Cascade Mapping: Optimizing Memory Efficiency for Flash-based Key-value Caching

Kefei Wang
Computer Science and Engineering
Louisiana State University
kwang@csc.lsu.edu

Feng Chen
Computer Science and Engineering
Louisiana State University
fchen@csc.lsu.edu

ABSTRACT

Flash-based key-value caching plays an important role in Internet services. Compared to in-memory key-value caches, flash-based key-value caches can provide a 10 to 100 times larger cache space, allowing to accommodate more data for a higher hit ratio. However, the current design relies on a simple hash-based indexing structure, which maintains the entire mapping table in DRAM memory. As the cache capacity continues to grow, such an "all-in-memory" approach raises concerns on cost, power, and scalability.

To address this critical memory challenge, we propose a hierarchical mapping scheme, called *Cascade Mapping*. This scheme exploits a widely existing phenomenon in key-value caches—only a small set of key-value items in the cache is frequently requested. Leveraging the strong temporal locality, we create a multi-tier mapping structure, aiming to serve the most popular key-value mappings within a small memory space and organize the majority in a highly optimized indexing structure in flash. We have implemented a prototype, called *SlickCache*, based on Twitter's Fatcache. Our experimental results show that we can achieve nearly identical performance as the conventional all-in-memory scheme, while using only a fraction (10%) of the required memory. Alternatively, this approach allows us to build a 10 times larger flash cache with the same amount of memory, which in turn increases the hit ratio by up to 8.2 times and the throughput by up to 125 times.

CCS CONCEPTS

• **Information systems → Flash memory**; **Key-value stores**; *Hierarchical storage management*; • **Software and its engineering** → *Main memory*;

KEYWORDS

Flash Memory, SSD, Key-value Cache, Memory Efficiency

ACM Reference Format:
Kefei Wang and Feng Chen. 2018. Cascade Mapping: Optimizing Memory Efficiency for Flash-based Key-value Caching. In *Proceedings of SoCC '18: ACM Symposium on Cloud Computing, Carlsbad, CA, USA, October 11–13, 2018 (SoCC '18),* 13 pages.
https://doi.org/10.1145/3267809.3267847

1 INTRODUCTION

In the past years, we have witnessed an unprecedented quick growth of Internet data. According to recent statistics in 2017, every 60 seconds, over 25,000 Tumblr posts, 210,000 Snaps, 350,000 Tweets, and 243,000 Facebook photos were uploaded [2, 12]. Much of such Internet data are managed in the form of key-values.

In order to handle the huge Internet traffic in a low latency, Internet service providers often deploy a fleet of key-value cache servers, such as Memcached [9, 33] and Redis [10], in data centers to absorb time-consuming I/Os to backend data stores [38, 45]. These memory-based key-value caches, though effective, heavily rely on a large amount of expensive DRAM memory, raising increasingly high concerns on cost, power, and scalability.

A recent trend in the industry is to adopt a more efficient alternative, *flash-based key-value cache*. Two representative systems are Facebook's McDipper [8] and Twitter's Fatcache [3]. Unlike memory-based key-value caches, flash-based key-value cache systems use high-speed flash SSDs as the main storage media for caching the key-value data, and maintain a hash-based mapping table in memory, which tracks each key-value item and its location in flash. Upon a key-value query (`GET`), the system first queries the mapping table in memory, and then loads the corresponding key-value data from the flash SSD. This design effectively guarantees that a key-value query needs no more than one flash I/O, providing satisfactorily high throughput and low latency. It is reported that McDipper allows Facebook to reduce the cache server deployment by as much as 90% while still delivering over 90% of `GET` responses with sub-millisecond latencies [8].

1.1 Scalability Challenges

Although the above-said mapping scheme works reasonably well with a relatively small-capacity flash cache, as flash capacity further grows, such an "all-in-memory" approach is reaching its scalability limit. This is for two main reasons.

First, the high index-to-data ratio of key-value cache drives the memory overhead up to an intolerably high level. A unique property of key-value caches is the dominance of *small objects*. A study on Facebook's Memcached workloads reports that 90% of the cache space is used for storing objects smaller than 500 bytes [18], meaning that indexing a huge number of small key-value items in a large-capacity flash cache demands an excessively large amount of DRAM memory. For example, Twitter's Fatcache uses a 44-byte index entry for each key-value item. Assuming an average key-value item size of 300 bytes, we need about 150 GB of memory space to index a 1-TB flash cache, solely for the mappings, not to mention the other overhead. As a consequence, we are facing an undesirable but must-address dilemma—the quick growth of flash capacity allows us to accommodate more data in cache, but we lack sufficient amount of DRAM memory to index these data, which in effect limits the usable cache space.

Second, it is a well-known fact that the capacity of DRAM memory grows at a much slower pace (25-40% per year) than flash memory (50-60% per year) [35]. In today’s market, the cost of 1-TB flash is already below $500, while the same amount of server-grade DDR4 memory could cost over $10,000. The widening capacity and price gaps have a strong implication for us—it will become increasingly more difficult to have *proportionally* enough memory to index all the key-value data in the future. Further considering that the capacity of DRAM memory in a system is also limited by the processors and the DIMM slot budget on the motherboard, users would have to purchase unnecessarily high-end processors or multi-socket systems, simply for the purpose of being able to install more memory on board, further boosting the cost up.

In short, the current mapping structure design in flash-based key-value cache is cost-inefficient and unable to scale in a long run. We need an alternative solution.

1.2 Cascade Mapping

In this paper, we present a highly efficient hierarchical mapping scheme, called *Cascade Mapping*, to address the above-said memory challenge for flash-based key-value caching.

Essentially, the struggles in the current design are all due to an unrealistic attempt to maintain the entire indexing structure in memory, which is understandable for the performance reason but unfortunately infeasible in reality. We argue that with a careful design and implementation, we do *not* need to preserve all the key-value mappings in memory, and we will still be able to achieve nearly identical performance as the all-in-memory solution by using only a fraction of the needed memory.

This seemingly ambitious goal is indeed achievable due to a simple fact—only a small set of key-value items in the cache is of high interest and being frequently accessed [18]. Leveraging the strong temporal locality in key-value workloads, we have designed a hierarchical mapping structure to differentiate the key-value mappings based on their popularity and manage them in multiple tiers. Such a differentiation offers us an opportunity to optimize the use of the precious memory space, holding the most important key-value mappings in DRAM memory, while leaving the majority in flash. This approach would naturally involve extra flash I/Os. We have designed a set of effective techniques to minimize the incurred flash I/Os and their effects on performance.

Cascade Mapping is a three-tier mapping structure. Tier 1 maintains the “hot” mappings in memory, serving most incoming query requests and incurring no flash I/Os for mapping lookups; Tier 2 retains the “warm” mappings in a highly efficient structure in flash, supporting a fast, parallel search among a batch of blocks in a high-bandwidth manner; Tier 3 holds the majority of the mappings in a memory-efficient structure on flash, serving a small percentage of requests to the “cold” key-value mappings.

Cascade Mapping is designed to not only reduce the memory demand but also deliver fast and reliable caching services. For this purpose, our design exploits the unique properties of the hierarchical mapping structure and is particularly optimized for key-value caches. Leveraging the multi-tier structure, the Garbage Collection (GC) procedure is optimized to improve the cache hit ratio. Exploiting the persistent mappings and the FIFO-based slab management, a highly efficient checkpointing mechanism is designed for quick and reliable crash recovery.

We have implemented a prototype, called *SlickCache*, based on Twitter’s Fatcache with the proposed Cascade Mapping scheme. Our experimental results show that SlickCache can achieve nearly identical performance as the all-in-memory mapping scheme by using only a fraction (10%) of the needed memory. Alternatively, for a given memory capacity, we can build a 10 times larger flash cache, which in turn increases the hit ratio by up to 8.2 times and the throughput by up to 125 times. Our design only introduces a small (about 6%) flash space overhead, which is trivial compared to the significant memory saving and performance improvement.

The rest of the paper is organized as follows. Section 2 and Section 3 present the background and motivations. Section 4 introduces the design. Section 5 discusses evaluation results. Related work is presented in Section 6. The final section concludes this paper.

2 BACKGROUND

In this section, we briefly introduce flash memory, SSDs, and the current flash-based key-value cache system design.

Flash memory. A typical NAND flash memory chip is composed of multiple *planes*. Each plane contains thousands of *blocks*. A block consists of multiple *pages*. Flash memory supports three major operations, `read`, `write`, and `erase`. Reads are typically fast (10s of microseconds), and writes are relatively slow (100s of microseconds). Reads and writes are normally performed in pages, while erases must be performed in units of blocks. A unique constraint is that the pages in a block must be written in a sequential manner. Once a page is programmed (written), it cannot be overwritten again until the entire block is erased, which takes several milliseconds.

Flash SSD. A flash SSD includes several components, including a host interface logic, an SSD controller, a dedicated RAM buffer, flash memory controllers and chips. To provide a high bandwidth, modern SSDs often have multiple (e.g., 2-10) *channels* to connect the controllers to flash memory chips. A channel may be shared by multiple chips. To address the technical constraints of flash memory, a *Flash Translation Layer* (FTL) is implemented in the firmware for flash management, such as garbage collection and wear-leveling, and to provide a generic Logical Block Address (LBA) interface to mimic a Hard Disk Drive (HDD). More details can be found in prior studies [14, 22, 25, 26]. In this paper, we use “flash” and “flash SSD” interchangeably, unless otherwise specified.

Flash-based Key-value Cache. Flash memory provides high bandwidth and low latency [14, 25], which makes it suitable to serve as a cache or fast storage in various environments [17, 23, 24, 36, 38, 41, 43, 44, 47, 48]. A recent application is to move key-value cache from memory to flash. Two popular examples are Facebook’s McDipper [8] and Twitter’s Fatcache [3], which use flash as the storage media to provide key-value cache services. Here we take Fatcache as an example to briefly introduce the current design.

Fatcache adopts a classic slab-based space management [20], following the Memcached protocol. In Fatcache, the flash space is segmented into fixed-size *slabs*, each being further divided into a group of fixed-size *slots*. Each slot stores a key-value item. The slabs are logically grouped in different *slab classes* based on the slot size.

Given a key-value item, the smallest slot that can accommodate the key-value and the related metadata is selected. A hash table is maintained in DRAM memory to index the key-value items stored in flash. A query (GET) is completed by searching the mapping table to find the location (slab/slot) of the corresponding value in flash and then loading it into memory. An update (SET) writes the data to a new location in flash and updates the mapping accordingly. Deleting a key-value item (DELETE) only removes the mapping from the hash table. The deleted or obsolete key-value items are left in the slabs for Garbage Collection (GC) to reclaim later.

3 MOTIVATIONS

A large capacity of flash space allows us to build a huge cache to accommodate more key-value data. However, to locate and retrieve the target data, the key-value items cached in flash must be indexed. The current all-in-memory mapping scheme, unfortunately, demands an excessively large amount of memory. Here we discuss several critical issues that motivated this work.

3.1 Inefficiency of All-in-memory Mapping

The current design of flash-based key-value cache is a close emulation of their memory-based counterpart. As the main reason to adopt flash for caching is to expand the cache space, a natural solution is to directly move the key-value data into flash, while leaving the mapping structure, which receives intensive small and random accesses, in DRAM memory for fast queries.

Such an approach is effective (only one flash I/O is needed), but *unscalable*. Due to the small size of key-value items in typical Internet services, the memory demand for holding the mappings entirely in memory turns DRAM memory into a critical and expensive bottleneck. In the current Fatcache, as mentioned previously, 150 GB of memory is needed for indexing 300-byte key-values in a 1-TB cache. For the same reason, the limited memory capacity would become the factor constraining the usable flash cache space. In the same example, assuming a memory capacity of 75 GB, only half of the flash space would be usable for caching.

The root cause of the above-said large memory demand problem is that, the current design simply assumes all the key-value items are equally important, which is a *false assumption*. A recent analysis on Facebook's workloads points out that a strong locality is observed in real-world cache systems—a few key-value items are accessed millions of times a day, while most key-values are only accessed a handful of times [18]. Further considering that a typical flash key-value cache is 10 to 100 times larger than an in-memory cache, we cannot continue to assume all the data are the same and treat them equally. In other words, it is unnecessary to hold them all in memory. We may reduce the memory needs by only holding the mappings for the most popular key-values in memory.

3.2 Limitations of Swapping

A possible solution to address the memory challenge is to directly use a flash SSD as a swap device [37]. Modern operating systems, such as Linux, allow to use an external block device as a swap space to temporarily store "cold" memory pages. For key-value caching, unfortunately, such a solution is unsuitable. This is for several reasons unique to key-value cache systems.

First, compared to a mapping entry (44 bytes in Fatcache), the system-level swapping happens in a much coarser granularity (4-KB page in default). An access to a hot mapping entry would misguide the operating system to label the whole page as a hot page, even when the other entries in the same page could be cold. Second, due to its random nature, the hash mapping entries are uniformly distributed in the hash space, meaning that hot and cold entries are mixed together and evenly distributed in the hash space. Consequently, it is difficult to differentiate the pages based on their access temperature. Third, since the flash-based key-value cache writes data in an out-of-place manner, each SET operation would result in an update to the corresponding mapping entry. If directly using flash as a swap space, a large number of small, random writes would be generated, which is an undesirable I/O pattern for flash devices [14, 25]. In Section 5.5, we will show an experimental comparison of our scheme with the swapping solution.

4 DESIGN

To address the memory challenge, we propose a hierarchical mapping scheme, called *Cascade Mapping*, for efficiently managing a large-capacity flash-based key-value cache.

4.1 Cascade Mapping Structure

A critical component in key-value cache is the mapping structure. Upon a query, the mapping table must be first looked up to find the key-value's location in flash. Thus, its design directly affects performance. Our goal is to minimize the memory usage of the mapping structure while still achieving high performance.

In this paper, we present a highly efficient mapping structure, called *Cascade Mapping*, which exploits the strong temporal locality in key-value workloads. As illustrated in Figure 1, Cascade Mapping is a hierarchical structure consisting of three tiers, one in memory and two in flash:

- **Tier 1**: A hash-based mapping table is completely maintained in memory to record the mappings of *hot* key-value items. Though small, this in-memory hash mapping structure is expected to serve the majority of incoming queries, incurring no flash I/Os.
- **Tier 2**: A set of mapping blocks is maintained in flash to record the mappings of *warm* key-value items, which are the most recently demoted from the first tier. Leveraging the internal parallelism of modern flash SSDs, a lookup in a batch of blocks can be quickly completed in a parallelized, high-bandwidth manner.
- **Tier 3**: A dual-mode hash table resides in flash and maintains two bucket-and-link based structures to manage the mappings of *cold* key-value items, which account for the most cache space but receive only a small percentage of queries.

With such a 3-tier mapping structure, we aim to move the infrequently queried key-value mappings out of the DRAM memory to the high-speed flash SSD, allowing us to significantly reduce the demand for memory while still being able to deliver nearly identical performance. To achieve this goal, the key challenge is—*how can we minimize the involved flash I/Os as much as possible?* In the following, we will discuss the design details of each tier and the techniques to address this challenge.

4.1.1 *Tier 1: An In-memory Indexing Table.*

As discussed previously, the strong locality of key-value accesses implies that only a small set of mappings is frequently requested [18]. The Tier-1 indexing structure maintains the *hot* mappings in DRAM memory to handle most queries with a fast response time.

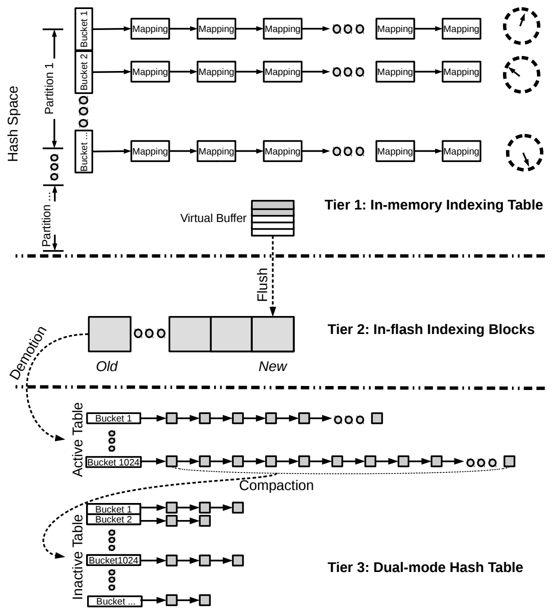


Figure 1: An Illustration of Cascade Mapping Structure.

Figure 1 illustrates the design of the in-memory indexing structure. We first divide the hash space into multiple logical *partitions*. This is for two purposes. First, each partition can be operated independently. Second, the multiple partitions allow us to create parallelized I/Os. During demotion, each partition can independently produce an I/O stream and flush the demoted mapping entries to the SSD in parallel, fully exploiting the great bandwidth potential of flash devices [22, 26].

Caching Policy. To evict the cold mappings, a simple Least Recently Used (LRU) policy can be used to track the access recency of the key-values as in Memcached. This approach is accurate but demands extra memory space for maintaining the LRU stack. Similar to prior work [32], we adopt the CLOCK algorithm [27], which approximates the LRU replacement, working as follows.

We associate each key-value mapping entry with a *reference* bit, initialized to 0. Upon a request, the mapping entry's reference bit is set to 1. Thus, a reference bit of 0 indicates that the mapping has not been recently accessed, i.e., the cold mappings for demotion.

When the number of Tier-1 mappings exceeds a predefined limit (e.g., 10% of the entire set of mapping entries), we demote roughly an equal number of cold mappings from each bucket to the second tier. Assuming N mapping entries to be demoted from M hash buckets, each hash bucket needs to demote $n = \lceil \frac{N}{M} \rceil$ mapping entries. We start from the first bucket and scan the list of mapping entries. If the entry has a reference bit of 0, its mapping entry ID is recorded for demotion; otherwise, we reset its reference bit to 0 and skip it. If n entries are collected, we move to the next hash bucket. This process repeats until all N mapping entries are collected.

Virtual Buffering. During the demotion, the cold mapping entries need to be written to the flash SSD. In order to create a large, sequential I/O pattern, we virtually maintain a pool of cold mapping entries by recording their mapping entry IDs (not the mapping content). The pool size is equal to the direct mapping block in flash (see Section 4.1.2). Once the pool is filled up, we copy the mapping entries into a dynamically allocated buffer and flush to the flash SSD in bulk. This design reduces the memory overhead and ensures to generate large flash I/Os, being optimal for flash devices.

4.1.2 Tier 2: A Direct Indexing in Flash.

The second mapping tier uses flash to store the mapping entries demoted from Tier 1. It can be regarded as an *in-flash* extension of the Tier-1 indexing structure, holding the *warm* key-value mappings, which receive less but considerable amount of requests.

The basic operation unit in the Tier-2 indexing structure is a *direct mapping block*, which contains a set of key-value mapping entries demoted from the first tier as described above. For each partition, the second tier maintains in flash an array of direct mapping blocks, which are organized in the First-In-First-Out (FIFO) order. When a new direct mapping block is added into the second tier, correspondingly the oldest one is demoted to the third tier, creating a "waterfall" process.

Compared to Tier 1, which allows a quick hash-index based search in memory, the entries of the Tier-2 mapping blocks are not indexed, simply being stored in flash in their original FIFO order. It holds two advantages: First, it saves memory and simplifies the design. Second, it allows us to easily determine the most up-to-date mapping according to the FIFO order (the newest one always appears in the latest position). Also, unlike in a linked list structure, multiple mapping blocks can be loaded simultaneously.

Parallelized Batch Search. To achieve a quick search within the array of direct mapping blocks, the key idea is to leverage the high bandwidth of a flash SSD, which is enabled by its rich *internal parallelism* [22, 26]. It works as follows.

Upon a query, we first identify the hash partition with the hashed key. Then we load a batch of direct mapping blocks from flash using *parallel* I/Os, starting from the newest block. In one batch, each I/O thread loads one mapping block and directly carries out a one-to-one comparison in memory to search for the target key. Since modern flash SSDs are able to provide a high bandwidth (e.g., 100s of Megabytes/Sec to Gigabytes/Sec), such parallel I/O operations can be completed at a very high speed—multiple parallel I/Os in effect can be completed in the time of one flash I/O, because data can be simultaneously transferred over multiple channels internally [22, 26]. This process is repeated until finding the target key or completing the scan of all the direct mapping blocks.

The block size can affect the response time. A large block size (e.g., 128 KB) is more I/O efficient, but it takes longer to load and scan a block each time; a smaller block size (e.g., 4 KB) splits the scan into multiple batches, but due to the FIFO order, if the target mapping is found, it can immediately stop and return.

The optimal parallelism degree is determined by the available flash channels and CPU cores. According to prior work [22, 26], a reasonable number of parallel I/Os is sufficient to fully utilize the bandwidth. In our experiments, we find that 8 threads with 16-KB blocks work well in most cases. We will study the effect of the mapping block size and the parallelism degree in Section 5.4.

4.1.3 Tier 3: A Dual-mode Linked Hash Table.

Tier 3 organizes a bucket-and-link based hash indexing structure in flash to manage the mappings for the *cold* key-values, which account for the majority in the cache. The challenge is how to minimize the memory usage and also the incurred flash I/Os.

A naïve solution is to divide the key-value mappings into a set of hash *buckets* based on the key's hash index. Each bucket has a

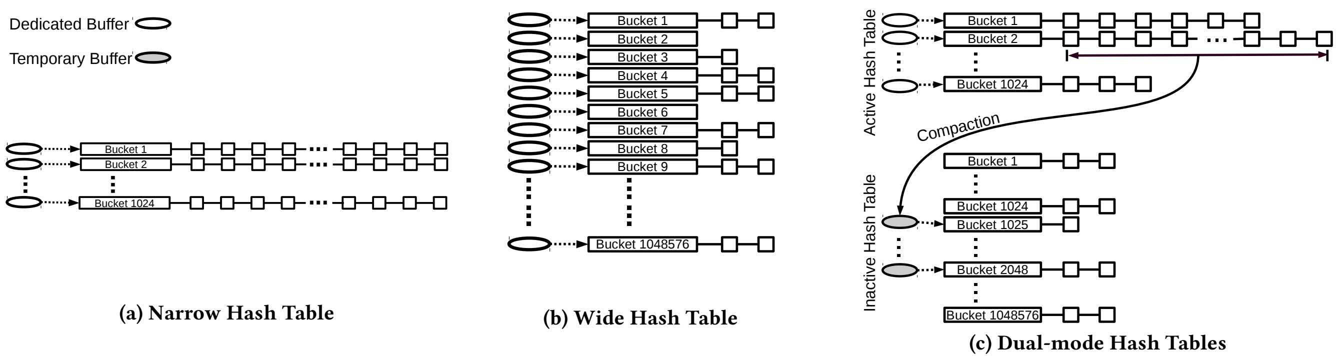


Figure 2: A Comparative Illustration of Narrow, Wide, and Dual-mode Hash Tables. *The ellipses represent the block buffers in memory. The white boxes represent the indirect mapping blocks stored in flash.*

linked list of *indirect mapping blocks*, each of which stores a set of mapping entries on flash, forming a FIFO queue. Upon a query, the hash index is taken to find the corresponding bucket and then traverse the link of blocks to search for the queried key.

The above-said hypothetical in-flash hash mapping structure raises a critical issue in practice: Since we cannot write each individual, small hash mapping entry into the flash (otherwise, it would result in many small, random writes), we have to first temporarily hold them in a memory buffer, whose size is equal to the indirect mapping block. When the buffer is filled up, we flush the entire mapping block into flash to avoid the flash-unfriendly, random write traffic. Each bucket needs to have such a buffer.

A technical dilemma emerges—if we maintain a large number of hash buckets (e.g., 1 million buckets), the list of the corresponding bucket would be short, meaning that less I/Os are needed for walking the linked list. However, it would suffer from high memory overhead. Assuming a 4-KB block size, the buffers would account for about 4 GB memory in total, which is non-trivial. On the contrary, if we maintain a small number of hash buckets (e.g., 1 thousand buckets), the memory consumption is low (only 4 MB), but the list of the indirect mapping blocks in each bucket would be excessively long, meaning that a large number of flash I/Os would be involved for the list walk, causing performance issues.

Dual-mode Mapping. To address this critical challenge, we have developed a *Dual-mode Mapping* scheme. We maintain two bucket-and-link structures on flash, an *active hash table* and an *inactive hash table*. The former is a narrow hash table, which has a small number of hash buckets, while the latter is a wide hash table, which has a large number of hash buckets. The key difference between the two lies on the memory buffers: In the active hash table, each bucket has a dedicated memory buffer; in the inactive hash table, for the sake of memory saving, only a subset of the buckets are given a dynamically allocated temporary buffer.

The dual-mode mapping works as a two-level structure (see Figure 2c). The key-value mappings demoted from Tier 2 are first accommodated in the buffer of the corresponding hash bucket of the active hash table. When it is filled up, the buffered mappings are flushed into an indirect mapping block, which is added to the head of the corresponding bucket list in the FIFO order. When the list length exceeds a threshold (e.g., 64 blocks), the entire FIFO list is taken off the active bucket, and a *compaction procedure* is launched to move the mappings into the inactive hash table.

The compaction procedure walks through the full list of the active hash table, loads each key-value mapping entry, and inserts it into a dynamically allocated buffer of the corresponding hash bucket in the inactive hash table. Note that since the inactive table is much "wider" than the active table, compacting one active hash list could involve a range of inactive hash buckets. For example, assuming that we have 1,024 active hash buckets and 1,048,576 inactive hash buckets (i.e., *Inactive/Active Ratio* of 1,024), it would take 1,024 inactive buckets to cover the range of one active bucket in the hash space. Since each time the compaction procedure only handles one active hash bucket, it would *only* involve a small set of inactive hash buckets at a time.

The dual-mode mapping design dramatically reduces the need for memory. In the worst case, we only need 2,048 block buffers (1,024 for the active table and 1,024 for the inactive table) in the example above. It allows us to enjoy the advantages of both—the narrow active table allows us to reserve only a small amount of memory to maintain the buffers for collecting incoming mappings; the wide inactive flash table allows us to walk through a short list and use only a few flash I/Os to load the blocks for a lookup.

Upon a query, the active hash table is first searched and then the inactive hash table. The active and inactive hash tables do not form a cache hierarchy but simply a two-level structure, i.e., the active hash table is essentially a large buffer. We will study the effect of compaction and the active table list length in Section 5.4.

Jump List. Compared to the Tier-2 indexing, which allows a fast, parallel scanning of a batch of blocks in a high-bandwidth manner, the Tier-3 hash tables organize the mapping blocks in a linked list structure, which is suitable for lookups among a large and growing amount of mapping entries. However, walking through a long hash bucket list would cause a chain of flash I/Os.

In the narrow active hash table, each hash bucket has a relatively long list of mapping blocks, meaning that a query may involve many flash I/Os in a sequence. We present a simple method, called *Jump List*, for optimization. Every D blocks, we insert a Bloom filter [1] in the list to indicate the existence of the key-value mappings in the following D blocks. When walking through the FIFO list, we first check the Bloom filter, if it indicates that the queried key is not resident in the following D blocks, we simply skip these blocks and check the next Bloom filter, and so on.

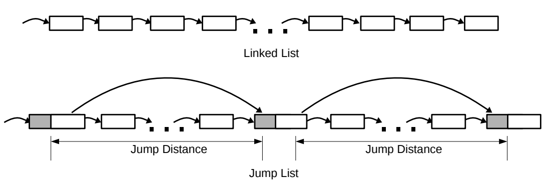


Figure 3: Linked List vs. Jump List. *The grey boxes represent the Bloom filters. The white boxes represent the mapping blocks.*

We set the jump distance D as $\lceil \sqrt{K} \rceil$, where K is the list length (in blocks). It ensures that scanning the list would take no more than $D \times 2$ flash I/Os. In our prototype, we use a relatively large block size, 128 KB, since the jump list search is a serial operation and a large I/O request size is more efficient [25].

For I/O efficiency, a Bloom filter and its immediate following mapping block are stored together in flash, and also loaded together into memory during the list walk (see Figure 3). We have considered to add a large Bloom filter to cover the entire list, but we find that as the list grows, the Bloom filter needs to be updated for each insertion and its effectiveness also quickly weakens. Our prototype uses a 128-KB Bloom filter for every 8 blocks (2 bytes per mapping entry), which has a false positive rate of 0.5%.

4.2 Key-value Cache Operations

There are four basic operations in SlickCache, read, update, insertion, and deletion. The workflow of each one is as follows.

Read. Upon a key-value query (GET), SlickCache first calculates a hash value $Hash(key)$ for the key being searched. (1) We first look up the hashed key in the Tier-1 hash table of the corresponding partition in memory. If found, we follow the pointer (*Slab_ID* and *Offset*) to retrieve its value data from the flash SSD. If the hashed key is not found in memory, we continue the search on the Tier-2 indexing structure in flash. (2) In Tier 2, multiple parallel I/Os load the direct mapping blocks into memory to quickly search for the target key. It may take multiple batches, and if the key is found, the search stops. If more than one matched record is found, the latest one is returned. If the above two steps both fail, which is a relatively rare case, we need to search among the Tier-3 mappings via the dual-mode hash tables. (3) In Tier 3, the active hash table is searched first by walking through the list of indirect mapping blocks. Jump list and Bloom filter accelerate this process. If nothing found, we continue to search the inactive hash table. In the above steps, if the mapping is found in flash, we promote it to Tier 1 and retrieve the value data; otherwise, the key-value item is not cached, and the client needs to request the data directly from the backend data store and then inserts the key-value into SlickCache.

Insertion/Update. In SlickCache, insertion and update are treated in a similar way. Upon an insertion request, SlickCache first writes the key-value data into a slab, which is selected according to the slot size. Then we add its mapping entry to the Tier-1 hash table in memory. Update is similar, except that if the mapping is found in memory, we simply update the mapping entry with the key-value item's new location; otherwise, a new mapping is created and inserted into the Tier-1 in-memory hash table.

Deletion. In Fatcache, deleting a key-value item simply removes its mapping with no further actions. With our in-flash mapping design, this does not suffice to ensure that the deleted data are unreachable, since the mapping could still be found in the two in-flash tiers. There are two possible solutions: (1) Search and delete the mappings of the target key from flash, or (2) insert a "tombstone" key-value with a magic number (a randomly selected 160-bit SHA-1 hash) in the value field and mark the mapping entry as "deleted". Both have advantages and disadvantages. The former incurs high time overhead, while the latter consumes extra flash space. Considering the performance requirement and the abundance of flash capacity, we choose the second option and let GC to reclaim the space later. It is also worth noting that inserting the tombstone key-values in effect logs the deletion operations, which facilitates safe crash recovery (see Section 4.4).

4.3 Garbage Collection

Flash SSDs favor large and sequential writes. Upon operations, such as deletion and update, the flash cache simply writes the update to a new location and changes the mapping accordingly. The deleted or obsolete data are left in the original slab. When the system runs low on free slabs, Garbage Collection (GC) is triggered to reclaim the oldest slab in the FIFO order.

The victim slab may contain both alive and invalid (deleted or obsolete) data. SlickCache implements two eviction schemes.

• *Space-oriented Eviction*: A simple scheme is to erase the entire slab, disregarding the aliveness of the key-values in the slab. This process is fast, since no data need to be copied to a new location. However, due to the FIFO nature, such a simple approach may blindly evict hot data, negatively affecting the hit ratio. Twitter's Fatcache adopts such a scheme for its simplicity.

• *Hit Ratio-oriented Eviction*: Cascade Mapping has a unique advantage that allows us to easily obtain the locality information—if a key-value can be found in the Tier-1 mapping, it implies that this is a hot item. Leveraging this opportunity, we can quickly identify the locality of a key-value item in the victim slab by querying the Tier-1 mapping. If found, this key-value is reasonably hot, then we copy it to a new slab, preserving frequently accessed key-values in cache. Otherwise, we simply drop it. Any deleted or obsolete item will also be dropped during this process, since it is impossible to be referenced by any valid mapping entry in memory.

The above-said two policies both have advantages and disadvantages. The former can quickly reclaim space but may impair the hit ratio, while the latter incurs costly data copy but retains a high hit ratio. Thus, the former is more suitable for urgent situations where free slabs are needed immediately, while the latter is suitable for maintaining a long-term cache effectiveness.

Adaptive Two-phase GC. SlickCache adopts an adaptive two-phase GC approach, similar to prior work [48]. It dynamically switches between the two policies. We set a low watermark W_{low} and a high watermark W_{high}. (1) When the system is under high pressure (the number of free slabs is below W_{low}), a quick space reclamation is needed, so the space-oriented eviction is triggered to free slab space quickly. (2) When the number of free slabs reaches a moderate level (between W_{low} and W_{high}), the GC policy switches to the hit ratio-oriented eviction for a better overall performance. (3) When the number of free slabs reaches a high level (above W_{high}), the GC stops until it falls below the watermarks again. We will study the effect of GC and the watermarks in Section 5.4.

Zero-I/O Demapping. When reclaiming the alive key-values in a victim slab, we must remove the corresponding mappings from the indexing structure (i.e., demapping). Otherwise, they will become dangling pointers. A critical challenge in our scheme is how to invalidate the in-flash mappings without incurring costly I/Os.

A simple solution is to arbitrarily generate a "delete" request to directly remove the key-value. However, it would generate much traffic to the indexing structure and the key-value cache, incurring non-trivial overhead. In the following, we present an zero-I/O demapping scheme.

We maintain a global *Slab Sequence Counter*, S_g, which increments upon each allocation of a free slab. The *Slab_ID* recorded in the mapping entry is a combination of the ID of the physical slab and the global sequence counter value upon allocation. Since the slabs are reclaimed in the FIFO order, if a slab i's sequence number, s_i, is no greater than $S_g - N$, where N is the total number of slabs, it must have already been reclaimed. Leveraging this rule, we can simply reclaim the key-values in the victim slab without explicitly removing the mappings. This is safe—upon a request, after the mapping lookup, we can easily recognize if the found mapping entry points to an already-reclaimed slab or not, since a reclaimed slab's sequence number must be no greater than $S_g - N$. If true, we simply discard it and return; otherwise, we can safely retrieve the data. Using this demapping strategy, no flash I/Os are needed during GC. The obsolete mappings only consume a trivial amount of flash space. If needed, they can be removed offline by a routine service, which is not implemented in our current prototype.

4.4 Crash Recovery

A critical problem with the all-in-memory mapping approach is that upon power failure or system crash, the mapping information would be destroyed, rendering the loss of the entire key-value cache. As a result, the cache has to be warmed up again, which often takes a long period of time (hours or days) for a large cache [51].

In Cascade Mapping, when the system is restarted, only a small amount of in-memory mappings would be lost, but the in-flash mappings would survive, making it possible to keep the cache warm across power cycles. However, a complication must be addressed: A mapping entry stored in flash could be outdated, since a more recent version may have existed in memory before the crash. In such a case, an obsolete value would be returned erroneously.

A possible solution is to log every change made to the in-memory mappings, such as deletion and update. Although it ensures that all the in-memory updates be made persistent immediately, this approach would incur many small, synchronous writes to flash, affecting performance. A battery-backed NVM can partially address the problem, but it would cause additional cost. We have developed a more efficient solution to handle this situation.

Slab-based Log Recovery. Our method is a type of data journaling [46] for efficient crash recovery. We leverage the fact that the slabs are allocated and reclaimed in the FIFO order. As each slab slot contains the complete key-value information, the FIFO sequence of slabs in effect forms a big "log", i.e., journal.

- *Checkpointing*: We periodically flush the dirty in-memory mappings and record the latest slab sequence counter (see Section 4.3) in a reserved space in flash, as a *checkpoint*. It indicates the point before which all the mappings have been persistently saved.
- *Crash Recovery*: Upon crash recovery, we first reload the mappings from the reserved space to restore the in-memory mappings to the latest checkpoint. Then we scan the slabs in the order of their sequence numbers, starting from the one right after the checkpoint, to reconstruct the mapping structure by rolling it forward to the most recent state before the crash.

It is possible that a restored mapping entry might also be in flash due to the demotion. It is safe, since the mappings are in the FIFO order. We may avoid such duplication by querying each recovered key, but it would significantly extend the recovery time. As crash recovery happens rarely and such a flash space waste is minimal, we choose to simply leave them in flash and phase them out by GC. We will study the effect of crash recovery in Section 5.6.

Discussions. The crash recovery process can quickly restore the cache to its original state before the crash. In practice, it is still possible that the cache server becomes out of sync with the backend data store during the short downtime. In fact, it is a common challenge for persistent caches, especially in distributed environment [31]. Although completely addressing this issue is out of the scope of this paper, simple solutions exist. For example, the client or a proxy can log the uncommitted changes during the downtime and resubmit (or invalidate) the out-of-sync key-values when the cache server is back on-line. The bottom line is, if necessary, the entire cache can be easily invalidated, which in effect makes the handling of system crash identical to the current key-value cache design.

5 EVALUATION

5.1 Implementation

We have prototyped SlickCache based on Twitter's Fatcache [3]. Our implementation adds about 3,800 lines of code in C to Fatcache-Async [4, 48]. Several implementation details are as follows.

Our current prototype divides the hash space into 32 partitions in Tier 1. We currently reserve dedicated space for the reference bits for the CLOCK algorithm. A bit in the unused `expiry` of the mapping entry could be borrowed as the reference bit. In Tier 2, we create a shared pool of 64 threads using `pthread` to carry out the parallel I/Os. For the compaction in Tier 3, we maintain a temporary buffer for each involved inactive hash bucket. If the active hash list is short and the inactive/active ratio is large, the buffer may not be completely filled up for a large (128 KB) mapping block. In our current prototype, we bypass this issue by reserving a block-size space in flash but allowing to append data multiple times. Each I/O must be no smaller than 4 KB. Another solution is to use a smaller inactive block size, but it would incur more I/Os upon queries.

5.2 Experimental Setup

Our experiments are conducted on three Lenovo ThinkServers. Each server is equipped with a 4-core Intel Xeon 3.4 GHz processor, 16 GB memory, and a 7,200 RPM 1-TB Seagate disk drive. The cache server is equipped with a 240-GB Intel 730 SSD, which is used as our cache media, and runs 8 key-value cache instances as in prior work [48]. To complete our tests in a reasonable time frame, we only use part of the SSD capacity for caching in our experiments, being proportional (6-12%) to the workload's dataset size. Our backend data store is a MongoDB 3.4 database running on

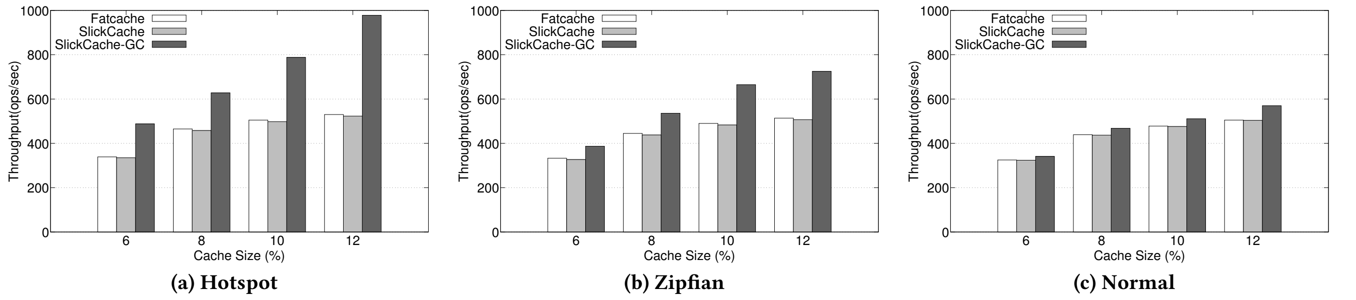


Figure 4: Overall Performance with Fixed Cache Size. *The key-value cache size is set in proportion (6% to 12%) to the workload's dataset size. Results are collected for the workloads with Hotspot, Zipfian, and Normal distributions.*

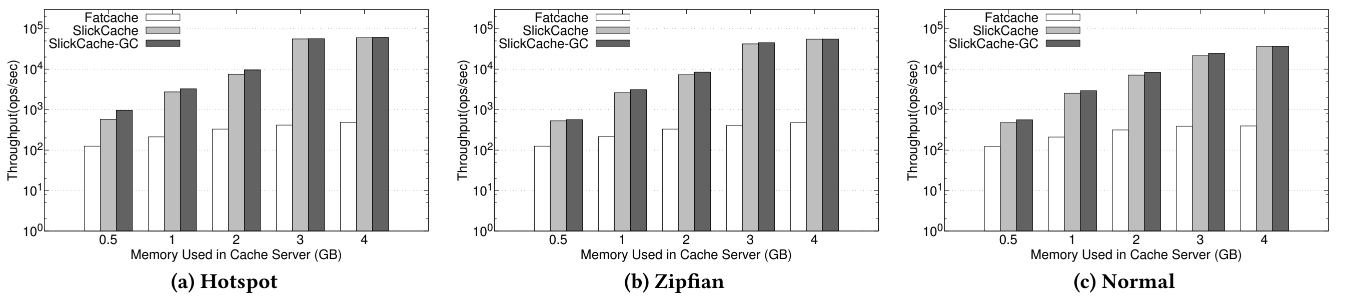


Figure 5: Overall Performance with Fixed Memory Size. *The memory portion is configured to hold 10% of the entire set of mapping entries for SlickCache and SlickCache-GC. Results are collected for the workloads with Hotspot, Zipfian, and Normal distributions.*

a 7,200 RPM Seagate 2-TB hard drive. The system and database are installed on two separate disks. The client server runs 32 clients for generating requests to the cache server. The three machines are connected in a 1Gbps local Ethernet network. For the software, we use Ubuntu 16.04 with Linux kernel 4.12 and Ext4 file system.

We use the Yahoo! Cloud Serving Benchmark (YCSB) [6, 13] as the tool to generate key-value workloads, following three popularity distributions, *Hotspot* and *Zipfian* in YCSB, and *Normal* as described in prior work [21, 49], to simulate the traffic in cloud services. The key-value data content is not of interest in this study and thus filled with random data. Our synthesized workloads follow the size distribution reported in prior work [18]. The average key size is 30 bytes and average value size is 270 bytes. All throughput and latency data are collected on the client machine.

In our evaluation, we run *Fatcache* with all-in-memory mapping, *SlickCache* with Cascade Mapping, and *SlickCache-GC*, which is further optimized with our two-phase GC (see Section 4.3). Our first experiment set evaluates overall performance with a complete system setup. Then we focus on the cache server to study each component individually. Finally, we give the overhead analysis.

5.3 Overall Performance

We first evaluate the overall system performance, given a fixed flash cache size or a fixed memory size. Our experimental system simulates a typical key-value caching environment, consisting of multiple clients, a key-value cache server, and a MongoDB database server as the backend. The parameter settings of SlickCache are as described in Section 5.4.

• **Fixed Cache Size**. For a given flash capacity, SlickCache is expected to consume significantly less memory while still retaining satisfactory performance. We vary the cache size in a range from 6% to 12% of the entire workload dataset. We first generate 1 billion key-value pairs, which account for about 300 GB. All key-value pairs are inserted into MongoDB, indexed by the keys. We generate 500 million access requests to the cache server with a `SET/GET` ratio of 1:30 as described in prior work [18]. We first warm up the cache with the first 400 million requests, and then collect results for the rest 100 million requests.

Our SlickCache is configured to hold the hottest (10%) mappings in memory at Tier 1. As we use direct I/O in our experiments, the memory is only for storing the mappings not for caching data. A possible use of the saved memory is to create a data cache to further improve performance. We leave it as our future study.

Although SlickCache only uses about 10% of the memory used by Fatcache, it still achieves comparable or even better performance. Figure 4 shows the throughput results. Despite the much lower memory usage and more complicated indexing structure, SlickCache achieves comparable performance to the stock Fatcache (less than 1% difference). SlickCache-GC further increases the throughput by 85% (in Hotspot distribution) when cache size is set to 12%. Under such setting, the hit ratios for Fatcache and SlickCache are 67.8% and 90.2%, respectively. This is due to the optimized GC policy, which leverages the locality information enabled by our multi-tier design and effectively keeps the hot key-value items in the cache server (see Section 4.3), yielding a higher performance.

• **Fixed Memory Size**. As an alternative approach for optimization, for a given memory space, SlickCache can significantly enlarge the usable cache space by maintaining part of the mapping table in flash, allowing us to deploy a much larger cache than the stock Fatcache, whose usable cache space is fundamentally limited by the DRAM memory capacity. In this experiment, we use the same data set as above and vary the total usable memory space in the

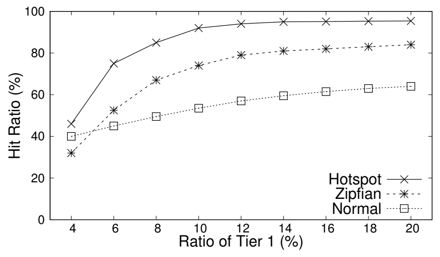


Figure 6: Hit Ratio vs. Memory Size (CLOCK).

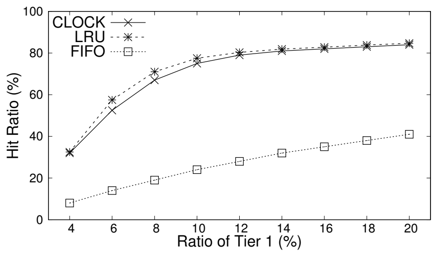


Figure 7: Replacement Algorithms.

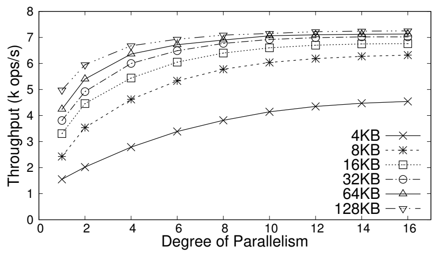


Figure 8: Parallelism vs. Size (Throughput).

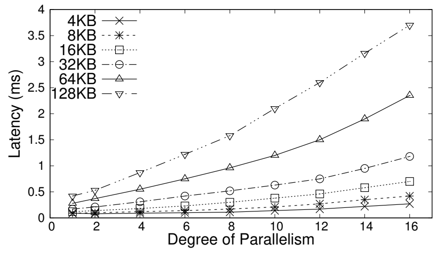


Figure 9: Parallelism vs. Size (Latency).

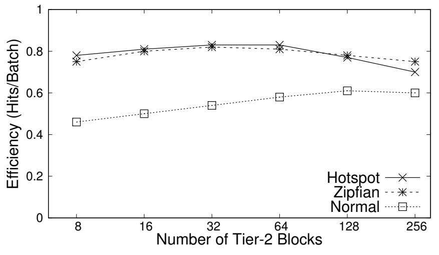


Figure 10: Efficiency vs. Number of Blocks.

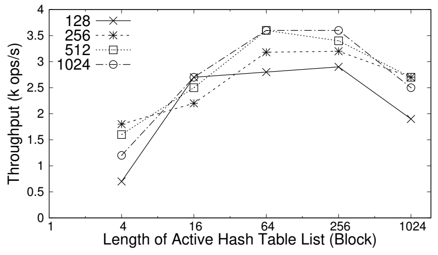


Figure 11: Length of Active Hash Table List.

system from 512 MB to 4 GB. The Tier-1 is configured to cover 10% of the entire mapping entries. Figure 5 shows that both SlickCache and SlickCache-GC significantly increase the throughput by up to 125 times, which is due to the significant hit ratio increase (by up to 8.2 times) enabled by the much larger cache space.

5.4 Cache Server Performance

In this section, we focus on the cache server and study each design component individually, as well as important parameter settings. All requests are directly sent to the cache server without involving the backend database to fully load the cache system for testing.

• **Tier-1: In-memory Indexing**. The design of Cascade Mapping leverages the access skewness of key-value workloads. Thus the size of its in-memory portion (Tier 1) has a strong effect on the cache performance. We test a variety of in-memory mapping table sizes (4% to 20% of the "all-in-memory" mapping table size as in Fatcache). In this test, we synthesize a working set of 100 million key-value pairs with a SET/GET ratio of 1:30, following the three popularity distributions.

Figure 6 shows the cache hit ratio with the increase of the portion of mapping entries in memory. The results show diminishing gains. By increasing the in-memory portion from 4% to 10%, the cache hit ratio significantly increases by 46.1, 42, and 13.5 percentage points (p.p.) for Hotspot, Zipfian, and Normal distributions, respectively. Further increasing the Tier-1 portion from 10% to 20% only receives a small (3.4-10.5 p.p.) hit ratio increase but doubles the memory cost. This trend is less noticeable with Normal distribution due to the weaker temporal locality. As our goal is to balance between the overall performance and memory consumption, caching 10% of the workload's mappings in memory works well in general.

The Tier-1 cache replacement algorithm identifies the cold mappings for demotion. Figure 7 compares the performance of three algorithms, FIFO, LRU, and CLOCK, for the Zipfian distribution workload. As an approximation of the LRU algorithm, CLOCK achieves nearly identical performance but is more memory efficient (only 1 bit per mapping entry). FIFO performs the worst, as expected, which also explains the suboptimal performance of the FIFO based slab management in the stock Fatcache. Our enhanced GC addresses this issue effectively as shown in Section 5.3.

• **Tier-2: Direct In-flash Indexing**. The second tier manages an array of direct mapping blocks, which are stored on flash in the FIFO order. To speed up the lookup, Tier 2 uses parallel I/Os to aggressively load a batch of mapping blocks for a one-to-one comparison in memory. A proper setting of parallelism degree and block size is important for both throughput and latency.

We first test a variety of block sizes (4 KB to 128 KB) and parallelism degrees (1 to 16 jobs) to study their effect on performance. As this micro-benchmark is for testing the Tier-2 structure, we only use the demotion buffer of the first tier to accumulate mapping entries into an in-flash block and we directly insert the key-value if not found in Tier-2, which in effect bypasses the two other tiers. This test uses a SET/GET ratio of 1:30 with Zipfian distribution.

Figure 8 and Figure 9 show the throughput and the latency with different parameter combinations. We can see that with 8 parallel I/Os, the overall throughput reaches nearly the peak (about 7,100 ops/sec), and further increasing parallelism degree only brings limited throughput benefit but quickly increases the latency, since the device is saturated. Block size affects the latency significantly. With 8 parallel I/Os, the average latency is 0.29 ms for 16-KB blocks, only about half of using 32-KB blocks. It is because as the device bandwidth reaches the peak, increasing the block size would proportionally take more time to transfer the data and to complete the one-to-one comparison in memory, resulting in a longer response time. Thus, using 8 parallel I/Os with 16-KB blocks on our platform (4 Cores, 8 Channels) generally reaches a balance between fully using the bandwidth and retaining a low latency.

The Tier-2 size (i.e., the total number of direct mapping blocks) is important. Maintaining a large array of blocks allows more mapping entries to be scanned in a high-bandwidth manner, rather than walking a long linked list as in Tier 3, but more batches of I/Os could be involved without contributing much to hit ratio. Figure 10 shows the I/O efficiency (i.e., the number of hits per batch). We enable Tier 1 and use 8 parallel I/Os with 16-KB blocks in this test.

We can see that increasing from 8 blocks (one batch) to 64 blocks (8 batches), the efficiency stays stable, about 0.8 hits/batch, meaning that one batch of parallel I/Os can find 0.8 mapping. Compared to Tier 3, which needs 8 to 9 flash I/Os for a mapping query, this efficiency is satisfactory. When it exceeds 64 blocks, the efficiency decreases. Thus we set an array of 64 mapping blocks in Tier 2.

• **Tier-3: Dual-mode Mapping**. The dual-mode mapping in Tier 3 is a hybrid indexing structure with a narrow active table and a wide inactive table. The shape of the two hash tables are determined by the list length of the active hash table and the inactive/active ratio, both affecting the lookup performance.

We vary the number of mapping blocks in the linked list of active hash table bucket from 4 blocks to 1,024 blocks, and the inactive/active ratio from 128 to 1,024. Since we are only interested in Tier-3 mappings, we set the Tier-1 table size to 16 KB and the Tier-2 mapping block to 1, so both tiers simply pass the mapping data to Tier 3. To illustrate the effect of compaction, we configure a mixed read-write workload (SET/GET ratio of 1:1) and generate 100 million requests with Zipfian distribution. We use a 128-KB block size for I/O efficiency, as described in Section 4.1.3.

Figure 11 shows that increasing the length of active hash table list from 4 blocks to 64 blocks effectively improves the throughput by 2.9 times (for ratio 1,024). With a short list, the compaction happens frequently and interferes incoming requests, but less indirect mapping blocks need to be scanned, meaning less I/Os. A long list, on the contrary, involves more I/Os but triggers compaction less often. This effect is evident in the curves. Setting the active list length to 64 blocks reaches the peak performance. We find this setting also works well in read-intensive workloads (SET/GET ratio of 1:30). We choose an inactive/active ratio of 1,024, which creates a wider hash table compared to using a lower ratio (512) and demands a reasonable amount of temporary buffer (at most 128 MB).

• **Garbage Collection (GC)**. The GC process reclaims slabs by adaptively switching between two eviction policies, the space-oriented eviction and the hit ratio-oriented eviction. The former is fast but disregards the locality; the latter is slower but retains hot items in cache. The setting of low and high watermarks determines which policy is used when the system runs low on free slabs.

	GC Watermarks: Low-High (%)				
	1-4	2-5	3-8	5-10	10-15
Hit Ratio (%)	89.0	89.4	87.1	82.5	76.6
Throughput (k ops/s)	137	141	139	133	127
Latency (μs)	592	543	538	526	517

Table 1: Garbage Collection Watermark Settings.

To study the effect of the watermarks, we configure a write-intensive workload (SET/GET ratio of 1:1) with Zipfian distribution. We vary the watermark settings as shown in Table 1. As we raise the watermarks, the system tends to quickly clean slabs, including hot key-values. Although it reduces the latency, the hit ratio decreases substantially (from 89% to 76.6%), affecting throughput. If the backend database is involved, a significant performance loss would happen due to the high miss penalty. Thus we choose to set the low and high watermarks as 2% and 5%, respectively.

• **Put All Together**. Finally, we run a set of experiments to test the cache server with the above-said parameter settings. Figure 12 and Figure 13 show the throughput and latency results, respectively.

We can see that the performance of SlickCache is comparable to Fatcache in most tests, even though it uses only 10% of the memory. The worst case scenario comes in Normal distribution with 100% GET operations, in which SlickCache and SlickCache-GC are about 23% slower than Fatcache due to the weaker locality. In Hotspot and Zipfian workloads, SlickCache-GC shows a similar throughput to Fatcache, if not better. Noticeably, with a write-intensive workload (SET/GET ratio of 75:25), SlickCache-GC outperforms Fatcache by 8.4% in terms of throughput for Zipfian distribution. For write-intensive workloads, GC runs frequently. SlickCache-GC excels due to the more effective GC policy and the higher hit ratio. In comparison, SlickCache performs slightly worse (6.6%) than Fatcache due to the extra flash I/Os. In the worst case, SlickCache and SlickCache-GC take about 237 μs and 241 μs longer than Fatcache to process a query, as shown in Figure 13c. When the database is involved, such a small loss is almost negligible, not to mention the significant hit ratio benefit (see Section 5.3).

	Tier 1	**Tier 2**	**Tier 3**
Hotspot	95.2%	4.3% (2.41)	0.5% (8.22)
Zipfian	85.8%	11.9% (3.57)	2.3% (8.86)
Normal	71.7%	22.1% (6.80)	6.2% (9.72)

Table 2: Hit Ratio and I/Os for Mapping Lookups. *The average number of I/Os per mapping lookup is shown in parentheses.*

Table 2 shows the percentage of mapping lookups served by each tier. The results confirm that the majority of requests can be satisfied in the fast Tier 1 and Tier 2. Only a small portion (0.5% to 6.2%) of the mapping lookups reaches the slower Tier 3, each incurring 8.22 to 9.72 flash I/Os on average.

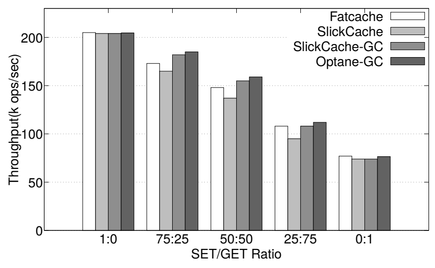
5.5 Comparison with Other Solutions

We also compare SlickCache with two alternative solutions, using an expensive but fast NVM-based SSD and using a simple swapping-based solution. Similar to the previous test, the backend database is not involved in this set of experiments.

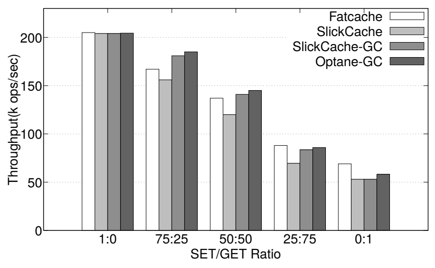
• **SlickCache on NVM SSDs**. Besides NAND flash SSDs, the recently available Intel Optane SSD [5] can also be used as a high-speed storage media to maintain the mapping entries. The Optane SSD is built on 3D XPoint Non-volatile Memory (NVM). It is a block device but provides high bandwidth (2.5 GB/sec) and low latency (10 μs). Such a high speed makes it a potential storage media for holding the mapping structures.

In this set of experiments, we move the Tier-2 and Tier-3 mappings to a 280-GB Intel Optane 900P SSD. The key-value data are still stored in the flash SSD for a fair comparison. Optane-GC in Figure 12 and Figure 13 represents SlickCache-GC with all the mappings stored on the Optane SSD. Interestingly, despite the 6 times higher throughput and 5 times lower latency compared to the flash SSD, the performance improvement by running SlickCache-GC on the Optane SSD is rather limited (3.4% to 16.6%). This is because Cascade Mapping successfully retains the hot mapping entries in memory, which absorbs the most queries. Only a small portion of the mapping queries needs to access the SSD.

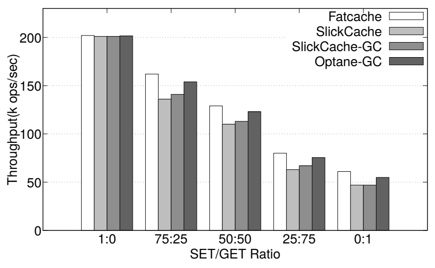
• **Flash as a Swap Device**. As discussed in Section 3.2, a straightforward solution for addressing the memory challenge is to use a flash SSD as the swap device. In this set of experiments, we run the stock Fatcache with mixed read and write operations (SET/GET ratio



(a) Hotspot

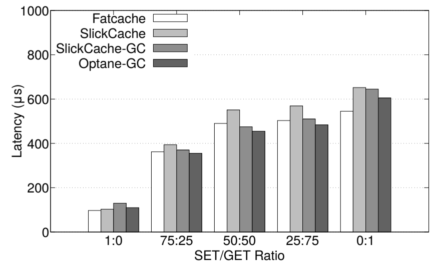


(b) Zipfian

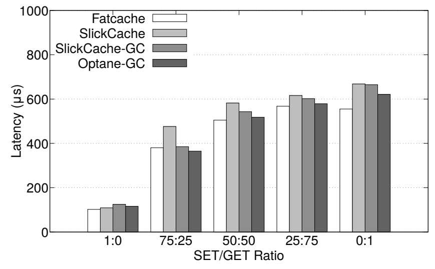


(c) Normal

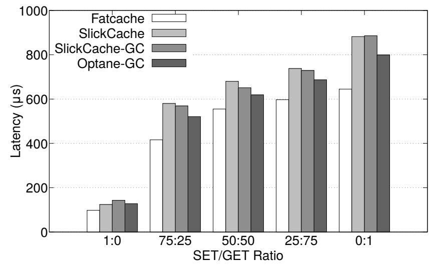
Figure 12: Performance of Cache Server (Throughput). *Fatcache stores the mapping entries all in memory. SlickCache and SlickCache-GC store the off-memory mapping entries in the flash SSD. Optane-GC stores the off-memory mapping entries in the Optane SSD.*



(a) Hotspot



(b) Zipfian



(c) Normal

Figure 13: Performance of Cache Server (Latency). *Fatcache stores the mapping entries all in memory. SlickCache and SlickCache-GC store the off-memory mapping entries in the flash SSD. Optane-GC stores the off-memory mapping entries in the Optane SSD.*

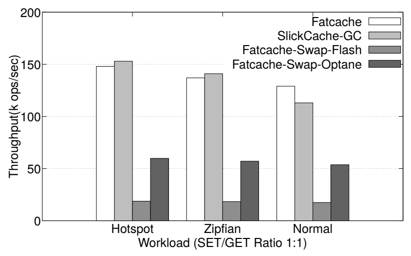


Figure 14: SlickCache vs. Swapping.

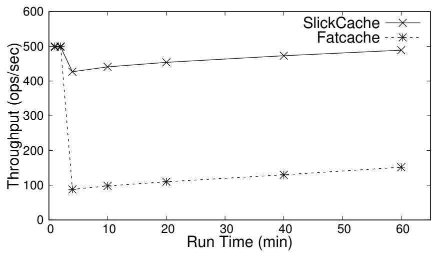


Figure 15: Crash Recovery.

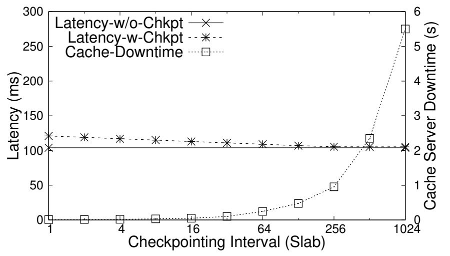


Figure 16: Checkpointing Frequency.

of 1:1). We configure to use the same amount of DRAM memory as SlickCache, and use the flash and the Optane SSDs as the swap devices. Fatcache and SlickCache-GC data in Figure 12 are also shown for reference. As shown in Figure 14, we can find that using flash as a swap device results in a poor performance (17.5-18.7 kops/sec), compared to Fatcache (129-148 kops/sec). Swapping on Optane is much faster (53.7-59.8 kops/sec) thanks to the advanced NVM technology, but the overall throughput is still 58.3-59.6% worse than the all-in-memory Fatcache and 52.5-60.9% worse than SlickCache-GC. It shows that simply using the swapping mechanism cannot satisfy the performance requirement. A well-designed structure, such as Cascade Mapping, is needed to exploit the SSD performance.

5.6 Crash Recovery

In SlickCache, the dirty mapping entries in memory are periodically saved in flash as a checkpoint. Upon crash recovery, the in-memory mapping structure is reconstructed by loading the latest checkpoint and then applying changes made after that. Thus, the frequency of checkpointing plays an important role and affects performance.

We simulate a sudden crash on the cache server by killing the cache manager process, and restart it immediately after the crash. We keep a working set of 300 GB in MongoDB, the cache server is configured to hold 10% of key-value data for the Hotspot workload, as in Section 5.3. Upon crash, SlickCache reacts with its recovery policy (see Section 4.4), while Fatcache loses the entire cache and needs to reload all the data from the backend database.

Figure 15 shows the throughput over time with the Hotspot workload. The crash happens 4 minutes after the cache is warmed up. We can see that SlickCache is still capable of serving requests at a relatively high throughput after the crash. A 14.2% performance drop was observed due to the loss of clean mappings, but it recovers to a stable level quickly. In contrast, Fatcache suffers an 82.4% throughput decrease, since the entire cache is lost and needs to be warmed up from scratch again, taking hours to bring the throughput back to the normal level.

Figure 16 shows the effect of the frequency of checkpointing. We can see that as the checkpointing interval increases from every 1 slab to every 256 slabs, its impact to the normal latencies is reduced to almost negligible, and the crash recovery time increases to nearly 1 second, since more slabs need to be scanned. Considering that system crash rarely happens and the normal performance is more important, we choose to checkpoint every 256 slabs.

5.7 System Resource Usage

Cascade Mapping is a deeply optimized key-value caching mapping scheme with a small memory demand. In this section, we analyze the system resource usage in our current prototype.

• **Flash Usage**. Several structures are needed in flash. (1) Checkpoint for Tier 1 takes at most 176 MB for a checkpointing interval of 256 slabs. (2) On Tier 2, for each partition, 64 mapping blocks with the size of 16 KB each are kept on flash. (3) On Tier 3, each mapping uses a compact 16-byte entry (8 bytes for hash prefix, 4 bytes for Slab_ID, and 4 bytes for Offset) in flash. Assuming the average object size is 300 bytes, it adds about 6% overhead on flash storage. The entry size can be doubled to 32 bytes to support a larger capacity at the cost of about 12% overhead. As the price and power consumption of flash is much cheaper than DRAM, such an extra flash space overhead is rather small.

• **Memory Usage**. To manage the data in flash, and to keep track of the hot mapping entries in memory, memory overhead in SlickCache is as follows: 1 bit for dirty/clean denotation; 1 bit for reference bit; a 2-byte clock hand pointer for each in-memory hash table list; a 2-byte pointer for each Tier-2 mapping block; a 128-KB dedicated buffer for each Tier-3 active hash table list, and 1,024 dynamically allocated 128-KB buffers for Tier-3 compaction.

Scheme	SET	GET	SET/GET (1:1)
Fatcache	45.5%	21.9%	38.7%
SlickCache	50.5%	22.7%	40.8%
SlickCache-GC	52.5%	23.1%	42.5%

Table 3: A Sample of CPU Usage.

• **CPU Usage**. Our prototype adds a shared pool of 64 threads for parallel I/Os in Tier 2 and a compaction thread in Tier 3, which incur extra computational overhead. Table 3 shows a sample of the average CPU usage in the cache server experiments (Section 5.4). We can find that the CPU usage increase in SlickCache is trivial. The worst case is observed with 100% SET, in which SlickCache-GC uses an extra of 7% CPU resource over Fatcache.

6 RELATED WORK

Memory efficiency has been studied in the context of flash storage and key-value systems in prior work. This section discusses earlier studies that are most related to this paper.

Flash device firmware manages a logical-to-physical mapping table as part of the FTL. As the flash capacity increases, the limited on-board RAM space becomes a technical challenge at the device level. For example, DFTL [34] selectively stores popular mappings in RAM to support page-level mapping. GeckoFTL [28] introduces a structure, called Logarithmic Gecko, to reduce the integrated RAM requirement and to speed up recovery. Nameless Writes [50] removes the need for the large and costly indirection table by exposing a new interface to the host system. SlickCache aims to improve memory efficiency for flash-based key-value caches at the application level, being orthogonal to these prior research.

Flash memory has been recently used in key-value caches. Besides Facebook's McDipper [8] and Twitter's Fatcache [3], which store key-value data completely in flash, Blott et al. has proposed an FPGA based hybrid solution to store key-value data in both DRAM and flash SSD according to the value size [19]. In contrast, SlickCache focuses on managing the mapping information.

Memory efficiency has been considered in prior research on key-value systems. For example, MemC3 [32] improves the memory efficiency and throughput for Memcached with CLOCK and Concurrent Cuckoo hashing techniques. NVMKV [42] leverages native FTL capabilities for processing key-value requests and reducing RAM overhead. DIDACache [48] removes the intermediate mapping layer by directly controlling the flash hardware on Open-Channel SSDs. SlickCache optimizes flash-based key-value caches and uses off-the-shelf SSDs, requiring no special hardware.

Several prior works particularly minimize the memory usage in key-value systems. For example, FAWN [16] reduces memory usage by using only a fragment of each hash key at the cost of extra flash reads for verification. BufferHash [15] holds a hash table in memory as a buffer and flushes it to flash if being filled up, but a lookup possibly needs to load and search in multiple in-flash hash tables. BloomStore [40] maintains a chain of Bloom filters to keep the indexing structure all in flash, demanding a sequence of flash I/Os to locate a key-value pair. SkimpyStash [29] aggressively reduces the memory footprint size by keeping a small in-memory hash table, of which each slot points to a chain of key-value records in flash with a Bloom filter. Thus a query also involves a chain of flash reads. SILT [39] optimizes the memory usage by simultaneously maintaining three types of stores with different memory costs, which demands a substantial flash space and involves frequent conversion and merge operations. In contrast, SlickCache does not require to compact any value data. Compared to these earlier work, SlickCache, as a cache solution, is more lightweight and uses a hierarchical mapping structure to optimize the memory usage while still retaining a low overhead.

Flash has also been used in open-source key-value databases. For example, RocksDB [11] is a widely used key-value database based on LevelDB [7] and is optimized for flash SSDs. Facebook recently presents another system, called MyNVM [30], which integrates NVM to reduce the DRAM footprint of MyRocks, an MySQL database built on top of RocksDB. Unlike key-value databases, which need to persistently store data, our design is optimized for key-value caching, which has different requirements and unique properties, e.g., if needed, even valid data can be safely dropped.

7 CONCLUSION

Flash-based key-value cache is essential in today's web applications and Internet services. However, the all-in-memory approach adopted in the current systems is unscalable due to the limited DRAM memory capacity. In this paper, we present a hierarchical mapping structure, called Cascade Mapping, to efficiently manage the mapping information in a combination of DRAM and flash SSD. We have implemented a prototype based on Twitter's Fatcache, called SlickCache. Our experimental results show that SlickCache can substantially reduce the memory needs while still achieving satisfactory performance.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive feedback and insightful comments, which have substantially improved this paper. This work was supported in part by the U.S. National Science Foundation under Grants CCF-1453705 and CCF-1629291.

REFERENCES

[1] Bloom Filter. https://en.wikipedia.org/wiki/Bloom_filter.
[2] Done in 60 Seconds. https://www.statista.com/chart/13157/what-happens-in-the-digitalized-world-in-one-minute-in-2017/.
[3] Fatcache. https://github.com/twitter/fatcache.
[4] Fatcache-Async. https://github.com/polyuszy/Fatcache-Async-2017.
[5] Intel Optane Technology. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html.
[6] Kvtracer. https://github.com/ryanphuang/imembench/.
[7] LevelDB. https://leveldb.org.
[8] McDipper: A Key-value Cache for Flash Storage. https://www.facebook.com/notes/facebook-engineering/mcdipper-a-key-value-cache-for-flash-storage/10151347090423920/.
[9] Memcached. https://memcached.org.
[10] Redis. https://redis.io.
[11] RocksDB. http://rocksdb.org.
[12] Things that Happen on Internet Every 60 Seconds. https://www.go-globe.com/blog/things-that-happen-every-60-seconds/.
[13] Yahoo! Cloud Serving Benchmark. https://github.com/brianfrankcooper/YCSB.
[14] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design Tradeoffs for SSD Performance. In *Proceedings of the 2008 USENIX Annual Technical Conference (USENIX'08)*, Boston, MA, June 22-27 2008.
[15] A. Anand, C. Muthukrishnan, S. Kappes, A. Akella, and S. Nath. Cheap and Large CAMs for High Performance Data-intensive Networked Systems. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation (NSDI'10)*, San Jose, CA, April 28-30 2010.
[16] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP'09)*, Big Sky, Montana, USA, October 11-14 2009.
[17] Apple. Mac Mini (Late 2012 and Later), iMac (Late 2012 and Later): About Fusion Drive. https://support.apple.com/en-us/HT202574, Mar 22 2016.
[18] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of A Large-scale Key-value Store. In *Proceedings of 2012 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'12)*, London, UK, June 11-15 2012.
[19] M. Blott, L. Liu, K. Karras, and K. Vissers. Scaling Out to a Single-node 80Gbps Memcached Server with 40 Terabytes of Memory. In *Proceedings of the 7th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage'15)*, Santa Clara, CA, July 6-7 2015.
[20] J. Bonwick. The Slab Allocator: An Object-Caching Kernel Memory Allocator. In *Proceedings of 1994 USENIX Summer Technical Conference (USENIX'94)*, pages 87–98, Boston, MA, June 6-10 1994.
[21] D. Carra and P. Michiardi. Memory Partitioning in Memcached: An Experimental Performance Analysis. In *Proceedings of 2014 IEEE International Conference on Communications (ICC'14)*, Sydney, Australia, June 10-14 2014.
[22] F. Chen, B. Hou, and R. Lee. Internal Parallelism of Flash Memory-based Solid-State Drives. *ACM Transactions on Storage*, 12(3):13:1–13:39, May 2016.
[23] F. Chen, S. Jiang, and X. Zhang. SmartSaver: Turning Flash Drive into a Disk Energy Saver for Mobile Computers. In *Proceedings of 2006 International Symposium on Low Power Electronics and Design (ISLPED'06)*, Tegernsee, Germany, October 4-6 2006.
[24] F. Chen, D. Koufaty, and X. Zhang. Hystor: Making the Best Use of Solid State Drives in High Performance Storage Systems. In *Proceedings of the 25th ACM International Conference on Supercomputing (ICS'11)*, Tucson, AZ, May 31–June 4 2011.
[25] F. Chen, D. A. Koufaty, and X. Zhang. Understanding Intrinsic Characteristics and System Implications of Flash Memory based Solid State Drives. In *Proceedings of 2009 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS/Performance'09)*, Seattle, WA, June 15-19 2009.
[26] F. Chen, R. Lee, and X. Zhang. Essential Roles of Exploiting Internal Parallelism of Flash Memory based Solid State Drives in High-speed Data Processing. In *Proceedings of the 17th International Symposium on High Performance Computer Architecture (HPCA'11)*, San Antonio, Texas, February 12-16 2011.
[27] F. Corbató. *A Paging Experiment with the Multics System*. Project MAC (Massachusetts Institute of Technology). 1968.
[28] N. Dayan, P. Bonnet, and S. Idreos. GeckoFTL: Scalable Flash Translation Techniques For Very Large Flash Devices. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD'16)*, pages 327–342, San Francisco, CA, June 26-July 1 2016.
[29] B. Debnath, S. Sengupta, and J. Li. SkimpyStash: RAM Space Skimpy Key-value Store on Flash-based Storage. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD'11)*, pages 25–36, Athens, Greece, June 7-11 2011.
[30] A. Eisenman, D. Gardner, I. AbdelRahman, J. Axboe, S. Dong, K. Hazelwood, C. Petersen, A. Cidon, and S. Katti. Reducing DRAM Footprint with NVM in Facebook. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys'18)*, pages 42:1–42:13, Porto, Portugal, April 23-26 2018.
[31] M. Eshel, R. Haskin, D. Hildebrand, M. Naik, F. Schmuck, and R. Tewari. Panache: A Parallel File System Cache for Global File Access. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST'10)*, San Jose, CA, 2010.
[32] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI'13)*, pages 371–384, Lombard, IL, April 2-5 2013.
[33] B. Fitzpatrick. Distributed Caching with Memcached. *Linux Journal*, 2004(124):5–5, Aug. 2004.
[34] A. Gupta, Y. Kim, and B. Urgaonkar. DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'09)*, pages 229–240, Washington, DC, USA, March 7-11 2009.
[35] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*, chapter 1, pages 17–18. Morgan Kaufmann, 5th edition.
[36] A. Klimovic, H. Litz, and C. Kozyrakis. ReFlex: Remote Flash ≈ Local Flash. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*, pages 345–359, Xi'an, China, April 8-12 2017.
[37] J. Lee, S. Park, M. Ryu, and S. Kang. Performance Evaluation of the SSD-based Swap System for Big Data Processing. In *Proceedings of 2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom'14)*, Beijing, China, September 24-26 2014.
[38] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim. A Case for Flash Memory SSD in Enterprise Database Applications. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD'08)*, pages 1075–1086, Vancouver, Canada, June 9-12 2008.
[39] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A Memory-efficient, High-performance Key-value Store. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*, pages 1–13, Cascais, Portugal, October 23-26 2011.
[40] G. Lu, Y. J. Nam, and D. H. Du. BloomStore: Bloom-Filter based Memory-efficient Key-Value Store for Indexing of Data Deduplication on Flash. In *Proceedings of IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST'12)*, San Diego, CA, April 16-20 2012.
[41] T. Luo, S. Ma, R. Lee, X. Zhang, D. Liu, and L. Zhou. S-CAVE: Effective SSD Caching to Improve Virtual Machine Storage Performance. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT'13)*, Edinburgh, Scotland, September 7-11 2013.
[42] L. Marmol, S. Sundararaman, N. Talagala, and R. Rangaswami. NVMKV: A Scalable, Lightweight, FTL-aware Key-value Store. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC'15)*, pages 207–219, Santa Clara, CA, July 8-10 2015.
[43] J. Matthews, S. Trika, D. Hensgen, R. Coulson, and K. Grimsrud. Intel Turbo Memory: Nonvolatile Disk Caches in the Storage Hierarchy of Mainstream Computer Systems. *ACM Transactions on Storage*, 4(2):4:1–4:24, May 2008.
[44] I. Moulster. SuperFetch, ReadyBoost and ReadyDrive: Some New Feature Names for You. https://blogs.msdn.microsoft.com/ianm/2006/04/06/superfetch-readyboost-and-readydrive-some-new-feature-names-for-you/, April 6 2006.
[45] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI'13)*, pages 385–398, Lombard, IL, April 2-5 2013.
[46] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Analysis and Evolution of Journaling File Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'05)*, Anaheim, CA, April 10-15 2005.
[47] M. Saxena, M. M. Swift, and Y. Zhang. FlashTier: A Lightweight, Consistent and Durable Storage Cache. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys'12)*, Bern, Switzerland, April 10-13 2012.
[48] Z. Shen, F. Chen, Y. Jia, and Z. Shao. DIDACache: A Deep Integration of Device and Application for Flash-based Key-value Caching. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*, Santa Clara, CA, February 27-March 2 2017.
[49] X. Wu, L. Zhang, Y. Wang, Y. Ren, M. Hack, and S. Jiang. zExpander: A Key-Value Cache with Both High Performance and Fewer Misses. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys'16)*, London, UK, April 18-21 2016.
[50] Y. Zhang, L. P. Arulraj, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. De-indirection for Flash-based SSDs with Nameless Writes. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*, San Jose, CA, February 14-17 2012.
[51] Y. Zhang, G. Soundararajan, M. W. Storer, L. N. Bairavasundaram, S. Subbiah, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Warming Up Storage-level Caches with Bonfire. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*, pages 59–72, San Jose, CA, February 12-15 2013.