Pushing the Limits of In-Network Caching for Key-Value Stores

Gyuyeong Kim
Sungshin Women's University

Abstract

We present OrbitCache, a new in-network caching architecture that can cache variable-length items to balance a wide range of key-value workloads. Unlike existing works, OrbitCache does not cache hot items in the switch memory. Instead, we make hot items revisit the switch data plane continuously by exploiting packet recirculation. Our approach keeps cached key-value pairs in the switch data plane while freeing them from item size limitations caused by hardware constraints. We implement an OrbitCache prototype on an Intel Tofino switch. Our experimental results show that OrbitCache can balance highly skewed workloads and is robust to various system conditions.

1 Introduction

Key-value stores are the fundamental building blocks for online services owing to fast access to variable-length data [2, 5, 16, 30]. Workloads are generally read-intensive, and most items are hundreds of bytes [7, 8, 12]. A challenge in scaling key-value stores is mitigating load imbalance among storage servers caused by skewed key popularity (e.g., trending events). The load imbalance leads to overloading hot item servers, causing overall performance degradation [21, 27, 29].

In-network caching is a promising load-balancing approach that builds a cache in the switch by leveraging programmable switch ASICs like Intel Tofino [3]. The idea is to store hot items in the on-chip switch memory. Requests read the item value by referring to the cache lookup table where the item key is the table index. Compared to server-based caching, in-network caching provides much higher performance without extra cache nodes or accelerators thanks to a Tbps-scale processing throughput.

Unfortunately, existing solutions [21, 32, 34] can cache only tiny items due to hardware constraints. Specifically, the key size is limited to 16 bytes due to the maximum match-key width of the match-action table. A limited number of match-action stages and a small accessible byte size per stage make it difficult for the value size to exceed 128 bytes. These numbers are far from typical workloads where key and value sizes are still small enough to be stored in a single packet, but larger than the limit [7, 8, 12, 33, 37]. For example, in 54 Twitter workloads [37], most keys are tens of bytes, and many values are less than 1024 bytes. However, the existing solutions cannot cache even a single item in 42 out of 54 workloads because the keys and values are generally larger than the limits. Between the other 12 workloads, only 2 workloads have a portion of cacheable items larger than 50%. This trend is similar in Facebook workloads as well [12]. In this context, we ask the following question: *how can we cache variable-length items in the programmable switch for balancing typical key-value workloads?*

We present OrbitCache, a new in-network caching architecture that can cache variable-length key-value items in the switch data plane. Our idea is to make hot items visit the switch data plane continuously in the form of cache packets instead of caching the hot items in the switch memory. We efficiently use packet recirculation, a built-in feature of the programmable switch ASIC. This enables a packet to revisit the switch data plane through an internal loopback port without going outside. The switch can cache variable-length key-value pairs without being limited by the hardware constraints owing to circulating cache packets. For cache serving, the switch maintains small request metadata in the switch memory and forwards the cache packet to the client if there is a pending request for the key. To support variable-length keys, we use key hashes for cache lookups and resolve hash collisions at the client by comparing the requested key and the returned key, which is contained in the cache packet.

The key insight behind the idea is that it is hard to overcome the memory access constraint of switch hardware if we stick to the approach of caching items in the switch memory. For example, the idea of using key hashes for variable-length keys is difficult to realize in the existing solutions. This is because the switch does not have enough hardware resources to store keys and values together. In OrbitCache, cache packets contain both keys and values, allowing to handle hash colli-

sions at the client. Meanwhile, we may recirculate requests multiple times to read larger values. However, this limits scalability because the number of in-flight packets in the recirculation port increases proportionately to the number of requests, making a bottleneck. In addition, the switch has one internal recirculation port. OrbitCache avoids a bottleneck in the recirculation port. This is because 1) requests are never recirculated; 2) the number of in-flight cache packets is small and constant; 3) the switch can process multiple cache packets simultaneously with hardware-level parallelism, thus minimizing queueing between cache packets.

Realizing the idea imposes various technical challenges. First, the switch should buffer multiple request metadata for cached items. We design a request table as a circular queue structure using multiple register arrays. Our request table provides isolated data access among different keys. Second, a cache packet should serve multiple requests once fetched, rather than a single request. To achieve this, the switch clones the cache packet with low overhead before forwarding it to the client using the packet replication engine (PRE), a special module of the switch ASIC. Next, we should ensure cache coherence. We design an invalidation-based coherence protocol that can fetch new values and send back a reply to the client simultaneously. Lastly, the switch should deal with dynamic workloads where key popularity changes over time. We design an efficient cache update mechanism in the switch control plane.

We have implemented a OrbitCache prototype on an Intel Tofino switch. To evaluate OrbitCache, we build a testbed consisting of commodity servers. Our experimental results show that OrbitCache provides load balancing for many skewed workloads having diverse item sizes. In addition, OrbitCache is robust to various workload conditions like key access distributions, write ratios, and the number of servers. We also show that OrbitCache can adapt to dynamic workloads where key popularity changes over time.

In summary, this work makes the following contributions.

- We design OrbitCache, an in-network caching architecture where the switch can cache variable-length hot items to balance a wide range of workloads for distributed key-value stores.
- We propose several techniques to address technical challenges when realizing the idea of variable-length in-network caching.
- We implement a OrbitCache prototype on Intel Tofino switches and show the efficiency and robustness of OrbitCache through extensive testbed experiments.

2 Background and Motivation

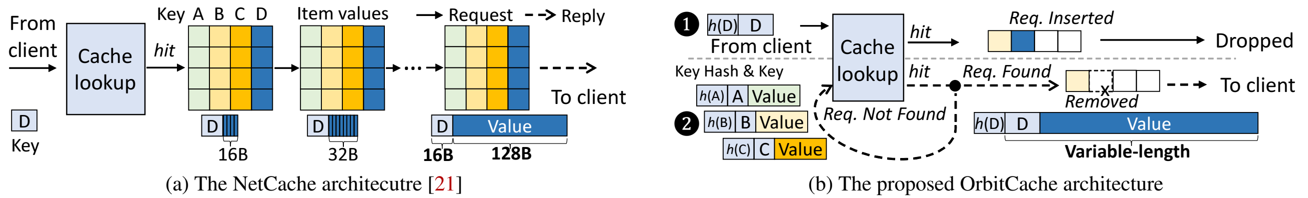
2.1 In-Network Load-Balancing Caches

Balancing key-value stores with a small cache. In distributed key-value stores, balancing imbalanced loads across storage servers caused by different key popularity is a primary challenge. Caching is a powerful technique to address the challenge, which is based on a theoretical result called the *small cache effect*: we can balance loads for N servers (or partitions) by caching the $O(N \log N)$ hottest items, regardless of the number of items [15]. Caching on commodity servers is a natural option to build the cache [15, 16, 29]. However, the performance of a cache node is not sufficient to handle many requests due to the limited throughput of CPUs. Building a high-performance caching layer using multiple replicated cache nodes is expensive and causes poor write performance for cache coherence.

Why in-network caching? Building a load-balancing cache in the switch by leveraging programmable switch ASICs like Intel Tofino [3] is an attractive approach. Unlike server-based caching, this approach can provide high-performance load balancing in a single box without additional nodes. This is because the switch is highly optimized for packet I/O and can process a few billion packets per second, which is an order of magnitude higher than that of the CPU in servers. The switch also provides a low packet processing delay within hundreds of nanoseconds.

Limited cacheable item size. Existing works [21, 32, 34] demonstrate the efficiency of in-network caching, but they limit the cacheable item size. Specifically, they support items of up to 16-byte keys and 128-byte values. In the reconfigurable match table (RMT) switch architecture [10] like Intel Tofino, the switch data plane consists of n match-action stages [3]. Each stage has a static memory and a few ALUs that can perform simple arithmetic operations on k bytes. The existing works store the value of cached items across multiple stages after fragmentation, limiting the maximum value size to $n \times k$ bytes. Unfortunately, $n \times k$ of the current switch is quite small, and the available number of stages for accessing the value is less than n since other non-caching functions also consume match-action stages. The key size is also limited by the maximum match-key width of the match-action table in a single match-action stage. This is because the existing works use a match-action table to implement the cache lookup table where the item key is the match key.

Why is it not enough? Many key-value items are indeed small, but typically exceed the existing size limits. We have analyzed 54 Twitter workloads [37], and observe that existing solutions are insufficient to handle typical workloads. For example, only 3.7% of the workloads have over 80% of keys $\leq$ 16 B. 38.9% of the workloads have over 80% of values $\leq$ 128 B. The existing works can cache less than 10% of items for 85% of the workloads because, to be cacheable, both key and value sizes must be within the limits. Furthermore, they cannot cache even a single item for 77.8% of the workloads. Facebook workloads [12] show similar size distributions. The average key size is 27.1 bytes, and the median value size is 235 bytes. These numbers exceed the size limits of the existing solutions.



(a) The NetCache architecutre [21]

(b) The proposed OrbitCache architecture

Figure 1: Comparison of the high-level idea with the NetCache architecture. *In NetCache, requests read fragmented values stored in the memory across stages. In OrbitCache, clients submit requests, and then circulating cache packets read request metadata. Since both keys and values are in cache packets, hardware constraints do not limit the item size.*

2.2 Variable-Length In-Network Caching

Design rationale. We argue that it is hard to overcome the memory access constraint if we stick to the approach of caching data in the switch memory because the constraint is determined at the time of manufacture. Therefore, we propose OrbitCache, a different approach to in-network caching.

Our high-level idea is to keep cached key-value pairs circulating within the switch data plane through packet recirculation, a built-in feature of the switch that makes the packet revisit the switch data plane. In our approach, cached items read requests rather than the requests read the cached items. Specifically, clients submit requests, and the switch maintains small metadata like the client IP address. *Cache packets*, which are reply packets containing the key and the value of cached items, continue to revisit the data plane via a recirculation port while actively checking pending requests. Note that cache packets do not go outside the switch. If a request for the key is found, the matched cache packet is forwarded to the client. Otherwise, the cache packet is recirculated. In Figure 1, we illustrate the difference between OrbitCache and the NetCache architecture [21], which is the reference architecture of existing in-network caching solutions [21, 32, 34].

Variable-length keys. Our approach enables us to support variable-length key-value pairs. For variable-length keys, one possible solution within the limited match-key width of the match-action table is to use a fixed-size key hash as the match key. To handle potential hash collisions, we should compare the requested key and the key of the value that has been read. To do this, the switch data plane should also store the item keys alongside values. Unfortunately, it is hard to realize in the existing architecture because there are not enough match-action stages to store both keys and values. OrbitCache can realize the idea of using key hashes since the switch maintains cached key-value pairs in the form of circulating reply packets. Therefore, the client can get the correct value from the storage server if the requested and returned keys differ.

Variable-length values. For variable-length values, a possible approach in the existing architecture is recirculating requests. Specifically, requests can access the switch memory multiple times using packet recirculation. However, this approach is not scalable because the number of in-flight packets per second in the recirculation port increases in proportion to the number of requests. For example, if every request is recirculated 7 times to read a 1024-byte value, the effective throughput of the recirculation port is reduced to 1/8 of the bandwidth. Unfortunately, a pipeline in the programmable switch has only one internal recirculation port, while there are tens of regular front ports. This means that the recirculation port limits the performance excessively.

Our recirculation-based caching design avoids the bottleneck in the recirculation port as follows. First, the switch never recirculates requests. Second, only a small, constant number of cache packets are recirculated. The time to recirculate and process a cache packet is a few hundred nanoseconds like normal packets. Therefore, the queueing delay in the recirculation port rarely increases as the request rate grows.

Trade-off. There is a trade-off of using recirculation to achieve variable-length caching. Although we are free from size limitations, we should limit the cache size. This is because, in OrbitCache, requests should wait until cache packets handle them. For a cache packet, the time to read a request is impacted by the other in-flight cache packets. This means that the number of in-flight cache packets in the recirculation port determines the latency for cache serving. Although the switch can process many cache packets simultaneously, the request may wait excessively until being served if there are too many cache packets. The sacrifice of cache size for variable-length caching is backed by the small cache effect. As described in Section 2.1, caching a small number of hot items is enough to balance skewed workloads [15, 27].

Technical challenges. We should address several technical challenges to translate the idea into a working system.

- The switch should maintain multiple requests, especially for the same cached item. If not, many requests for cached items would be forwarded to the server as they cannot be stored in the data plane.
- A cache packet should serve multiple requests once fetched. Otherwise, the switch must fetch the cache packet from the server again, degrading performance.
- We should design a cache coherence mechanism between the switch and storage servers. If not, a request may read a stale value for the requested key.

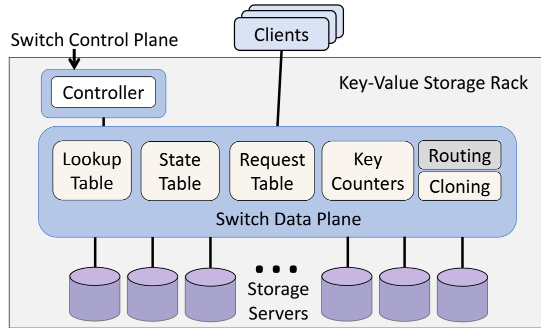


Figure 2: OrbitCache architecture.

- The switch should adapt to key popularity changes.

3 OrbitCache Design

3.1 OrbitCache Architecture

Figure 2 shows the overview of the OrbitCache architecture.

Switch data plane. The switch data plane in OrbitCache consists of several custom tables and modules as follows.

- *Lookup table* is for cache lookups, which is a match-action table that uses a hash of item keys as the match key. The table returns a table index needed to access other tables and modules. Cache entries are managed by the controller in the switch control plane.
- *State table*, implemented as a register array, maintains the validity of the value of cached items. This is needed to prevent read requests from obtaining the stale value when there are pending writes for the requested key.
- *Request table*, implemented with multiple register arrays and registers[1], stores request metadata like client IP address, L4 port number, and request sequence number.
- *Key counters* consist of two registers and one register array. The key popularity counter is a register array that tracks the key popularity for each key. The cache hit counter and the overflow request counter are registers that track the total number of cache hits and the total number of overflow requests for all cached keys. The controller uses these for cache sizing.
- *Cloning module* comprises a few match-action tables for cloning reply packets. Packet cloning is done by the PRE, a hardware module in the switch ASIC [3].

Meanwhile, the switch invokes the custom processing logic only for OrbitCache packets by referring to reserved L4 ports. We use UDP to handle messages for better latency like existing works [21,27,29] while using TCP for top-k item reports in cache updates. For normal packets, the switch only applies the traditional packet forwarding logic.

[1] In P4 language, a register means an indexed register array. However, in this paper, we define a register as a single-slot register and a register array as an indexed register array for better clarification.

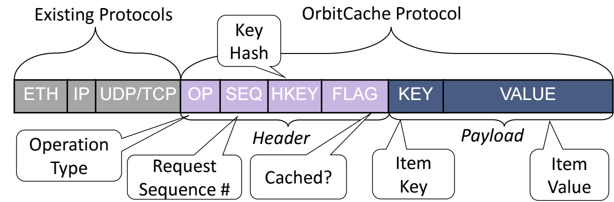


Figure 3: OrbitCache packet format.

Switch control plane. The controller in the switch control plane performs cache updates and switch configurations. It evicts the least popular keys and inserts new hot keys based on server-side periodic top-k reports for uncached keys and the switch-side report for cached keys. Note that while the controller handles key insertion, value fetching is done via the switch data plane. It also handles switch configurations like the rule update for packet forwarding tables.

Clients and servers. Clients should specify the operation type, the item key to request, and the key hash. Storage servers run a server application that acts as a shim layer that translates OrbitCache messages to API calls for key-value stores and vice versa. The server returns replies for read and write requests like regular storage servers. However, in cases of write requests for cached items, the server returns a write reply with the item value as well to fetch the latest value.

3.2 Packet Format

Figure 3 depicts the OrbitCache packet format. The message consists of the header and the payload. The switch only parses the header. The payload of OrbitCache message consists of the key and the value. Our header is 22 bytes. Therefore, OrbitCache supports a key-value pair of up to 1438 bytes for a single packet when we consider a 1500-byte MTU packet where 40 bytes are for L3/L4 headers. For example, the switch can cache an item with a 16-byte key and a 1422-byte value. Our header fields are as follows.

- `OP` (1 byte): the operation type, which can be `R-REQ` (Read request), `W-REQ` (Write request), `R-REP` (Read reply), `W-REP` (Write reply), `F-REQ` (Fetch request), `F-REP` (Fetch reply), and `CRN-REQ` (Correction request).
- `SEQ` (4 bytes): a request ID assigned by the client, which is used for resolving hash collisions.
- `HKEY` (16 bytes): the key hash as cache lookup index.
- `FLAG` (1 byte): a flag field to distinguish write requests for cached items from those for uncached items.

3.3 Basic Request and Reply Processing

In Figure 4, we illustrate the packet processing logic.

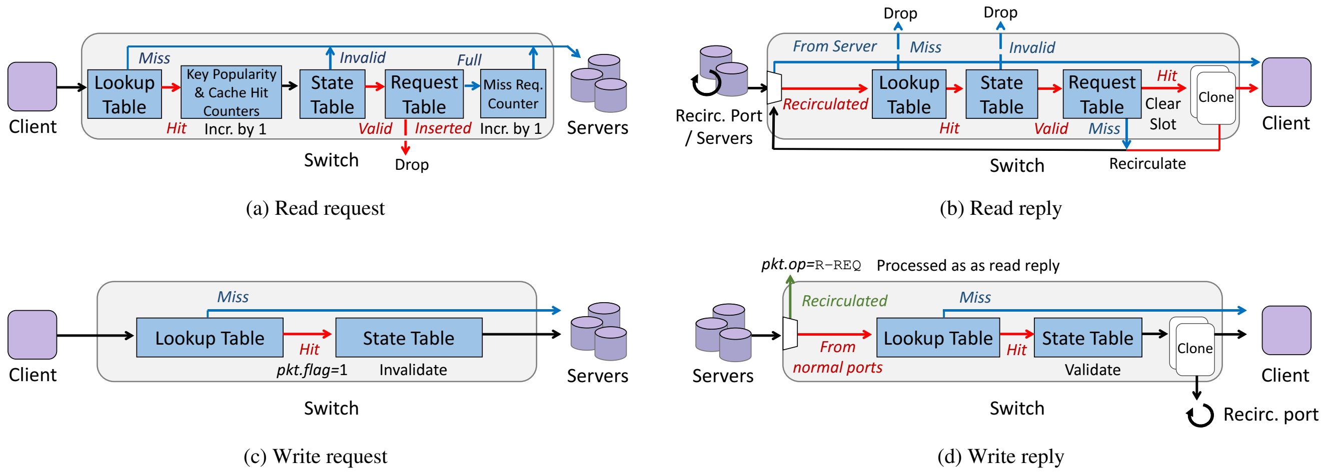


Figure 4: Request processing. *(a) the switch drops the request after inserting request metadata into the request table; (b) If a circulating cache packet reads request metadata, the switch clones the packet so that the original packet is forwarded to the client and the cloned one is recirculated again for further serving; (c) the switch invalidates the item to avoid inconsistent reads if a write request is for a cached item; (d) upon receiving a write reply for a cached item, the switch validates the item. After that, the switch clones the packet. The cloned packet is processed as a read reply after updating the operation type.*

Request generation at clients. Clients should specify the OP and HKEY fields. The SEQ field is increased by one for every request, which is used to resolve hash collisions. The destination storage server is determined by hashing the key.

Read requests. The switch first refers to the lookup table using a key hash to get a table index, which is used to access the slots in other tables. If missed, the switch forwards the packet to the server as the request is for an uncached item. In case of a cache hit, the key popularity counter and the cache hit counter are incremented by one. This key popularity is collected by the controller in the switch control plane periodically for cache updates. Next, the switch checks the validity of the requested key by looking into the state table. The state is binary: valid or invalid. Being invalid means that there are pending write requests for the key. In this case, the switch forwards the read request to the server to avoid reading stale item values. If the key is valid, the switch checks the request table to see whether there is a free slot. The switch puts the request metadata into the table when a free slot is found. Otherwise, the request is destined to the server after the overflow request counter is increased. Request metadata includes the client IP address, L4 port number, and SEQ as request IDs. After insertion, the switch drops the packet. This is acceptable since a cache packet will soon service the stored request.

Read replies. When receiving a read reply, the switch first checks to see if the ingress port is the recirculation port. If it is, the reply is a cache packet. Otherwise, it is a reply for an uncached item sent by the server. For cache packets, if the cache misses or the state of the value is invalid, the switch drops the packet. A cache miss for a cache packet means that the controller evicted the key from the lookup table due to a change in key popularity or the cache size. The invalid state indicates a write request with a new value is in progress.

The switch goes through the lookup and state tables, and then looks for pending requests in the request table. If a request is found, the switch forwards the cache packet to the client after updating the header with metadata and removing the metadata from the table slot. To make the cache packet serve more requests, the switch clones the packet before forwarding. The switch forwards the original packet to the client and the cloned one to the recirculation port. The switch recirculates the packet if there are no pending requests.

Write requests. The switch checks whether the requested key is in the cache lookup table. If it is, the value for the key is invalidated to prevent reading the outdated value. In addition, the switch sets the FLAG field to 1 to indicate that this request is for a cached item. This makes the storage server append the value in the write reply. Regardless of a cache hit, the switch forwards the request to the storage server to update the value of the key in the server.

Write replies. If the key is cached, the switch validates the value to allow read requests to get the latest value. The switch clones the packet so that the client receives the write reply while the switch has a new cache packet simultaneously. The switch updates the OP field to R-REP of the cloned packet after the first recirculation since cache packets should be read replies. For a cache miss, the switch forwards the packet to the client since the reply is for an uncached item.

Other types of messages. Fetch messages are for cache updates. Correction messages are for resolving hash collisions. Fetch/correction requests are delivered to the storage servers. The fetch reply is processed as a write reply.

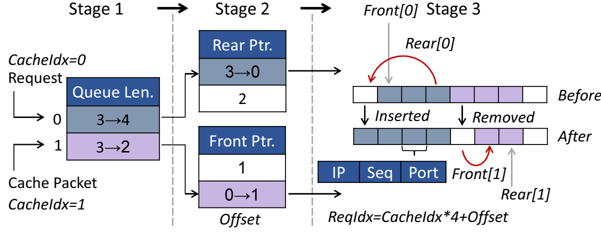


Figure 5: An example of operations in the request table.

3.4 Buffering Request Metadata

Circular queue-based request table. We should store multiple request metadata for cached items because the requests should wait until cache packets serve them. The switch should be able to buffer concurrent read requests for the same cached item. If not, many requests for the cached item would be forwarded to the storage server due to the lack of vacancies. Furthermore, a request for a key should be isolated from requests for other keys since the request may undergo excessive queueing delay due to the different requests.

We design a request table based on a circular queue structure. Our request table provides a logical queue for each cached key, and the queue can be accessed in $O(1)$ by efficient indexing. This provides fast and isolated queue access for a key regardless of other keys. The request table consists of 6 register arrays: the first three arrays are for storing request metadata for each key (i.e., client IP address, request sequence number array, and L4 port number). The other three arrays are for managing queue operations. The queue length array maintains the number of stored request metadata for each key. The front pointer array handles dequeue operations by tracking the first request metadata for each key. The rear pointer arrays perform enqueue operations by keeping track of the last request metadata for each key.

The queue management arrays are indexed using a table index (*CacheIdx*) returned by the cache lookup table. The request metadata arrays are accessed via a request index, which is defined as $ReqIdx = CacheIdx \times S + i$ where S is the maximum queue size per key and i is the offset for ith slot in the logical queue for a key and is given by the pointer arrays. The switch uses three match-action stages for a request table. Specifically, the switch first checks the queue status (Stage 1), and performs en/dequeue operations if the queue is not full/empty (Stage 2). Next, the switch gets/puts metadata from/into the request table (Stage 3).

An example for table operations. Figure 5 shows a simple example of table operations. We assume that the request table can maintain up to 4 requests for each key. We consider enqueueing and dequeueing operations using read requests and cache packets. In stage 2, the rear pointer changes to 0 from 3 since we implement a circular queue. It is also easy to see that the request metadata for different keys does not collide since we partition the metadata arrays using the indexing formula of $ReqIdx = CacheIdx \times S + i$.

3.5 Cache Serving

Handling multiple requests via packet cloning. In OrbitCache, cache packets pass through the switch data plane repeatably to serve pending requests for cached items. One challenge is serving multiple requests with a cache packet fetched only once because the cache packet is forwarded to the client after updating the header using the retrieved metadata. A strawman is to fetch the cache packet from the server again, but this approach is inefficient as the switch cannot serve pending requests for the key until the fetching is completed. Fetching multiple copies of the cache packet may be a solution, but this incurs excessive queueing delay between cache packets. In addition, it is hard to know how many cache packets should be fetched because we cannot predict the number of requests for each cached item in advance.

To address this, we utilize packet cloning, another built-in feature of the programmable switch ASIC besides packet recirculation. Packet cloning is done by the PRE, a hardware module specialized for packet cloning in the switch ASIC. Cloning has low overhead for the following reasons. First, the PRE is located after the ingress pipeline. This means the switch does not repeat the ingress pipeline processing for the cloned packet, not causing extra ingress processing delay. Second, the switch does not copy the entire packet. It only copies the small descriptor pointing to the memory location of the packet and reuses the packet data.

We use multicast to forward the original and clone packets. Through a match-action table using the destination IP address acquired from the request table, the switch gets a multicast group ID that specifies the regular port number directed to the client and the recirculation port number. The switch forwards the original packet to the client and the cloned one to the switch data plane again, making it serve more requests.

3.6 Handling Hash Collisions

The existing works use the item key as the match key of the cache lookup table, which is implemented as a match-action table. Since the match-key width is limited by hardware, we cannot use the key exceeding the size limit. We may use a register array, but the size limitation in the register index is stricter than in the match-action table. Therefore, we use the fixed-sized key hash as the match key. A challenge is how to resolve potential hash collisions. We should handle this because a request may read the wrong value of other keys.

Client-side collision resolution. We handle hash collisions at the client level by maintaining a list of the keys for each request that has not yet received a reply. The list is indexed by *pkt.seq*. Request packets contain both the original key and the key hash since we should use the original key

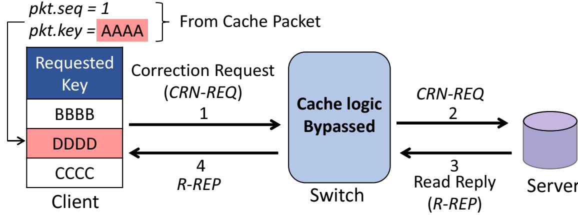


Figure 6: An example of handling hash collisions. *The client gets the correct value by sending a correction request.*

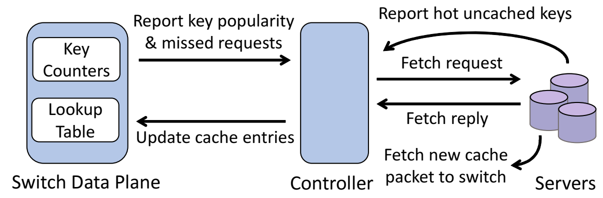


Figure 7: Cache update. *The controller updates the entries of the cache lookup table by tracking workload changes.*

to get the value in servers. Upon receiving read replies, the client checks whether the requested key in the list and the returned key in the reply header are identical. If different, the client sends a correction request to the server. The switch then forwards the request without applying the cache logic so that the client gets the correct item value from the storage server. We use a simple, low-overhead hash function to calculate key hashes. In addition, the complexity of accessing a matched slot of the key list is $O(1)$ as it is an indexed array. Note that a key in the list exists only until the reply arrives.

One issue is that requests for uncached but colliding keys always undergo this process. However, this is rare since our 128-bit key hash provides a low collision probability of $1/2^{128}$. In our experience, we never see a hash collision. There is a corner case when a new hot item uses the table index of the evicted item after cache updates. In this case, the pending requests for the evicted item are served by the new but wrong cache packet. However, they eventually get the correct value through the correction process with 1-RTT latency overhead. Meanwhile, the switch drops the colliding cache packet when hash collisions occur due to write requests. To handle this, the server can send the cache packet as a fetch reply to the switch again if the received write request is with $pkt.flag = 1$ but not in the hot key list.

An example of resolving hash collisions. We plot an example of handling hash collisions in Figure 6 where the requested key is `DDDD` but the returned key is `AAAA`. Upon receiving the reply, the client detects that the retrieved value is wrong by referring to the key list using $pkt.seq = 1$. The client then sends a correction request to the storage server. The switch bypasses the cache logic, and forwards the packet to the server. The storage server returns the value as a read reply, and the client finally gets the correct value for the key `DDDD`. The client removes the key from the list. $pkt.seq$ wraps around if it reaches the maximum value.

3.7 Cache Coherence

To ensure cache coherence between the switch and servers, we design an invalidation-based coherence protocol illustrated in Figure 4 (c) and (d). The switch invalidates the value when handling a write request for a cached item and revalidates the value upon receiving a write reply. Ideally, the outdated cache packets should be replaced with the latest cache packet. However, requests may be returned with the stale value since the outdated packets are still circulating in the switch data plane even if the item value becomes invalid. To address this, the switch drops the cache packet if the item is cached but its value is invalid. Owing to the fact that the cache packet is dropped before accessing the request table, we can prevent read requests from retrieving the stale value until a new cache packet is fetched. The storage server sends a single reply packet, and the switch updates the value and replies to the client simultaneously by cloning the packet.

3.8 Cache Updates

To handle key popularity changes over time, we update cache entries based on periodic popularity reports from the switch and storage servers. We count the popularity of cached keys in the switch data plane using the key popularity counter. The controller keeps track of key popularity by reading the counter periodically. Meanwhile, storage servers periodically report the top-k keys to the controller, which are popular uncached keys. The servers use a count-min sketch with five hash functions to track key popularity in a memory-efficient manner while ensuring accuracy. To reflect the recent status only, we reset all the counters to zero after reporting.

If the cache needs to be updated due to a change in key popularity, the controller deletes the victim key from the cache lookup table and inserts a new popular key instead. The controller then sends a fetch request to the storage server containing the latest value for the key. Upon receiving the request, the server fetches a new cache packet to the switch data plane and replies to the controller. The new popular key inherits the table index (*CacheIdx*) of the evicted key. With this, the pending requests for the evicted key can be handled by the new cache packet and the hash collision resolution mechanism. Note that the hash collision here is due to the same table index, not that the hashes actually collide.

3.9 Handling Practical Requirements

Failure handling. For packet loss, we can use an application-level mechanism. Our controller uses UDP with a timeout-based mechanism to exchange fetch requests/replies. Storage

servers use TCP to report top-k items to the controller. Meanwhile, server failures do not cause a OrbitCache-specific problem. Switch failures result in the loss of cached items, but the cache can be reconstructed quickly by the controller after the switch is recovered because the switch failure is similar to the rapid key popularity changes.

Multi-rack deployment. OrbitCache can be deployed with multiple racks because the ToR switch caches hot items of storage servers belonging to its rack only. For example, assume that we have two racks where ToR switches, $ToR1$ and $ToR2$, are interconnected via a spine switch SPN. When CLI, a client in rack 1, sends a request to SRV, a server in rack 2, the path for an uncached item is $CLI-ToR1-SPN-ToR2-SRV-ToR2-SPN-ToR1-CLI$. If the item is cached, $ToR2$ is the only switch that applies the cache logic. There is no issue even if rack 1 is the replicated rack of rack 2 where the two ToR switches have the same cache entries because the request is handled by $ToR1$. If the client is located in another rack and the other two racks are replicated ones, we can balance requests by implementing load-balancing mechanisms in the ToR switch of the client-side rack.

3.10 Discussion

Multi-packet items. Many key-value items are less than the MTU size. Therefore, existing works in key-value stores generally consider item values up to 1024 bytes [6,17,26,29–31]. However, some items may exceed the MTU size, such as articles and photo objects for some workloads (e.g., Wikipedia and Flickr [8]). To cache multi-packet items, we should fetch multiple cache packets with fragmented values for the same key. To do this, we maintain the number of forwarded cache packets for each item by placing another register array alongside the request table, the *ACKed packet counter*. The initial value of each slot is 1 since most items are single-packet. When fetching item values, the storage server puts the number of packets comprising the item in the `FLAG` field. When handling cache packets, the switch retrieves metadata as usual, but it does not manipulate the slots of all register arrays in the request table if the slot value in the ACKed packet counter is not equal to `FLAG`. Instead, the switch increases the value by one. When the value in the counter equals to `FLAG`, the switch removes metadata as it is ready to be finished.

Write-back caching. FarReach [34] is a recent work that enables write-back caching. Although it still cannot serve many workloads due to the size limitation, it provides high performance regardless of the write ratio. OrbitCache uses write-through caching, and our performance gain decreases as the write ratio grows. The difference between write-through caching and write-back caching is whether a write request for cached items updates the storage server or not. Therefore, OrbitCache can also use write-back caching by letting the switch return write replies upon receiving write requests after updating the cache only, though we need extra modules like snapshot generation.

Multi-pipeline deployment. The multi-terabits performance of the programmable switch ASIC comes from the multi-pipelined architecture. Each pipeline consists of tens of regular ports and one internal recirculation port. Metadata and memory data are not shared between the pipelines. Therefore, the pipelines of client-directed ports and server-directed ports should be the same. Otherwise, cache packets in a pipeline may not read requests since they are in another pipeline. This can be addressed by mapping each pipeline of the ToR switch to a client-directed port on the spine switches. This is especially feasible when the clients and servers are in different racks. We expect this to be addressed more easily with a new programmable switch architecture like MP5 [35] that achieves high performance with a logical single pipeline.

4 Implementation

Client-server application. We develop an open-loop application in C using NVIDIA Messaging Accelerator library (VMA) [4]. VMA bypasses kernel network stacks by intercepting the socket function calls and translating them to native RDMA verbs. The client application measures throughput and latency by generating requests. The time gap between consecutive requests follows an exponential distribution. The server application has multiple threads where each thread is pinned to a disjoint CPU core. To emulate multiple storage servers, we assign a partition per thread so that each thread acts as an independent storage server. We also limit the Rx throughput of each emulated server to 100K RPS to ensure the bottleneck is at servers in our testbed. This technique is used in existing works [21,29,34] as well. In a similar vein, like NetCache [21], we implement a key-value store with TommyDS [1], a high-performance hash table library.

Switch. We implement the switch data plane in $P4_{16}$ [9] for Intel Tofino [3]. We use Intel P4 Studio SDE 9.7.0 to compile the switch data plane. Our prototype uses 9 stages and 6.67% SRAM, 7.38% Match Input Crossbar, 9.29% Hash Bit, and 30.56% ALUs. The request table has a maximum queue size of 8 for each key. The controller is written in Python 3. In our prototype, we use an additional register array for the request table to store the timestamp of the request for latency measurement. The OrbitCache header has 3 extra fields of 1-B `Cached`, 4-B `Latency`, and 1-B `SrvID`. The first two fields are required to separately measure the latency of requests. The last field is to store the server ID as we emulate multiple storage servers as dedicated threads in a physical node.

5 Evaluation

5.1 Methodology

Testbed setup. We build a cluster consisting of 8 nodes, which are connected by an APS BF6064X-T switch with the

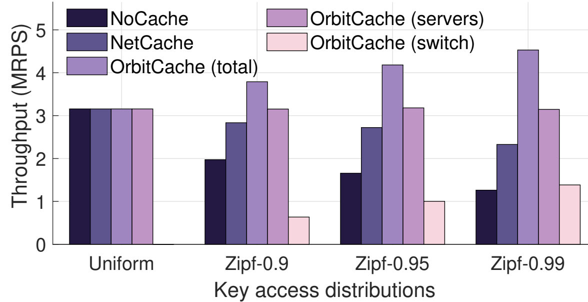


Figure 8: Throughput with different skewness.

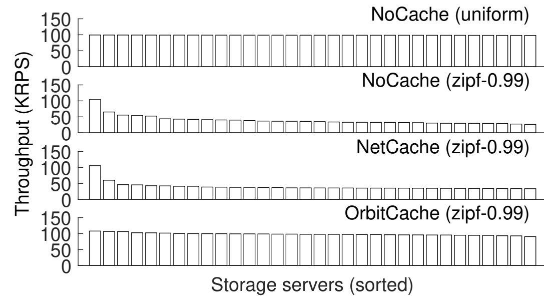


Figure 9: Load on individual storage servers.

Intel Tofino 1 [3]. The servers are equipped with a 10-core CPU (Intel i5-12600K @ 3.7 Ghz, 12 hyperthreads and 4 non-hyperthreads), 32 GB of DDR5 memory, and a 100GbE NVIDIA CX-5 NIC. The servers run Ubuntu 22.04 LTS with Linux kernel 6.5.0. The 4 nodes act as clients and the remaining 4 nodes are used to emulate multiple storage servers.

Compared schemes. We mainly compare our work with NoCache and NetCache [21]. NoCache is a mechanism without cache logic. NetCache is the representative in-network caching architecture. FarReach [34] and DistCache [32] also adopt the NetCache architecture. We implement the core caching logic of NetCache, but our implementation provides items up to 64-byte values across 8 stages with an 8-byte accessible size per stage. We find that the P4 compiler allocates only two cache read tables per stage, even with pragma statements. We suspect that this is due to compiler restrictions on our code. We clarify that this does not mislead the conclusions of our experiments, as there is a negligible difference in the latency required to read 64 bytes and 128 bytes at line rate using the same number of match-action stages.

Workloads. We emulate a single storage rack with 32 storage servers using 8 partitioned threads per node. We basically consider a workload with 10M key-value pairs whose key popularity skewness follows a Zipfian distribution with $\alpha = 0.99$, since it is regarded as typical skewness [7, 13]. Although both key and value sizes impact whether an item is cacheable by NetCache, we use 16-byte keys by default for simplicity, the maximum supported key size by NetCache. Instead, we represent the portion of cacheable items using different ratios between 64 bytes and 1024 bytes values. The 64-byte value represents a cacheable item value of NetCache. The 1024-byte value is a typical-sized item value in many workloads and is considered in many research papers [6, 17, 26, 29–31]. We use a bimodal distribution with 82% 64-byte and 18% 1024-byte values by considering the cacheable item ratio of NetCache for the `Cluster018` workload of Twitter [12]. Most experiments are for read-only workloads as we target read-intensive workloads.

Except for dynamic workloads, we preload the 10K and 128 hottest items for NetCache and OrbitCache, respectively. 128 is a nearly optimal cache size for OrbitCache that provides the best performance gains. Since NetCache can cache only 82% of items in the workload, the actual number of cached items is 8200, which is still larger than the cache size of OrbitCache by 64×. To be fair, we choose 82% of keys among the 10K hottest keys with a uniform distribution. We store the chosen keys as a text file to make experimental results consistent. The controller loads the file and puts the keys to the switch cache. We note that trends are usually consistent, even when using different key samples.

5.2 Main Results

Throughput with different key access distributions. We measure the throughput with different skewness and plot the results in Figure 8. We can see that OrbitCache provides high throughput regardless of skewness, unlike the others whose throughput decreases as the workload becomes more skewed. NetCache does not provide high throughput as expected since many hot items are not cacheable. In the Zipf-0.99 workload, OrbitCache has higher throughput than NoCache and NetCache by 3.59× and 1.95×, respectively. The server throughput in OrbitCache is consistent across the given skewness, and this means that the loads are balanced.

Individual server loads. We plot the load on individual servers in Figure 9. We can see that NoCache and NetCache do not balance loads well. This is due to the lack of caching logic for NoCache and many uncacheable hot items for NetCache. However, OrbitCache can balance the loads since OrbitCache can cache variable-length items.

Latency vs. throughput. We measure the latency by varying Tx throughput. Figure 10 shows the median and the 99th percentile latencies as a function of Rx throughput. OrbitCache provides the best throughput but slightly higher latency than NetCache of 1 microsecond. This is because NetCache has a larger cache size, and requests handled by the switch dominate in latency data. In addition, in OrbitCache, requests for cached items should wait until cache packets read them. Although NetCache has slightly better latency, throughput is very limited since it fails to balance loads.

Impact of write ratio. Figure 11 reports the throughput as a function of write ratios. The throughput of OrbitCache

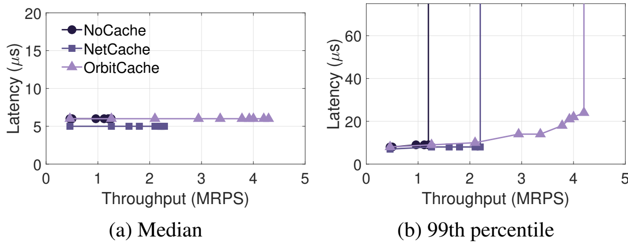


Figure 10: Latency vs. throughput.

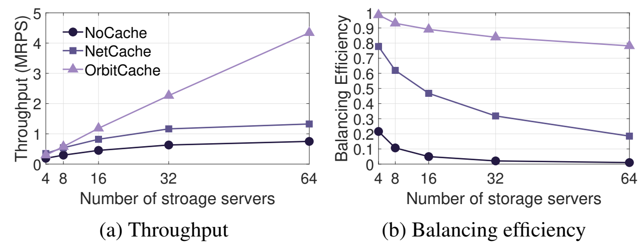


Figure 12: Scalability.

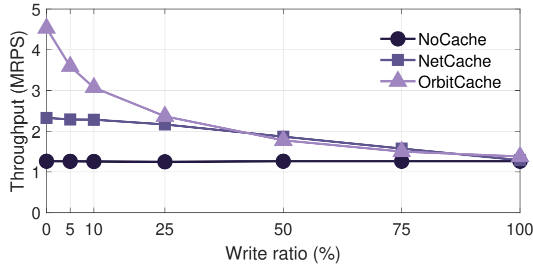


Figure 11: Impact of write ratio.

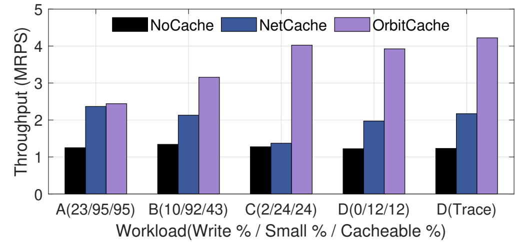


Figure 13: Performance with production workloads.

decreases as the write ratio grows. This is because the switch invalidates the cached key when handling a write request for a cached item to ensure cache coherence. The switch cache provides performance gains for read requests, the throughput is decreased with higher write ratios since read requests for invalid cached items are forwarded to storage servers. With the 100% write ratio, the throughput of OrbitCache converges to NoCache since the cache does not provide any benefit. Meanwhile, similar to OrbitCache, NetCache offers reduced throughput as the write ratio grows since it also invalidates the cached key if there is a write for the key.

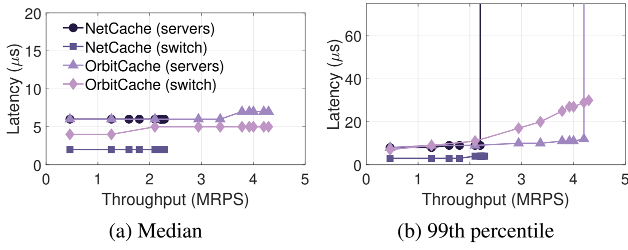
Scalability. We now inspect whether OrbitCache can balance loads of different numbers of servers. In this experiment, we limit the Rx throughput to 50K RPS to ensure that the bottleneck occurs at the storage servers rather than the clients, even when using 64 servers. In Figure 12 (a), we can see that the throughput of OrbitCache is improved almost linearly while the others do not. Figure 12 (b) clarifies that this is because NoCache and NetCache fail to balance imbalanced loads. *Balancing efficiency* is defined as the minimum throughput between the servers divided by the maximum throughput between the servers.

Performance with production workloads. We conduct experiments with several workloads of Twitter [37] to see how OrbitCache works with various production workloads. We pick 5 workloads based on the cacheable item ratio. We assign the workload ID A to D for `Cluster045/016/044/017`. We still use the 16-B keys for simplicity but vary the write ratio, the portion of 64-B values. Unlike the other experiments, the cacheable item ratio is controlled by choosing keys with a uniform distribution independent of the portion of 64-B values. We also use a non-bimodal version of workload D, by referring to the real trace of values in `Cluster/017` to show the fidelity of bimodal distributions in our experiments.

Figure 13 shows the results. Although the difference in throughput varies depending on the used workloads, we can see that OrbitCache shows the best performance for all the workloads. There is a little difference for Workload A. This is because NetCache can cache 95% of items, and the write ratio is relatively high. Workload E is a workload that shows a significant performance gap. This is because only 1% of items are cacheable for NetCache. We can also see that the trend in workloads D and D(Trace) is very similar. The slight throughput difference in workload D(Trace) is because the real trace contains more item values of less than 1024 bytes than the bimodal version. This demonstrates that our bimodal-distributed workloads successfully reflect the characteristics of real-world workloads.

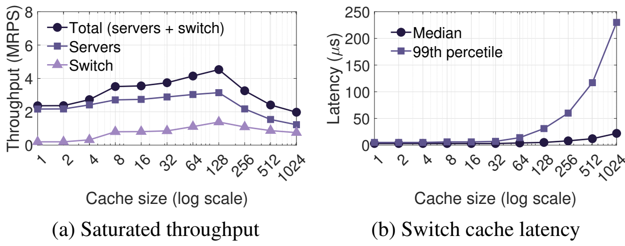
5.3 Deep Dive

Latency breakdown. Figure 14 (a) and (b) plot the median and the 99th percentile latency breakdowns, respectively. In the median latency, OrbitCache shows slightly higher switch latency than NetCache. This is because requests in OrbitCache should wait until circulating cache packets read them, resulting in latency overhead. In Figure 14 (b), we can see that the tail latency of OrbitCache (switch) increases as throughput grows. This is not surprising because we use a circular queue-based request table to maintain request metadata and packet cloning for a cache hit, resulting in additional latency overhead. We believe that this is acceptable since Or-



(a) Median (b) 99th percentile

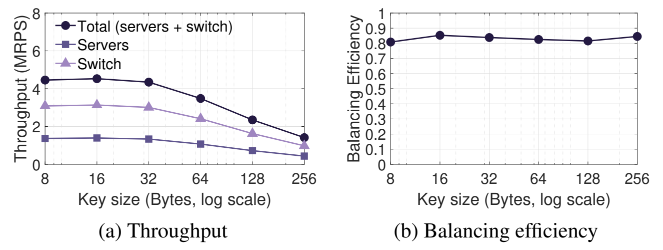
Figure 14: Latency breakdown.



(a) Saturated throughput (b) Switch cache latency

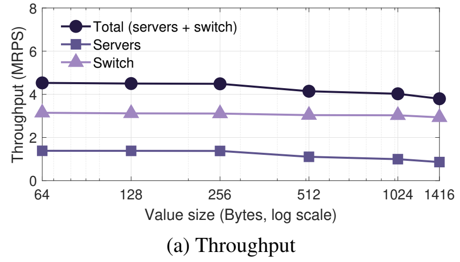
(c) Overflow request ratio

Figure 15: Impact of cache size.

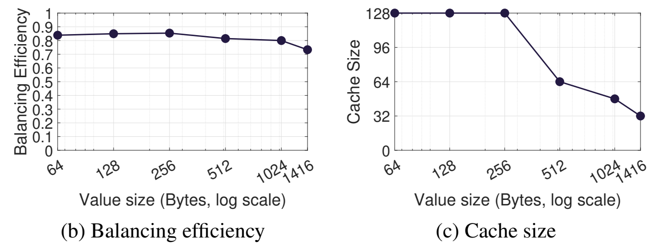


(a) Throughput (b) Balancing efficiency

Figure 16: Impact of key size.



(a) Throughput



(b) Balancing efficiency (c) Cache size

Figure 17: Impact of value size.

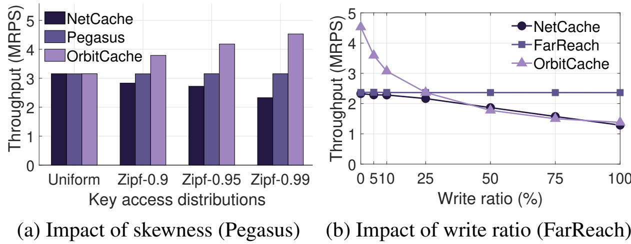
bitCache can provide much higher throughput than the others. In addition, the tail latency of the switch is still tens of microseconds even when the tail latency of servers soars by reaching the saturated throughput. We can decrease the tail latency of the switch by reducing the cache size as well. If the cache is based on a commodity server, the tail latency would be 10-100× longer than the switch cache.

Impact of cache size. Figure 15 plots the throughput breakdown, request latency handled by the switch, and the overflow request ratio of OrbitCache with different cache sizes. We can see that the total throughput increases as the number of cached entries grows. However, the throughput is saturated with around 128 cached items. Similarly, in Figure 15 (b), the tail latency increases quickly after 64-128 cached items. Figure 15 (c) clarifies the reason. From 256 cached items, the overflow request ratio rapidly increases, where the overflow requests indicate the requests for cached items forwarded to the storage servers due to the lack of free slots in the request table. The lack of free slots is caused by the excessive queueing delay between too many cache packets beyond the capability of the switch hardware. This clarifies that the cache size is the key to determining the performance. This also means we should choose an effective cache size between 32 and 128 to balance the trade-off.

Impact of key size. To see the impact of key size, we measure throughput and balancing efficiency using different key sizes. To show the impact of key size more clearly, we use a value of 100% 64-B. Figure 16 (a) shows that throughput decreases as key size increases. This is because the server consumes more computing power when key size is large. Figure 16 (b) shows that balancing efficiency remains high regardless of throughput changes.

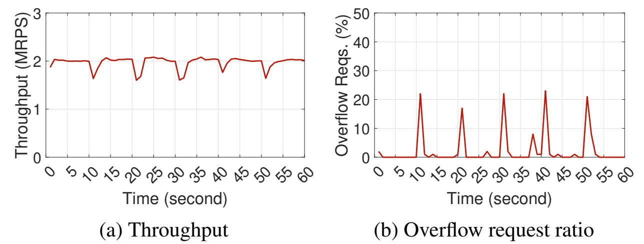
Impact of value size. We measure throughput, balancing efficiency, and the maximum cache size by varying the value size. In this experiment, we use the 100% same value size for all items to inspect how OrbitCache works in the worst case. Note that 16-B key and 1416-B value are the maximum size for a single packet payload with 28-B custom header fields.

Figure 17 (a) shows that OrbitCache can balance various workloads, including a workload where 100% values are MTU-sized. The slight drops in throughput are due to the increased value size. Figure 17 (b) shows the balancing efficiency. We see that OrbitCache generally maintains high balancing efficiency. Figure 17 (c) plots the effective cache size that provides the highest performance gain. We can see that



(a) Impact of skewness (Pegasus) (b) Impact of write ratio (FarReach)

Figure 18: Comparison to Pegasus [27] and FarReach [34].



(a) Throughput (b) Overflow request ratio

Figure 19: Performance with dynamic workloads.

the effective cache size decreases as the value size increases. This is because larger cache packets consume more switch resources, making the effective cache size small. Nevertheless, we can see that OrbitCache can balance the workloads well since we need only a small number of cached items to balance skewed workloads.

Comparison to Pegasus and FarReach. We now compare OrbitCache against Pegasus [27] and FarReach [34]. FarReach enables write-back caching in NetCache. Pegasus [27] is a selective replication solution where the switch replicates hot items across storage servers instead of caching items. We measure throughput using different key access distributions for Pegasus and different write ratios for FarReach. Figure 18 (a) shows the result for Pegasus. We can see that OrbitCache outperforms Pegasus for all the key access distributions. This is because the switch of OrbitCache provides extra throughput, whereas the throughput of Pegasus is limited to the throughput of storage servers. Pegasus is better than NetCache as it provides variable-length items. Figure 18 (b) shows the result for FarReach. OrbitCache outperforms FarReach until 25% of write ratio since FarReach has a limited cacheable item size like NetCache. However, when the write ratio grows, the performance gap with OrbitCache decreases. Over 25% of write ratio, FarReach shows better throughput than OrbitCache. This is because the write latency of FarReach is shorter than OrbitCache as it only updates the value of the item in the switch using write-back caching. OrbitCache updates the value of the switch and the server at the same time.

Handling dynamic workloads. We investigate how OrbitCache reacts to dynamic workloads. Like existing works [21, 29, 34], we use a *hot-in* pattern, which is the most radical workload change. We use 4 storage servers without server emulation and Rx rate limits similar to a previous work [34] to avoid inaccuracy due to the system state change. Every 10 seconds, the popularity of the 128 coldest items and the 128 hottest items is swapped.

Figure 19 (a) shows the throughput for 60 seconds. We can see that the throughput decreases when key popularity changes but recovers within a few seconds. This is because the controller in the switch control plane quickly updates the cache entries and fetches cache packets based on the top-k report of servers and the cache popularity counter of the switch data plane. Figure 19 (b) plots the change in the overflow request ratio over time. We can see that the ratio soars when the key popularity changes. This is because it takes time to fetch items from storage servers.

6 Related work

Storage systems are major target domains of recent in-network computing works (e.g., [11, 14, 18–25, 27, 28, 36, 38–40]). NetCache [21] shows the potential and limitations of switch-based caching. The limitation in the item size is passed down to recent works like FarReach [34] and DistCache [32]. OrbitCache enables variable-length in-network caching by leveraging built-in features of the switch. SwitchKV [29] employs the Openflow switch for cache lookups while items are cached in a cache node. Although this reduces the lookup overhead, it still provides limited performance due to server-based caching.

7 Conclusion

We proposed OrbitCache, an in-network caching architecture that is capable of variable-length caching in programmable switches. The key idea of OrbitCache is to make cached key-value pairs revisit the switch data plane continuously by leveraging packet replication efficiently. Experimental results demonstrated that OrbitCache can balance skewed workloads with diverse conditions. We hope that this work can contribute to the research community by providing insights that utilizing built-in hardware features has great potential to make in-network computing solutions effective.

Acknowledgement

The author would like to thank the shepherd, Xin Jin, and the anonymous reviewers for their insightful comments and feedback. The author also thanks Yeon-sup Lim for providing constructive feedback on the earlier version of the paper. This research was sponsored by the National Research Foundation of Korea (NRF) grants funded by the Ministry of Science and ICT (No. RS-2025-00522990).

References

[1] Tommyds c library. https://www.tommyds.it/, 2018.

[2] Redis key-value store. https://redis.io/, 2023.

[3] Tofino switch. https://github.com/barefootnetworks/Open-Tofino, 2023.

[4] Nvidia messaging accelerator (vma). https://docs.nvidia.com/networking/display/vmav9840, 2024.

[5] Rocksdb: A persistent key-value store for flash and ram storage. https://rocksdb.org/, 2024.

[6] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. Fawn: a fast array of wimpy nodes. In *Proc. of ACM SOSP*, page 1–14, New York, NY, USA, 2009. Association for Computing Machinery.

[7] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proc. of ACM SIGMETRICS*, page 53–64, New York, NY, USA, 2012.

[8] Michaela Blott, Ling Liu, Kimon Karras, and Kees Vissers. Scaling out to a single-node 80gbps memcached server with 40terabytes of memory. In *Proc. of USENIX HotStorage*, pages 8–12, USA, 2015.

[9] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[10] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *Proc. of ACM SIGCOMM*, pages 99–110, 2013.

[11] Tommaso Caiazzi, Mariano Scazzariello, and Marco Chiesa. Millions of low-latency state insertions on asic switches. *Proc. of ACM CoNEXT*, 1, nov 2023.

[12] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In *Proc. of USENIX FAST*, Santa Clara, CA, February 2020.

[13] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proc. of ACM SoCC*, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

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

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

[16] Brad Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004(124):5–, August 2004.

[17] Zerui Guo, Hua Zhang, Chenxingyu Zhao, Yuebin Bai, Michael Swift, and Ming Liu. Leed: A low-power, fast persistent key-value store on smartnic jbofs. In *Proc. of ACM SIGCOMM*, page 1012–1027, New York, NY, USA, 2023. Association for Computing Machinery.

[18] Matthias Jasny, Lasse Thostrup, Tobias Ziegler, and Carsten Binnig. P4db - the case for in-network oltp. In *Proc. of ACM SIGMOD*, page 1375–1389, 2022.

[19] Theo Jepsen, Alberto Lerner, Fernando Pedone, Robert Soulé, and Philippe Cudré-Mauroux. In-network support for transaction triaging. *Proc. VLDB Endow.*, 14(9):1626–1639, may 2021.

[20] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-rtt coordination. In *Proc. of USENIX NSDI*, pages 35–49, Renton, WA, April 2018.

[21] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proc. of ACM SOSP*, pages 121–136, 2017.

[22] Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. Tea: Enabling state-intensive network functions on programmable switches. In *Proc. of ACM SIGCOMM*, page 90–106, 2020.

[23] Gyuyeong Kim. Holistic in-network acceleration for heavy-tailed storage workloads. *IEEE Access*, 11:77416–77428, 2023.

[24] Gyuyeong Kim. Netclone: Fast, scalable, and dynamic request cloning for microsecond-scale rpcs. In *Proc. of ACM SIGCOMM*, page 195–207, September 2023.

[25] Gyuyeong Kim and Wonjun Lee. In-network leaderless replication for distributed data stores. *Proc. VLDB Endow.*, 15(7):1337–1349, Mar. 2022.

[26] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. Kvell: the design and implementation of a fast persistent key-value store. In *Proc. of ACM SOSP*, page 447–461, New York, NY, USA, 2019. Association for Computing Machinery.

[27] Jialin Li, Jacob Nelson, Ellis Michael, Xin Jin, and Dan R. K. Ports. Pegasus: Tolerating skewed workloads in distributed storage with in-network coherence directories. In *Proc. of USENIX OSDI*, pages 387–406, November 2020.

[28] Junru Li, Youyou Lu, Yiming Zhang, Qing Wang, Zhuo Cheng, Keji Huang, and Jiwu Shu. Switchtx: Scalable in-network coordination for distributed transaction processing. *Proc. VLDB Endow.*, 15(11):2881–2894, jul 2022.

[29] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G. Andersen, and Michael J. Freedman. Be fast, cheap and in control with switchkv. In *Proc. of USENIX NSDI*, pages 31–44, USA, 2016.

[30] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. Mica: A holistic approach to fast in-memory key-value storage. In *Proc, of USENIX NSDI*, page 429–444, USA, 2014.

[31] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. Incbricks: Toward in-network computation with an in-network cache. In *Proc. of ACM ASPLOS*, page 795–809, New York, NY, USA, 2017.

[32] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. Distcache: Provable load balancing for large-scale storage systems with distributed caching. In *Proc. of USENIX FAST*, pages 143–157, Boston, MA, February 2019.

[33] 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. Scaling memcache at facebook. In *Proc. of USENIX NSDI*, pages 385–398, Berkeley, CA, USA, 2013.

[34] Siyuan Sheng, Huancheng Puyang, Qun Huang, Lu Tang, and Patrick P. C. Lee. FarReach: Write-back caching in programmable switches. In *Proc. of USENIX ATC*, pages 571–584, Boston, MA, July 2023. USENIX Association.

[35] Vishal Shrivastav. Stateful multi-pipelined programmable switches. In *Proc. of ACM SIGCOMM*, page 663–676, New York, NY, USA, 2022. Association for Computing Machinery.

[36] Qing Wang, Youyou Lu, Erci Xu, Junru Li, Youmin Chen, and Jiwu Shu. Concordia: Distributed shared memory with In-Network cache coherence. In *Proc. of USENIX FAST*, pages 277–292, February 2021.

[37] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *Proc. of USENIX OSDI*, pages 191–208. USENIX Association, November 2020.

[38] Parham Yassini, Khaled Diab, Saeed Mahloujifar, and Mohamed Hefeeda. Horus: Granular In-Network task scheduler for cloud datacenters. In *Proc. of USENIX NSDI*, pages 1–22, Santa Clara, CA, April 2024.

[39] Zhuolong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. Netlock: Fast, centralized lock management using programmable switches. In *Proc. of ACM SIGCOMM*, page 126–138, 2020.

[40] Hang Zhu, Zhihao Bai, Jialin Li, Ellis Michael, Dan R. K. Ports, Ion Stoica, and Xin Jin. Harmonia: Near-linear scalability for replicated storage with in-network conflict detection. *Proc. VLDB Endow.*, 13(3):376–389, November 2019.