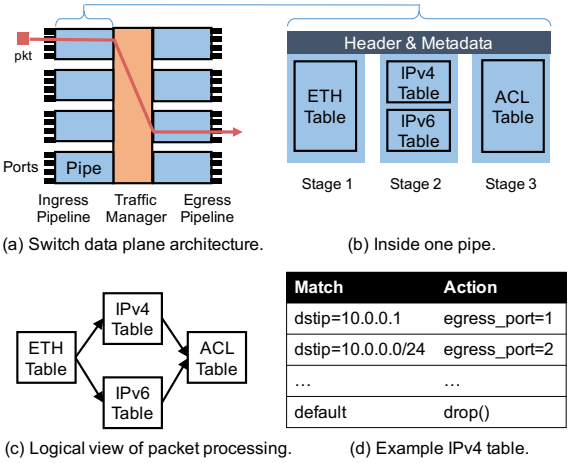


Figure 5: A primer on switch data plane.

(Figure 5(a)). A switch usually has multiple ingress and egress pipes, each with multiple ingress and egress ports. When a packet arrives at an ingress port, it is first processed by the tables in the ingress pipe, then queued and switched by the traffic manager to an egress pipe, and finally processed by the pipe and emitted via an egress port.

Each pipe contains multiple stages (Figure 5(b)). These stages process packets in a sequential order, and have their own dedicated resources, including match-action tables and register arrays. When processing a packet, the stages share the header fields and metadata of the packet, and can pass information from one stage to another by modifying the shared data. The entire packet processing can be abstracted using a graph of match-action tables. Figure 5(c) gives an example graph. These tables are mapped to the stages by compilation tools. Tables in the same stage cannot process packets sequentially. Each table matches on a few header fields and metadata, and performs actions based on the matching results. Figure 5(d) shows an example IPv4 table that picks an egress port based on destination IP address. The last rule drops all packets that do not match any of the IP prefixes in the match-action table.

Programmable switches allow developers to define their own packet formats, build custom processing graphs, and specify the match fields and actions for each table. Developers typically write a program with a domain-specific language like P4 [6], and then use a compiler provided by switch vendors to compile the program to a binary that can be loaded to switches. Developers need to carefully design their processing pipelines in order to meet the hardware resource and timing requirements of switch ASICs. The major constraints of the pipeline design include (*i*) the number of pipes, (*ii*) the number of stages and ports in each pipe, and (*iii*) the amount of TCAMs (for wildcard and prefix matching) and SRAMs (for prefix matching and data storage) in each stage.

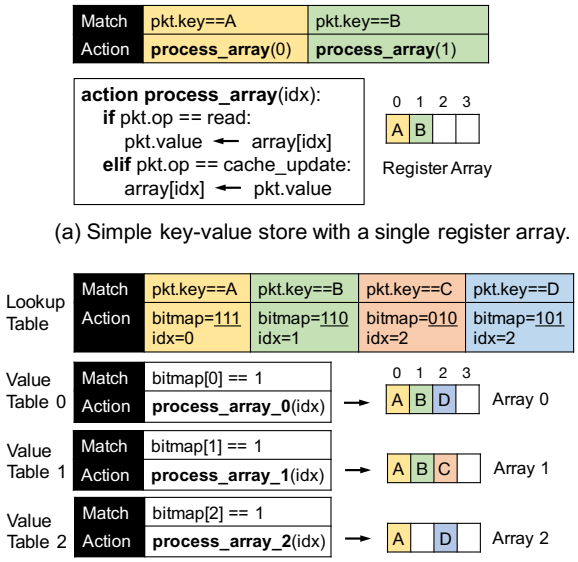


Figure 6: NetCache switch data plane key-value store.

4.4.2 *Variable-Length On-Chip Key-Value Store*

NetCache leverages the *stateful memory* in emerging programmable switch ASICs such as Barefoot Tofino chip [4] to store key-value items. The stateful memory is abstracted as *register arrays* in each stage. The data in the register array can be directly retrieved and updated at its stage at line rate through an index that indicates the memory location. Figure 6(a) shows how to construct a simple key-value store with a match-action table and a register array. The table uses *exact-match* to match on the key field in the packet header and gives an *index* for each matched key as the action data. The action is to process the data at the given index location based on the operation type in the packet header.

The size of data that each register array can read or write for each packet is fixed and limited for each stage, because a switch ASIC has to meet strict resource and timing requirements. In order to realize a variable-length key-value store with relatively large values, we need to use multiple register arrays, and concatenate the values from multiple stages to construct a large value. The challenge is how to design the key-value store with maximum resource efficiency.

There are three types of resource overheads to be minimized: (*i*) the number of entries in match-action tables, (*ii*) the size of action data given by a table match, and (*iii*) the size of intermediate metadata. A straightforward approach for variable-length key-value stores is to replicate the table in Figure 6(a) for each register array, which is obviously inefficient since the match entry for a key needs to be replicated multiple times. A better approach is to use one lookup table that matches on the keys and generates a list of indexes for

Algorithm 2 Switch Memory Management

- key_map: key ⇒ (index, bitmap)
- mem: array of bitmap, which marks available slots as 1 bits and occupied slots as 0 bits

```
1:  function EVICT(key)
2:      if key_map.hasKey(key) then
3:          index, bitmap = key_map[key]
4:          mem[index] = mem[index] | bitmap
5:          return true
6:      else
7:          return false                          ▷ The item is not cached
8:  function INSERT(key, value_size)
9:      if key_map.hasKey(key) then
10:         return false
11:     n = value_size/ unit_size
12:     for index from 0 to sizeof(mem) do
13:         bitmap = mem[index]
14:         if number of 1 bits in bitmap ≥ n then
15:             value_bitmap = last n 1 bits in bitmap
16:             mark last n 1 bits in mem[index] as 0 bits
17:             key_map[key] = (index, value_bitmap)
18:             return true
19:     return false                  ▷ No space available to cache the item
```

all the arrays. However, it still has high overhead in terms of action data and intermediate metadata, as it requires a separate index for each register array.

Figure 6(b) shows our approach to construct an efficient variable-length on-chip key-value store. It has one lookup table with match entries for each key. Each match action produces two sets of metadata: (*i*) a bitmap indicating the set of arrays that has the value for this key, and (*ii*) a single index pointing to the location of the data within these arrays. Each register array is processed with the same given index, after a simple check on the corresponding bit in the bitmap. The data in the register arrays is *appended* to the value field (Figure 2(b)) when the packet is processed by the match-action tables in the pipeline.

Memory management. NetCache's in-network key-value store minimizes hardware resource usage. The only restriction is that we cannot freely assign indexes in register arrays for a key. A key's value has to use the same index for all its register arrays. The NetCache controller manages the mapping between slots in register arrays and the cached items, as described in Algorithm 2. Evicting an item is simple as the controller only needs to free the slots occupied by the item (line 1-7). Cache insertion is more complicated, as the controller needs to decide which slots to allocate to the new item (line 8-19). The problem can be formulated as a bin packing problem. The values are balls with different sizes. The bins are slots in register arrays with the same index, e.g., bin 0 includes slots of index 0 in all register arrays. This is because the data plane design requires that an item is stored in the same index for all its register arrays (Figure 6). We use First Fit, a classical heuristic algorithm for bin packing problems, to allocate memory slots (line 12-18). It is an online algorithm to handle periodic item insertions and evictions. Our bitmap is flexible as it does not require an item to occupy the same index in *consecutive* tables, which alleviates the problem of memory fragmentation, though periodic memory reorganization is still needed to pack small values with different indexes into register slots with same indexes, in order to make room for large values.

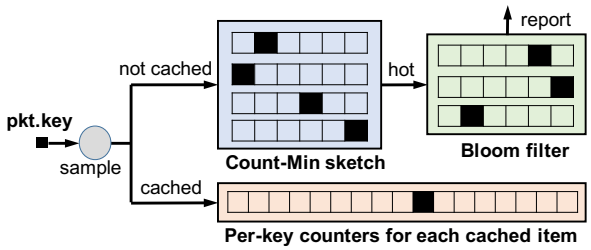


Figure 7: Query statistics in the data plane.

4.4.3 *Query Statistics*

The NetCache switch data plane provides query statistics for the controller to make cache update decisions. The query statistics module contains three major components (Figure 7): (*i*) a per-key counter array to keep query frequency of each cached item, (*ii*) a Count-Min sketch [12] to detect hot queries for uncached items, and (*iii*) a Bloom filter [8] that removes duplicate hot key reports (to reduce controller load). All statistics data are cleared periodically by the controller. The clearing cycle has direct impact on how quickly the cache can react to workload changes.

The three components are built upon register arrays and hashing functions in the switch data plane. In addition to data read and write operations described in previous sections, register arrays also support simple arithmetic operations such as add, subtract and compare. The per-key counter is just a single register array. Each cached key is mapped to a counter index given by the lookup table. A cache hit simply increases the counter value of the cached key-value item in the corresponding slot by one.

The Count-Min sketch component consists of four register arrays. It maps a query to different locations in these arrays, by hashing the key with four independent hash functions. It increases the values in those locations by one, uses the smallest value among the four as the key's approximate query frequency, and marks it as hot if the frequency is above the threshold configured by the controller.

A hot query for an uncached key should be reported to the controller for potential cache updates. Once a key frequency is above the threshold, it will keep being marked as hot by the Count-Min sketch before the counters are refreshed by the

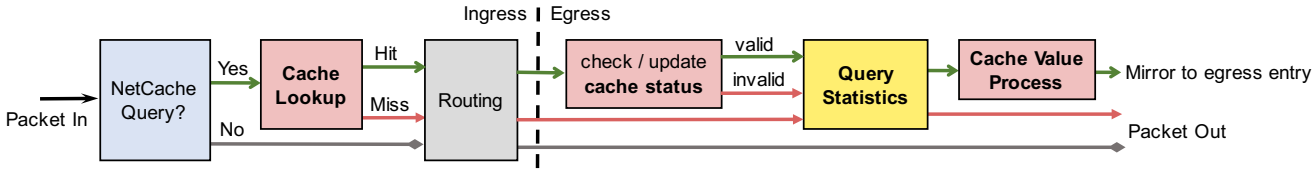


Figure 8: Logical view of NetCache switch data plane.

controller. We add a Bloom filter after the Count-Min sketch, so that each uncached hot key would only be reported to the controller once.

Ideally, we would like to minimize the slot number and slot size of each register array to maximize resource efficiency. However, a small slot number would lead to high false positives for Count-Min sketches and Bloom filters, and a small slot size would make counter values quickly overflow. To meet this challenge, we add a sampling component in front of other components. Only sampled queries are counted for statistics. The sampling component acts as a high-pass filter for the Count-Min sketch, as it can filter out queries for most of the non-frequent keys. It also allows us to use small (16-bit) slot size for cache counters and the Count-Min sketch. Same as the heavy-hitter threshold, the sample rate can be dynamically configured by the controller.

4.4.4 *Put Everything Together: Pipeline Layout*

Figure 8 shows the pipeline layout in the switch data plane and the packet flow in the switch for a NetCache query. Essentially, it realizes Algorithm 1 with the table designs described in §4.4.2 and §4.4.3. The cache lookup table is placed at the ingress pipeline, and is *replicated for each upstream ingress pipe of the switch*, such that the switch can handle client queries from any upstream ports. Its size is equal to the cache size. Since both the keys and actions are small, replicating the lookup table does not consume a lot of resources. On the other hand, the value tables and their register arrays are large and replicating them is too expensive. We place them at the egress pipeline. Each egress pipe *only stores the cached values for servers that connect to it*. This avoids value replications across pipes, reduces the switch memory consumption, and simplifies the routing behavior.

An ingress pipe first checks if an incoming packet is a NetCache query by looking at its header fields, and a cache lookup is performed for NetCache queries. The lookup table produces three sets of metadata for cached keys: a table bitmap and a value index as depicted in Figure 6, a key index used for cache counter as depicted in Figure 7 and for cache status array which will be described later, and an egress port that connects to the server hosting the key.

All packets will then traverse the routing module. When handling a read query for cached keys, the routing module performs the next-hop route lookup by matching on the source address because the switch will directly reply the query back to the client. The switch then saves the routing information as metadata, and sends the packet to the egress port given by the cache lookup table (which belongs to the pipe containing the cached value). The routing module forwards all other packets to an egress port by matching on the destination address.

At the egress pipe, queries that hit the cache are first processed by the cache status module. It has a register array that contains a slot for each cached key, indicating whether the cache is still valid. Write queries invalidate the bit and read queries check if the bit is valid. All NetCache read queries are processed by the statistics module as described in §4.4.3. Only queries for valid keys will go for value processing, as described in Figure 6(b). Since each cached item is bound to an egress pipe, in cases of extreme skew (e.g., all queries are destined to items bound to one pipe), the cache throughput is bounded by that of an egress pipe, which is 1 BQPS for a Tofino ASIC in our prototype.

So far all packets are assigned with the egress port connecting to the destination server. The read queries that get valid cache values need to be sent back to the client, through the port given by the routing table. We use *packet mirroring* to redirect the packets to the upstream port based on the routing information saved in the metadata.

5 DISCUSSION

NetCache has a few limitations. For example, it focuses on a single key-value storage rack; does not support arbitrary keys and big values; and cannot well handle highly-skewed, write-intensive workloads. We discuss these issues, their possible solutions, and our experiences with programmable switches in this section.

Scaling to multiple racks. We focus on rack-scale key-value stores in this paper. The ToR switch, which is directly connected to the storage servers, serves as a load-balancing cache for the storage rack. With this cache, NetCache is able to guarantee billions of QPS with bounded query latencies for the rack, which is sufficient for many use cases. Nevertheless, NetCache can scale out to multiple racks for large-scale deployments. Since ToR switches can only balance the load for servers in their own racks, the load on individual racks can become imbalanced when we have tens of such racks. This requires us to cache hot items to higher-level switches in a datacenter network, e.g., spine switches. Ideally, we would

like to provide a "one big memory" abstraction where the on-chip memories of all switches work as a single big cache. We need to carefully design a cache allocation mechanism to minimize routing detours and maximize switch memory utilization, as well as a cache coherence mechanism to ensure cache coherence with multiple switch caches. A full exploration is our future work.

Restricted key-value interface. The current NetCache prototype provides a restricted key-value interface, where the keys have fixed 16-byte length and the values have 128-byte maximum size. Variable-length keys can be supported by mapping them to fixed-length hash keys. The original keys can be stored together with the values in order to handle hash collisions. Specifically, when a client fetches a value from the switch cache, it should verify whether the value is for the queried key, by comparing the original key to that stored with the value. And if a hash collision happens, the client should send a new query directly to the storage servers to fetch the intended value. Supporting variable length keys in the data plane can be achieved through consuming more hardware resources and more complicated parsing logic. We leave this as a future work.

For the value size, by assigning different bitmaps and indexes to different keys, NetCache can store values with different sizes in the on-chip memory, at the granularity of output data size of one register array. The maximum value size is the total output data size of all register arrays, usually at the order of *hundreds of bytes*, which is large enough for many in-memory key-value applications for web services [2, 34]. Larger values (up to MTU size) can be supported by using packet mirroring/recirculation to let a packet go through the pipe for multiple rounds, with each round appending a few hundred bytes to the value field. Note that packet mirroring/recirculation reduces the effective throughput as a mirrored/recirculated packet consumes resources that can be otherwise used for a new packet. This attributes to a fundamental trade-off between throughput and resources (memory and compute) in modern switch ASIC design.

Write-intensive workloads. NetCache provides load balancing for read-intensive workloads, which are common in many real-world scenarios (e.g., the GET/SET ratio is 30:1 in the Memcached deployment at Facebook [2]). As all write queries have to be processed by storage servers, the load on storage servers would be imbalanced under high-skewed, write-intensive workloads. Conceptually, this problem could be solved by directly handling write queries on hot items in the switch. However, the results of write queries would be lost under switch failures. This can be solved by writing to multiple switches for fault-tolerance but requires a mechanism to ensure the consistency of multiple copies in switches, which we leave as future work.

Encryption. Encryption is often used when critical information is stored in a key-value store, especially in public clouds. Since NetCache does not need to interpret the meanings of the key-value items, NetCache can serve encrypted values indexed by encrypted keys, as long as the packet format can be recognized and parsed by the switch. NetCache cannot hide key access patterns without new mechanisms.

Experiences with programmable switches. During the design and development of NetCache, we sometimes found it challenging to fit the key-value store and the query statistics modules into switch tables and register arrays. This can be improved in next-generation programmable switches from both the software and hardware sides. In the software side, the current programming API and compiler of programmable switches are low-level. Designing the data plane modules for NetCache requires us to carefully deal with the stages, the size of register arrays in each stage, and the bytes that can be read and written in each register array. The programming API should provide higher-level abstractions for developers to write network applications and the compiler should intelligently map the applications to the switch data plane under various switch resource constraints. In the hardware side, we expect next-generation programmable switches to support larger slots for register arrays so that the chip can support larger values with fewer stages.

Programmable switches beyond networking. The performance and programmability of new-generation switches make them appealing to be used beyond traditional network processing, just as the way GPUs are used beyond graphics. Switches have high IO but limited programmability and resources, and servers have low IO but general-purpose computation and abundant resources. A new generation of heterogeneous distributed systems can be designed with these two types of devices, which have *disparate* but *complementary* capabilities. NetCache is only one concrete example of such heterogeneous systems. We expect programmable switches can be deeply integrated into cloud systems to build high-performance and robust cloud services in the future.

6 IMPLEMENTATION

We have implemented a prototype of NetCache, including all switch data plane and control plane features described in §4, a client library that interfaces with applications, and a server agent that provides a shim layer to in-memory key-value stores and guarantees cache coherence.

The switch data plane is written in P4 [6] and is compiled to Barefoot Tofino ASIC [4] with Barefoot Capilano software suite [3]. The cache lookup table has 64K entries for 16-byte keys. The value tables and register arrays spread across 8 stages. Each stage provides 64K 16-byte slots. This results

in a total of 8 MB for the cache with value size at the granularity of 16 bytes and up to 128-byte values. The Count-Min sketch contains 4 register arrays, each with 64K 16-bit slots. The Bloom filter contains 3 register arrays, each with 256K 1-bit slots. The Count-Min sketch and the Bloom filter uses hash functions provided by the Tofino ASIC, which perform random XORing of bits of the key field. The controller can configure the frequency to reset the Count-Min sketch and the Bloom filter. We reset them every second in the experiments. In total, our data plane implementation uses less than 50% of the on-chip memory available in the Tofino ASIC, leaving enough space for traditional network processing. We use standard L3 routing as the routing module, which forwards packets based on destination IP address.

The controller is written in Python. The P4 compiler generates Thrift APIs for the controller to update the data plane through the switch driver at runtime. The controller uses these APIs to receive heavy-hitter reports from the data plane, fetch counters, and updates the cached items, according to the mechanism described in §4.3.

The client library and the server agent are implemented in C with Intel DPDK [20] for optimized IO performance. The client library provides a key-value interface, and translates API calls to NetCache packets. The client can generate key-value queries according to a Zipf distribution with mixed read and write operations at up to 35 MQPS with the 40G NICs on our servers.

We have implemented a simple (not optimized) in-memory key-value store with TommyDS [41] for our evaluations. It provides up to 10 MQPS throughput. We have implemented a server agent which provides a shim layer for NetCache to hook up with the key-value store. Read queries only involve simple translations between NetCache packets and API calls of the key-value store. An additional mechanism is implemented to guarantee cache coherence for write queries and cache updates as described in §4. We have implemented a light-weight high-performance reliable packet mechanism to ensure a new value is updated to the switch data plane for write queries. Similar to the client library, our current implementation of the server agent provides up to 35 MQPS IO with the 40G NICs. Our server agent supports per-core sharding with Receive Side Scaling or DPDK Flow Director to handle highly concurrent workloads.

7 EVALUATION

In this section, we provide evaluation results on NetCache. The results demonstrate that NetCache runs on programmable switches at line rate (§7.2), provides significant performance improvements on throughput and latency for high-performance key-value stores (§7.3), and efficiently handles a wide range of dynamic workloads (§7.4).

7.1 Methodology

Testbed. Our testbed consists of one 6.5Tbps Barefoot Tofino switch and three server machines. Each server machine is equipped with a 16-core CPU (Intel Xeon E5-2630) and 128 GB total memory (four Samsung 32GB DDR4-2133 memory). One machine, which is equipped with two 40G NICs (Intel XL710), is used as a client to generate key-value queries; the other two machines, one with a 40G NIC (Intel XL710) and the other with a 25G NIC (Intel XXV710), are used as key-value storage servers.

Workloads. We use both uniform and skewed workloads. The skewed workloads follow Zipf distribution with different skewness parameters (i.e., 0.9, 0.95, 0.99), which are typical workloads for testing key-value stores [28, 37] and are evidenced by real-world deployments [2, 11]. We use Zipf 0.99 in most experiments to demonstrate that NetCache provides high throughput even under extreme scenarios, indicating that the storage rack can always meet its performance goals without much resource over-provisioning. We also show that NetCache still significantly improves the performance under less-skewed workloads (e.g., Zipf 0.9 and 0.95). Our client uses approximation techniques to quickly generate queries under a Zipf distribution [18, 28]. We use 16-byte keys and the keyspace is hash partitioned across storage servers. We use 128-byte values and a cache size of 10,000 items by default, and also include experiments to show the effects of value size and cache size to the system performance. Most experiments use read-only workloads, as NetCache targets for load balancing read-intensive workloads. Nevertheless, we show the system performance while varying the ratio and skewness of write queries.

We also evaluate NetCache with three dynamic workloads same as SwitchKV [28], by changing the popularity ranks of keys in the Zipf distribution. We denote N as the change size and M as the cache size.

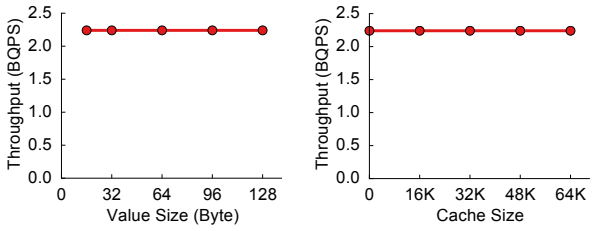
- **Hot-in.** For each change, the N coldest keys are moved to the top of the popularity ranks; other keys decrease their popularity ranks accordingly. This is a radical change since the system needs to immediately put the N keys to the cache in order to balance the storage servers.
- **Random.** For each change, N hot keys are randomly selected from the top M hottest keys, and are replaced with random N cold keys. This is a moderate change since the chosen N hot keys is unlikely to be the hottest N keys.
- **Hot-out.** For each change, the N hottest keys are moved to the bottom of the popularity ranks; other keys increases their popularity ranks accordingly. This is a small change since the hottest M-N keys are still in the cache.

Setups. A Barefoot Tofino switch is able to provide 256 25Gbps ports. Ideally, we would like to build a full storage rack with 128 storage serves, each of which connects to one

25Gbps port; the other half ports can be used as upstream ports to connect clients or other switches. Given the limited servers we have, we use the following three setups to evaluate NetCache from different aspects.

- **Snake test for switch microbenchmark (§7.2).** Snake test is a standard practice in industry to benchmark switch performance. For a switch with n ports, port 0 and port n-1 are connected to two servers separately, and port $2i$-1 is connect to port $2i$ for $0 < i < n/2$. By having the two servers send traffic to each other, all ports are sending and receiving traffic because of the *snake* structure. Specifically, a query packet sent by one server will enter the switch at port 0, and leave the switch at port 1. Since port 1 is directly connected to port 2 by a cable, the packet will enter the switch again at port 2. The switch will treat the packet as a new query packet, process it, and send it out of port 3, so on and so forth. Finally, the packet will be sent to port $n-1$ and be processed by the receiver. In this way, the query packets are looped back to the switch at each egress ports except for the last one. This allows us to stress test the switch performance at *full traffic load*. The process to read and update values is executed each time when the packet passes an egress port. To avoid packet size from increasing for read queries, we remove the value field at the last egress stage for all intermediate ports. The servers can still verify the values as they are kept in the two ports connected to them. Such a setup mimics a scenario where servers send traffic to all switch ports. We use this setup to demonstrate that programmable switches can process NetCache queries at line rate.
- **Server rotation for static workloads (§7.3).** We use the machine with two 40G NICs as a client, and the other two machines as two storage servers. We install the hot items in the switch cache as for a full storage rack and have the client send traffic according to a Zipf distribution. For each experiment, each storage server takes one key-value partition and runs as one node in the rack. The client server generates queries only destined to the corresponding partitions in the experiment based on the Zipf distribution. By rotating the two storage server for all 128 partitions (i.e., performing the experiment for 64 times), we aggregate the results to obtain the result for the entire key-value store rack. Such result aggregation is justified by (i) the *shared-nothing* architecture of key-value stores and (ii) the microbenchmark that demonstrates that the switch is not the bottleneck.

  To find the maximum effective system throughput, we first find the bottleneck partition and use that partition as one of the two partitions in the first iteration. The client generates queries destined to the two partitions, and adjusts its sending rate to saturate the bottleneck partition. We obtain the traffic load for the full system based on this sending rate, and use this load to generate per-partition query load for remaining partitions. Since the remaining partitions are not the bottleneck partition, they should be able to fully serve the load. We sum up the throughputs of all partitions to obtain the aggregate system throughput.
- **Server emulation for dynamic workloads (§7.4).** Server rotation is not suitable for evaluating dynamic workloads. This is because we would like to measure the transient behavior of the system, i.e., how the system performance fluctuates during cache updates, rather than the system performance at the stable state. To do this, we emulate 128 storage servers with two servers by using 64 queues in each server. Each queue processes queries for one key-value partition and drops queries if the received queries exceed its processing rate. Based on the number of received query replies, the client server adjusts its sending rate to find out the saturated system throughput. The key-value queries are generated according to a Zipf distribution, and the popularity ranks of keys are adjusted according to the three dynamic workloads. Because each server emulates 64 storage servers by using 64 queues, the throughput of each queue (i.e., each emulated storage server) becomes 1/64 of that of an actual storage server. Therefore, the aggregate system throughput is scaled down by a factor of 64. Such emulation is reasonable because in these experiments we are more interested in the relative system performance fluctuations when NetCache reacts to workload changes, rather than the absolute system performance numbers.



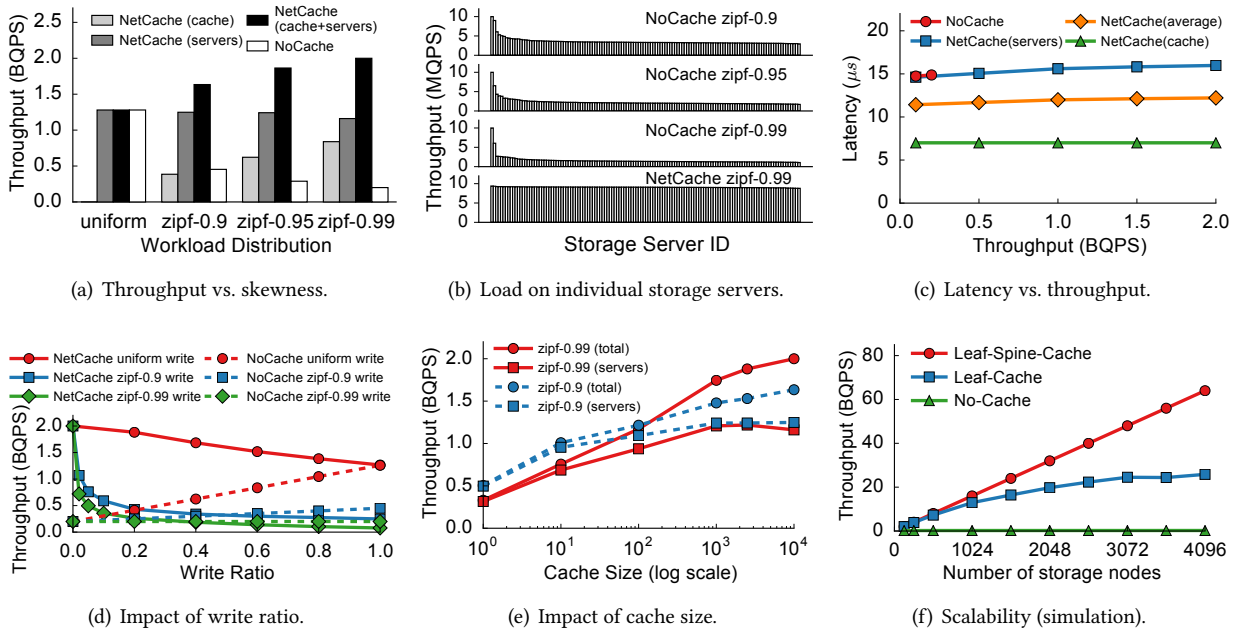
(a) Throughput vs. value size. (b) Throughput vs. cache size.

Figure 9: Switch microbenchmark (read and update).

7.2 Switch Microbenchmark

We first show switch microbenchmark results using snake test (as described in §7.1). We demonstrate that NetCache is able to run on programmable switches at line rate.

Throughput vs. value size. We populate the switch cache with 64K items and vary the value size. Two server machines and one switch are organized to a snake structure. The switch is configured to provide 62 100Gbps ports, and two 40Gbps ports to connect servers. We let the two server machines act as clients and send cache read and update queries to the switch to measure the maximum throughput. As described



(a) Throughput vs. skewness.

(b) Load on individual storage servers.

(c) Latency vs. throughput.

(d) Impact of write ratio.

(e) Impact of cache size.

(f) Scalability (simulation).

Figure 10: System performance.

in §7.1, a query sent by one server machine traverses the switch according to the snake structure, and is received and verified by the other server machine. Figure 9(a) shows the switch provides 2.24 BQPS throughput for value size up to 128 bytes. This is bottlenecked by the maximum sending rate of the servers (35 MQPS). Specifically, we have 2 (two servers) × 35 (per-server throughput) × 32 (each query is replicated 31 times due to the snake structure) MQPS = 2.24 BQPS. The Barefoot Tofino switch is able to achieve more than 4 BQPS. The throughput is not affected by the value size or the read/update ratio. This is because the switch ASIC is designed to process packets with strict timing requirements. As long as our P4 program is compiled to fit the hardware resources, the switch data plane is able to process NetCache queries at line rate.

Our current prototype supports value size up to 128 bytes. Bigger values can be supported by using more stages or using packet mirroring/recirculation for multiple rounds of packing processing as we discussed in §5.

Throughput vs. cache size. We use 128 bytes as the value size and change the cache size. Other settings are the same as the previous experiment. Similarly, Figure 9(b) shows that the throughput keeps at 2.24 BQPS and is not affected by the cache size. Since our current implementation allocates 8 MB memory for the cache, the cache size cannot be larger than 64K for 128-byte values. We note that caching 64K items is sufficient to balance the load for a key-value storage rack as we show in §7.3.

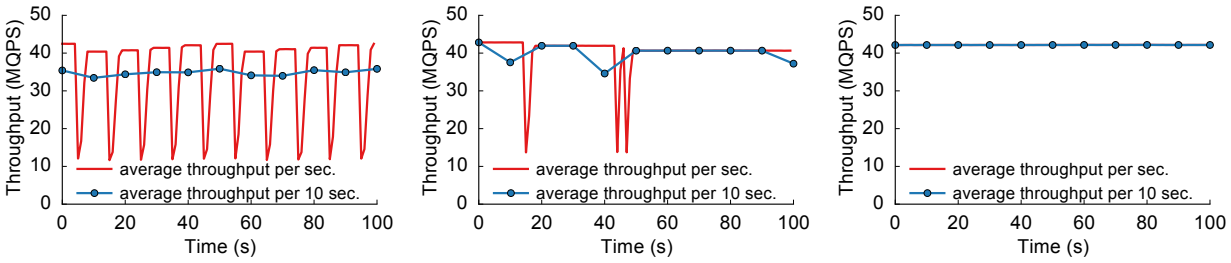
7.3 System Performance

We now present the system performance of a NetCache key-value storage rack that contains one switch and 128 storage servers using server rotation (as described in §7.1).

Throughput. Figure 10(a) shows the system throughput under different skewness parameters with read-only queries and 10,000 items in the cache. We compare NetCache with NoCache which does not have the switch cache. In addition, we also show the the portions of the NetCache throughput provided by the cache and the storage servers respectively. NoCache performs poorly when the workload is skewed. Specifically, with Zipf 0.95 (0.99) distribution, the NoCache throughput drops down to only 22.5% (15.6%), compared to the throughput under the uniform workload. By introducing only a small cache, NetCache effectively reduces the load imbalances and thus improves the throughput. Overall, NetCache improves the throughput by 3.6×, 6.5×, and 10× over NoCache, under Zipf 0.9, 0.95 and 0.99, respectively.

To zoom in on the results, Figure 10(b) shows the throughput breakdown on the individual storage servers when caching is disabled (top three) and enabled (bottom). NetCache uses the switch cache to absorb queries on the hottest items and effectively balances the load on the storage servers.

Latency. Figure 10(c) shows the average query response latency as a function of system throughput. NoCache process all queries with storage servers and the average latency is 15 μs. Its throughput saturates at 0.2 BQPS, after which the



(a) Hot-in workload (radical change). (b) Random workload (moderate change). (c) Hot-out workload (small change).

Figure 11: Handling dynamic workloads. The throughput is for a single client.

system would have queries infinitely queued up at bottleneck servers. NetCache processes queries on cached keys with negligible latency overhead. The 7 μs query latency is mostly caused by the client. Overall, by accounting for queries processed by the storage servers, the average latency of NetCache is 11–12 μs. The latency is steady when the system throughput grows to 2 BQPS.

Impact of write ratio. NetCache targets at read-intensive workloads, and certain types of write-intensive workloads can offset the benefits of in-network caching. This is because write queries invalidate cached items along their routes to storage servers. As a result, write queries not only do not benefit from caching, but also cause cache misses for read queries on the invalidated items. In the adversarial setting where write queries visit the *same* skewed set of hot keys as read queries, the effect of caching would disappear and the system performance would degrade to the baseline.

Figure 10(d) plots how the system throughput changes with different write ratios. The read queries follow Zipf 0.99 distribution. With uniform write queries, load across the storage servers is balanced, so increasing the write ratio reduces the overall throughput of NetCache linearly. In comparison, the throughput of NoCache is low under small write ratios because of skewed reads, and the throughput increases with bigger write ratios because the writes are uniform.

On the other hand, with the same skewed key distributions for both read and write queries, even with a write ratio of 0.2, most read queries on hot items have to reach the storage servers to retrieve the latest values. Because of the severe load imbalance, the overall system throughput of NetCache drops significantly. Thus, the throughput of NetCache is similar to or even slightly worse than that of NoCache with write ratio greater than 0.2 under *highly skewed* write workloads, because NetCache has to pay extra system overhead for maintaining cache coherence. When the write ratio is smaller than 0.2, NetCache has better throughput than NoCache, and the gap is bigger with smaller write ratios and less skewed writes. Therefore, NetCache is most effective for read-intensive workloads with highly-skewed reads, and uniform or moderate-skewed writes. For write-heavy workloads with highly-skewed writes, the switch cache should be disabled to avoid the extra overhead for maintaining cache coherence. This is a generic principle applicable to many caching systems.

Impact of cache size. Figure 10(e) shows how the number of cached items affect the system throughput for Zipf 0.9 and 0.99 workloads. With a cache size of only 1,000 items, the 128 storage nodes are well balanced and achieve the same throughput as with a uniform workload (see Figure 10(a)). The total system throughput (switch cache and storage servers) continues to grow as the number of cached items increases. Because of the skewness of the workload, additional throughput improvement on larger cache sizes is diminishing (note the log scale on x-axis). Furthermore, when the cache size is small, because Zipf 0.9 is less skewed than Zipf 0.99, the system achieves higher throughput under Zipf 0.9. On the other hand, when the cache size is large, the cache provides more throughput under Zipf 0.99, and thus the total throughput is higher under Zipf 0.99.

Scalability. This experiment evaluates how NetCache can potentially scale to multiple racks, as briefly discussed in §5. We use simulations with read-only workloads and assume the switches can absorb queries to hot items. We leave cache coherence and cache allocation for multiple racks as future work. Figure 10(f) shows the simulated results when scaling out the storage system to 4096 storage servers on 32 racks. In fact, the load imbalance problem is more severe with more servers, since the entire system is bottlenecked by the most loaded node. As a result, the overall system throughput of No-Cache stays very low and is not growing even when more servers are added. NetCache running only on ToR switches (Leaf-Cache) gives limited throughput improvements when scaling out to tens of racks: the workload inside a rack is balanced, but the load imbalance between racks still exists. Caching items in spine switches (Leaf-Spine-Cache) balances the load between racks, and so the overall throughput grows linearly as the number of servers increases.

7.4 Handling Dynamics

Finally, we evaluate how NetCache handles dynamic workloads using server emulation (as described in §7.1). Our experiments use the Zipf 0.99 workload with 10,000 items in the cache. Each experiment begins with a pre-populated cache containing the top 10,000 hottest items. The controller runs as a process in the switch OS, and refreshes the query statistics module every second via the switch driver. We evaluate how NetCache reacts to three types of dynamic workloads in terms of system throughput. We use the client to dynamically adjust its sending rate to estimate the real-time saturated system throughput. Specifically, if the client detects packet loss is above a high threshold (e.g., 5%), it decreases its rates; if the packet loss is less than a low threshold (e.g., 1%), client increases its rates. This method cannot *accurately* measure the real-time effective system throughput since the client may under-react or over-react.

Hot-in. We move 200 cold keys to the top of the popularity ranks every 10 seconds. Figure 11(a) shows how the average throughput per second and per 10 seconds change over time. With the in-network heavy hitter detector, the cache is frequently updated to include new hot keys. As a result, the per-second throughput recovers very quickly after a sudden workload change. We hypothesize that these radical changes are unlikely to happen frequently in practice. Nevertheless, this experiment demonstrates that NetCache is robust enough to react to dynamic workloads even with certain adversarial changes in key popularity.

Random. We randomly replace 200 keys in the 10,000 most popular keys every second. The highest ranked popular keys, in this case, are less likely to be replaced, so the deep drops in throughput are less frequent, as shown in Figure 11(b). If we look at the average throughput per 10 seconds, the performance looks almost unaffected by the workload changes.

Hot-out. We let 200 hottest keys suddenly go cold every second, and increase the popularity ranks of all other keys accordingly. Since it's only a change in relative ordering for most cached keys, the system throughput is almost not affected. Figure 11(c) shows that NetCache can easily handle hot-out workloads with very steady throughput over time.

8 RELATED WORK

In-memory key-value stores. Given the high-throughput and low-latency requirements of large-scale Internet services, key-value storage systems are shifting to in-memory designs [1, 15, 16, 22, 23, 26, 27, 29, 30, 34, 36, 38, 42, 43]. They use a variety of techniques to improve the system performance, from using new data structures and algorithms, to exploiting various system-level optimizations and new hardware capabilities.

Load balancing. When scaling out key-value stores, the overall performance is often bottlenecked by the overloaded servers, due to highly-skewed workloads [2, 11]. Traditional methods use consistent hashing [24] and virtual nodes [13] to mitigate load imbalance, but these solutions fall short when dealing with workload changes. "Power of two choices" [33] and data migration strategies [10, 25, 40] are designed to balance dynamic workloads, but introduce additional system overheads for replication and migration, and have limited ability to handle large skew. SwitchKV [28] uses an in-memory caching layer to balance the flash-based storage layer, but is inadequate when the storage layer is also in memory. EC-Cache [37] uses online erasure coding to balance in-memory key-value stores. It splits an item to multiple chunks and thus is not suitable for small items. As a result, it focuses on the workloads for date-intensive clusters, whose item sizes are much larger than typical key-value stores that support web services as targeted by NetCache.

Hardware acceleration. Recent work on network hardware presents new opportunities to improve datacenter key-value stores. IncBricks [31] designs middleboxes to cache items inside the network. IncBricks primarily focuses on improving the cache hit ratio for performance speedup, and thus requires larger storage space. Compared to NetCache, IncBricks requires deployments of specialized hardware (i.e., NPUs) to collocate with existing programmable switches, in order to function as an in-network caching layer. NPUs are not fast enough to satisfy the throughput requirement for a load-balancing cache. In contrast, NetCache provides a load balancing cache which requires little storage space, and is able to efficiently handle workload changes.

9 CONCLUSION

We present NetCache, a new rack-scale key-value store design that guarantees billions of QPS with bounded latencies even under highly-skewed and rapidly-changing workloads. NetCache leverages new-generation programmable switches to build an on-path caching layer to effectively balance the load for the storage layer and guarantees cache coherence with minimal overhead. We believe that NetCache is only one example of ultra-high performance distributed systems enabled by high-speed programmable switches.

Acknowledgments We thank our shepherd John Ousterhout and the anonymous reviewers for their valuable feedback. Xin Jin and Ion Stoica are supported in part by DHS Award HSHQDC-16-3-00083, NSF CISE Expeditions Award CCF-1139158, and gifts from Ant Financial, Amazon Web Services, CapitalOne, Ericsson, GE, Google, Huawei, Intel, IBM, Microsoft and VMware. Robert Soulé is supported in part by Swiss National Science Foundation Award 166132.

REFERENCES

[1] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. 2009. FAWN: A Fast Array of Wimpy Nodes. In *ACM SOSP*.

[2] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload Analysis of a Large-scale Key-value Store. In *ACM SIGMETRICS*.

[3] Barefoot. 2017. Barefoot Capilano. (2017). https://www.barefootnetworks.com/technology/#capilano.

[4] Barefoot. 2017. Barefoot Tofino. (2017). https://www.barefootnetworks.com/technology/#tofino.

[5] M. Berezecki, E. Frachtenberg, M. Paleczny, and K. Steele. 2011. Many-core Key-value Store. In *IEEE IGCC*.

[6] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-independent Packet Processors. *ACM SIGCOMM CCR* (July 2014).

[7] Broadcom. 2017. Broadcom Tomahawk II. (2017). https://www.broadcom.com/.

[8] Andrei Broder and Michael Mitzenmacher. 2004. Network applications of Bloom filters: A survey. *Internet mathematics* (2004).

[9] Cavium. 2017. Cavium XPliant. (2017). https://www.cavium.com/.

[10] Yue Cheng, Aayush Gupta, and Ali R. Butt. 2015. An In-memory Object Caching Framework with Adaptive Load Balancing. In *EuroSys*.

[11] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *ACM SOCC*.

[12] Graham Cormode and S. Muthukrishnan. 2005. An improved data stream summary: The Count-Min sketch and its applications. *Journal of Algorithms* (April 2005).

[13] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. 2001. Wide-area Cooperative Storage with CFS. In *ACM SOSP*.

[14] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *ACM CACM* (February 2013).

[15] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *USENIX NSDI*.

[16] Bin Fan, David G. Andersen, and Michael Kaminsky. 2013. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *USENIX NSDI*.

[17] Bin Fan, Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. 2011. Small Cache, Big Effect: Provable Load Balancing for Randomly Partitioned Cluster Services. In *ACM SOCC*.

[18] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J. Weinberger. 1994. Quickly Generating Billion-record Synthetic Databases. In *ACM SIGMOD*.

[19] Qi Huang, Helga Gudmundsdottir, Ymir Vigfusson, Daniel A. Freedman, Ken Birman, and Robbert van Renesse. 2014. Characterizing Load Imbalance in Real-World Networked Caches. In *ACM SIGCOMM HotNets Workshop*.

[20] Intel. 2017. Intel Data Plane Development Kit (DPDK). (2017). http://dpdk.org/.

[21] Jaeyeon Jung, Balachander Krishnamurthy, and Michael Rabinovich. 2002. Flash Crowds and Denial of Service Attacks: Characterization and Implications for CDNs and Web Sites. In *WWW*.

[22] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2014. Using RDMA efficiently for key-value services. In *ACM SIGCOMM*.

[23] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. Design Guidelines for High Performance RDMA Systems. In *USENIX ATC*.

[24] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *ACM STOC*.

[25] Markus Klems, Adam Silberstein, Jianjun Chen, Masood Mortazavi, Sahaya Andrews Albert, P.P.S. Narayan, Adwait Tumbde, and Brian Cooper. 2012. The Yahoo!: Cloud Datastore Load Balancer. In *CloudDB*.

[26] Sheng Li, Hyeontaek Lim, Victor W Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, David G Andersen, O Seongil, Sukhan Lee, and Pradeep Dubey. 2015. Architecting to achieve a billion requests per second throughput on a single key-value store server platform. In *ISCA*.

[27] Xiaozhou Li, David G Andersen, Michael Kaminsky, and Michael J Freedman. 2014. Algorithmic improvements for fast concurrent cuckoo hashing. In *EuroSys*.

[28] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G. Andersen, and Michael J. Freedman. 2016. Be Fast, Cheap and in Control with SwitchKV. In *USENIX NSDI*.

[29] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. 2011. SILT: A Memory-efficient, High-performance Key-value Store. In *ACM SOSP*.

[30] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: A Holistic Approach to Fast In-memory Key-value Storage. In *USENIX NSDI*.

[31] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. 2017. IncBricks: Toward In-Network Computation with an In-Network Cache. In *ACM ASPLOS*.

[32] Memcached. 2017. Memcached key-value store. (2017). https://memcached.org/.

[33] Michael Mitzenmacher. 2001. The Power of Two Choices in Randomized Load Balancing. *IEEE Transactions on Parallel and Distributed Systems* (October 2001).

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

[35] NoviFlow. 2017. NoviFlow NoviSwitch. (2017). http://noviflow.com/products/noviswitch/.

[36] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. 2015. The RAMCloud Storage System. *ACM Transactions on Computer Systems* (August 2015).

[37] K. V. Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. 2016. EC-Cache: Load-Balanced, Low-Latency Cluster Caching with Online Erasure Coding. In *USENIX OSDI*.

[38] Redis. 2017. Redis data structure store. (2017). https://redis.io/.

[39] Redis. 2017. Using Redis as an LRU cache. (2017). https://redis.io/topics/lru-cache.

[40] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J. Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. 2014. E-Store: Fine-grained Elastic Partitioning for Distributed Transaction Processing Systems. *VLDB* (November 2014).

[41] TommyDS. 2017. TommyDS C library. (2017). http://www.tommyds.it/.

[42] Vijay Vasudevan, Michael Kaminsky, and David G. Andersen. 2012. Using Vector Interfaces to Deliver Millions of IOPS from a Networked Key-value Storage Server. In *ACM SOCC*.

[43] Venkateshwaran Venkataramani, Zach Amsden, Nathan Bronson, George Cabrera III, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Jeremy Hoon, Sachin Kulkarni, Nathan Lawrence, Mark Marchukov, Dmitri Petrov, and Lovro Puzar. 2012. TAO: How Facebook Serves the Social Graph. In *ACM SIGMOD*.