SlimCache: An Efficient Data Compression Scheme for Flash-based Key-value Caching

YICHEN JIA, Louisiana State University
ZILI SHAO, The Chinese University of Hong Kong
FENG CHEN, Louisiana State University

Flash-based key-value caching is becoming popular in data centers for providing high-speed key-value services. These systems adopt slab-based space management on flash and provide a low-cost solution for key-value caching. However, optimizing cache efficiency for flash-based key-value cache systems is highly challenging, due to the huge number of key-value items and the unique technical constraints of flash devices. In this article, we present a dynamic on-line compression scheme, called *SlimCache*, to improve the cache hit ratio by virtually expanding the usable cache space through data compression. We have investigated the effect of compression granularity to achieve a balance between compression ratio and speed, and we leveraged the unique workload characteristics in key-value systems to efficiently identify and separate hot and cold data. To dynamically adapt to workload changes during runtime, we have designed an adaptive hot/cold area partitioning method based on a cost model. To avoid unnecessary compression, SlimCache also estimates data compressibility to determine whether the data are suitable for compression or not. We have implemented a prototype based on Twitter's Fatcache. Our experimental results show that SlimCache can accommodate more key-value items in flash by up to 223.4%, effectively increasing throughput and reducing average latency by up to 380.1% and 80.7%, respectively.

CCS Concepts: • **Information systems** → **Data compression**; **Flash memory**; **Key-value stores**;

Additional Key Words and Phrases: Flash memory, SSD, key-value caching, data compression

ACM Reference format:
Yichen Jia, Zili Shao, and Feng Chen. 2020. SlimCache: An Efficient Data Compression Scheme for Flash-based Key-value Caching. *ACM Trans. Storage* 16, 2, Article 14 (June 2020), 34 pages.
https://doi.org/10.1145/3383124

This is a revised version. A preliminary version of this work was published in the Proceedings of the 26th IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'18).
The work described in this article was partially supported by the U.S. National Science Foundation under Grants No. CCF-1453705, No. CCF-1629291, and No. CCF-1910958, the Louisiana Board of Regents under Grant No. LEQSF(2014-17)-RD-A-01, the Research Grants Council of the Hong Kong Special Administrative Region, China under Grants No. GRF 15273616, No. GRF 15206617, and No. GRF 15224918, and Direct Grant for Research, The Chinese University of Hong Kong (Project No. 4055096).
Authors' addresses: Y. Jia, Department of Computer Science and Engineering, Louisiana State University, Baton Rouge, Louisiana, USA; email: yjia@csc.lsu.edu; Z. Shao, Department of Computer Science and Engineering, The Chinese University of Hong Kong, Shatin, NT, Hong Kong; email: shao@cse.cuhk.edu.hk; F. Chen, Department of Computer Science and Engineering, Louisiana State University, Baton Rouge, Louisiana, USA; email: fchen@csc.lsu.edu.

1553-3077/2020/06-ART14 $15.00
https://doi.org/10.1145/3383124

1 INTRODUCTION

Today's data centers still heavily rely on hard disk drives (HDDs) as their main storage devices. To reduce the traffic of requests to backend data stores, in-memory key-value cache systems, such as Memcached [58], become popular in data centers for serving various applications [23, 78]. Although memory-based key-value caches can eliminate a large amount of key-value data retrievals (e.g., "User ID" and "User Name") from the back-end data stores, they also raise concerns on cost and power consumption issues in a large-scale deployment. As an alternative solution, flash-based key-value cache systems recently have attracted an increasingly high interest in industry. For example, Facebook has deployed a key-value cache system based on flash, called McDipper [28], as a replacement of the expensive Memcached servers. Twitter has a similar key-value cache solution, called Fatcache [78].

1.1 Motivations

The traditional focus on improving the caching efficiency is to develop sophisticated cache replacement algorithms [39, 57]. Unfortunately, it is highly challenging in the scenario of flash-based key-value caching. This is for two reasons.

First, compared to memory-based key-value cache, such as Memcached, flash-based key-value caches are usually 10–100 times larger. As key-value items are typically small (e.g., tens to hundreds of bytes), a flash-based key-value cache often needs to maintain billions of key-value items, or even more. Tracking such a huge number of small items in cache management would result in an unaffordable overhead. Also, many advanced cache replacement algorithms, such as ARC [57] and CLOCK-Pro [39], need to maintain a complex data structure and a deep access history (e.g., information about evicted data), making the overhead even more pronounced. Therefore, a complex caching scheme is practically infeasible for flash-based key-value caches.

Second, unlike DRAM, flash memories have several unique technical constraints, such as the well-known "no in-place overwrite" and "sequential-only writes" requirements [3, 12]. As such, flash devices generally favor large, sequential, log-like writes rather than small, random writes. Consequently, flash-based key-value caches do not directly "replace" small key-value items in place as Memcached does. Instead, key-value data are organized and replaced in large coarse-grained chunks, relying on Garbage Collection (GC) to recycle the space occupied by obsolete or deleted data. This unfortunately reduces the usable cache space and affects the caching efficiency.

For the above two reasons, it is difficult to solely rely on developing a complicated, fine-grained cache replacement algorithm to improve the cache hit ratio for key-value caching in flash. In fact, real-world flash-based key-value cache systems often adopt a simple, coarse-grained caching scheme. For example, Twitter's Fatcache uses a First-In-First-Out (FIFO) policy to manage its cache in a large granularity of slabs (a group of key-value items) [78]. Such a design, we should note, is an unwillingly-made but necessary compromise to fit the needs for caching many small key-value items in flash. This article seeks an *alternative* solution to improve the cache hit ratio. This solution, interestingly, is often ignored in practice—increasing the *effective* cache size.

The key idea is that for a given cache capacity, the data could be compressed to save space, which would "virtually" enlarge the usable cache space and allow us to accommodate more data in the flash cache, in turn increasing the hit ratio.

In fact, on-line compression fits flash devices very well. Figure 1 shows the percentage of I/O and computation time for compressing and decompressing random data in different request sizes. The figure illustrates that for read requests, the decompression overhead only contributes a relatively small portion of the total time, less than 2% for requests smaller than 64KB. For write requests, the compression operations are more computationally expensive, contributing for about 10%–30% of the overall time, but it is still at the same order of magnitude compared to an I/O access to

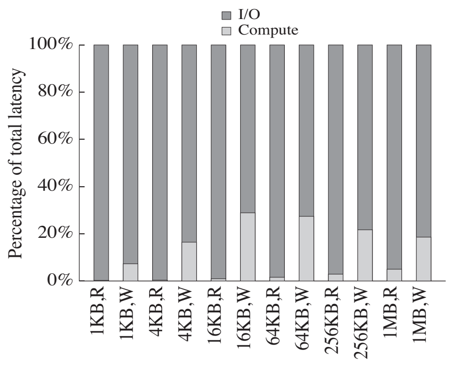


Fig. 1. I/O time vs. computation time.

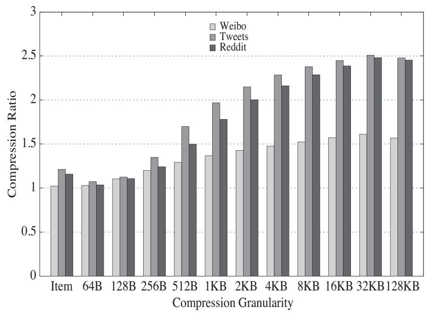


Fig. 2. Compression ratio vs. granularity.

flash. Compared to schemes compressing data in memory, such as zExpander [84], the relative computing overhead accounts for an even smaller percentage, indicating that it would be feasible to apply on-line compression in flash-based caches.

1.2 Challenges and Critical Issues

Though promising, efficiently incorporating on-line compression in flash-based key-value cache systems is non-trivial. Several critical issues must be addressed.

First, various compression algorithms have significantly different compression efficiency and computational overhead [75, 30, 52]. Lightweight algorithms, such as `lz4` [52] and `snappy` [30], are fast, but only provide a moderate *Compression Ratio*, which is calculated as $\frac{uncompressed}{compressed}$; heavyweight schemes, such as the deflate algorithm used in `gzip` [29] and `zlib` [66], can provide better compression efficacy, but are relatively slow and would incur higher overhead. We need to select a proper algorithm.

Second, compression efficiency is highly dependent on the compression unit size. A small unit size suffers from a *low compression ratio* problem, while aggressively using an oversized compression unit could incur a severe *read amplification* problem (i.e., read more than needed). Figure 2 shows the average compression ratio of three datasets (Weibo, Tweet, Reddit) with different container sizes. We can see that these three datasets are all compressible, as expected, and a larger compression granularity generally results in a higher compression ratio. In contrast, compressing each key-value item individually or using a small compression granularity (e.g., smaller than 4 KB) cannot reduce the data size effectively. In this article, we will present an effective scheme, which considers the properties of flash devices, to pack small items into a proper-size container for bulk compression. This scheme allows us to achieve both high compression ratio and low amplification factor.

Third, certain data are unsuitable for compression, either because they are frequently accessed or simply incompressible, e.g., JPEG images. We need to quickly estimate the data compressibility and conditionally apply on-line compression to minimize the overhead.

Last but not least, we also need to be fully aware of the unique properties of flash devices. For example, flash devices generally favor large and sequential writes. The traditional log-based solution, though being able to avoid generating small and random writes, relies on an asynchronous Garbage Collection (GC) process, which would leave a large amount of obsolete data occupying the precious cache space and negatively affect the cache hit ratio.

All these issues must be well considered for an effective adoption of compression in flash-based key-value caching.

1.3 Our Solution: SlimCache

In this article, we present an adaptive on-line compression scheme for key-value caching in flash, called *SlimCache*. SlimCache identifies key-value items that are suitable for compression, and applies a compression and decompression algorithm at a proper granularity, thus expanding the effectively usable flash space for caching more data.

In SlimCache, the flash cache space is dynamically divided into two separate regions, a hot area and a cold area, to store frequently and infrequently accessed key-value items, respectively. Based on the highly skewed access patterns in key-value systems [5], the majority, infrequently accessed key-value items are cached in flash in a compressed format for the purpose of space saving. A small set of frequently accessed key-value items is cached in their original, uncompressed format to avoid the read amplification and decompression penalty. The partitioning is automatically adjusted based on runtime workloads. To create the desired large sequential write pattern on flash, the cache eviction process and the hot/cold data separation mechanism are integrated to minimize the cache space waste caused by data movement between the two areas.

To our best knowledge, SlimCache is the first work introducing compression into flash-based key-value caches. Our compression mechanism achieves both high performance and high hit ratio by restricting compressed unit within one flash page, dynamically identifying hot/cold data for caching without causing thrashing, and maintaining a large sequential access pattern on flash without wasting cache space. We have implemented a fully functional prototype based on Twitter's Fatcache [78]. Our experimental evaluations on an Intel 910 PCIe SSD have shown that SlimCache can accommodate more key-value items in the cache by up to 223.4%, effectively increasing throughput and reducing average latency by up to 380.1% and 80.7%, respectively. Such an improvement is essential for data-intensive applications in data centers.

The rest of this article is organized as follows. Section 2 gives the background. Section 3 introduces the design of SlimCache. Section 4 presents the experimental results. Section 5 discusses the limitations. The related work is presented in Section 6. The final section concludes this article.

2 BACKGROUND

In this section, we briefly introduce flash memory SSD and key-value cache systems. The difference between the flash-based key-value cache and the in-memory cache has motivated us to design an efficient flash-based solution.

Flash Memory. NAND flash is a type of EEPROM devices. Typically a flash memory chip is composed of several *planes*, and each plane has thousands of *blocks*. A block is further divided into multiple *pages*. NAND flash memory has three unique characteristics: (1) *Read/write speed disparity*. Typically, a flash page read is fast (e.g., 25–100 μs), but a write is slower (e.g., 200–900 μs). An erase must be conducted in blocks and is time-consuming (e.g., 1.5–3.5 ms). (2) *No in-place update*. A flash page cannot be overwritten once it is programmed. The entire block must be erased before writing any flash page. (3) *Sequential writes only*. The flash pages in a block must be written in a sequential manner. To address these issues, modern flash SSDs have the Flash Translation Layer (FTL) implemented in device firmware to manage the flash memory chips and to provide a generic Logical Block Address (LBA) interface as a disk drive. More details about flash memory and SSDs can be found in prior studies [3, 11–13].

Flash-based Key-value Caches. Similar to in-memory key-value caches, such as Memcached, flash-based key-value cache systems also adopt a slab-based space management scheme. Here, we take Twitter's Fatcache [78] as an example. In Fatcache, the flash space is divided into fixed-size *slabs*. Each slab is further divided into a group of *slots*, each of which stores a key-value item. The slots in a slab are of the same size. According to the slot size, slabs are classified into *slab classes*. For a given key-value pair, the smallest slot size that is able to accommodate the item and the related metadata is selected. A *hash table* is maintained in memory to index the key-value pairs stored in flash. A query operation (`GET`) searches the hash table to find the location of the corresponding key-value item on flash and then loads that slot into memory. An update operation (`SET`) writes the data to a new location and updates the mapping in the hash table accordingly. A delete operation (`DELETE`) only removes the mapping entry from the hash table. A *Garbage Collection* (GC) process is responsible for reclaiming the deleted and obsolete items later.

Although in-memory key-value caches and in-flash key-value caches are similar in their structures, they show several remarkable distinctions. (1) *I/O granularity.* The flash SSD is treated as a log-structured storage. Fatcache maintains a small memory buffer for each slab class. This in-memory slab buffer is used to accumulate small slot writes, and when it is filled up, the entire slab is flushed to flash, converting small random writes to large sequential writes. (2) *Data management granularity.* Unlike Memcached, which keeps an object-level LRU list, the capacity-triggered eviction procedure in Fatcache reclaims slabs based on a slab-level FIFO order.

3 DESIGN OF SLIMCACHE

To fully exploit compression opportunities for key-value caching in flash, we need to carefully consider three critical issues: the compression overhead, the data compressibility and the constraints of flash hardware.

- *Compression overhead.* Though simple, naïvely compressing all key-value data and decompressing them upon every access would incur high computational overhead. We particularly need to separate "hot" and "cold" data and selectively apply compression to them. So, we have

  Rule #1: Do not compress the hot data.

- *Compressibility.* Certain data types, such as multimedia data and encrypted strings, are already compressed or by nature incompressible. So, we need a simple mechanism to estimate the compressibility of the target data beforehand and to determine a proper compression granularity to maximize the potential compression efficiency and avoid ineffective compression. So, we have

  Rule #2: Do not compress the incompressible data.

- *Flash constraints.* Since the underlying flash memory favors large sequential writes, the compression mechanism should not generate extra small random ones. Considering that all the invalid or duplicated values have to wait for the garbage collection process to asynchronously reclaim their occupied valuable cache space, we need to avoid generating much obsolete data. So, we have

  Rule #3: Optimization should be flash-aware.

3.1 Overview

SlimCache is a comprehensive on-line compression scheme for flash-based key-value caching. As shown in Figure 3, SlimCache adopts a similar structure as Fatcache: A *hash table* is held in memory to manage the mapping from a hashed key to the corresponding value stored in flash, compressed or uncompressed; An *in-memory slab buffer* is maintained for each slab class, which batches up writes to flash and also serves as a temporary staging area for making the compression decision.

Unlike Fatcache, SlimCache has an adaptive *on-line compression layer*, which is responsible for selectively compressing, decompressing, and managing the flash space. In SlimCache, the flash

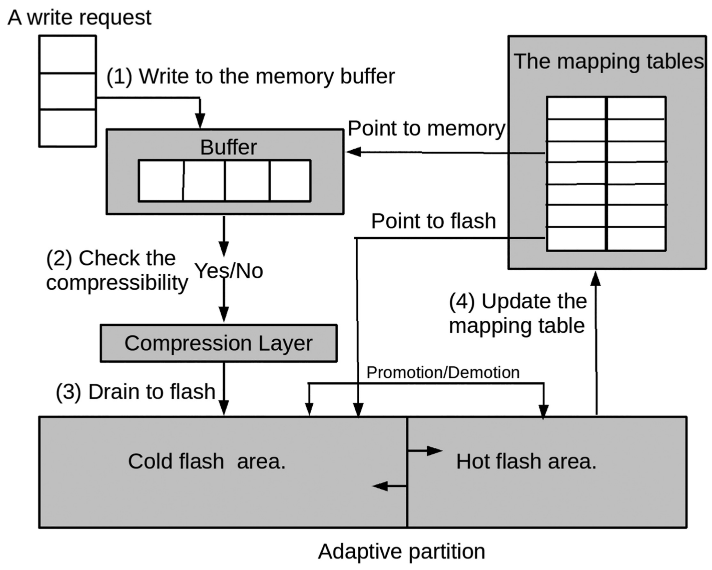


Fig. 3. An illustration of the SlimCache architecture.

space is segmented into two areas, a *hot area*, which stores the frequently accessed key-value data, and a *cold area*, which stores the relatively infrequently accessed data. Note that the key-value items in the hot area are stored in the original uncompressed format, which speeds up repeated accesses, while data in the cold area could be stored in either compressed and uncompressed format, depending on their compressibility. The division of the two regions is dynamically determined by the compression module at runtime. In the following, we will explain each of these components.

3.2 Slab Management

Similar to Fatcache, SlimCache adopts a slab-based space management: The flash space is sliced into *slabs*. A slab is further divided into equal-size *slots*, which is the basic storage unit. Slabs are virtually organized into multiple *slab classes*, according to their slot sizes. Differently, the slab slot in SlimCache can store compressed or uncompressed data. Thus, a slab could contain a mix of compressed slots and uncompressed slots. This design purposefully separates the slab management from the compression module and simplifies the management. A slab could be a *hot slab* or a *cold slab*, depending on its status. The hot slabs in aggregate virtually form the hot area, and similarly, the cold slabs together form the cold area. We will discuss the adaptive partitioning of the two areas later.

Slab Buffer. As flash devices favor large and sequential writes, a slab buffer is maintained to collect a full slab of key-value items in memory and write them to the flash in bulk. Upon an update (PUT), the item is first stored in the corresponding memory slab and completion is returned immediately. Once the in-memory slab becomes full, it is flushed to flash. Besides asynchronizing flash writes and organizing large sequential writes to flash, the buffer also serves as a staging area to collect compressible data.

Compression Layer. SlimCache has a thin compression layer to seamlessly integrate on-line compression into the I/O path. It works as follows. When the in-memory slab buffer is filled up, we iterate through the items in the slab buffer, and place the selected compressible ones into a *Compression Container* until full. Then an on-line compression algorithm is applied to the container, producing one single *Compressed Key-value Unit*, which represents a group of key-value items in the compressed format. Note that the compressed key-value unit is treated the same as other key-value items and placed back to the in-memory slab buffer, according to its slab class, and waiting for being flushed. In this process, the only difference is that the slot stores data in the compressed format. It is unnecessary for the slab I/O management to be aware of such a difference.

Mapping Structure. In SlimCache, each entry of the mapping table could represent two types of mappings. (1) *Key-to-uncompressed-value mapping*: An entry points to a slab slot that contains an original key-value item, which is identical to a regular flash-based key-value cache. (2) *Key-to-compressed-value mapping*: An entry points to the location of a slab slot that contains a compressed key-value unit, to which the key-value item belongs. That means, in SlimCache, multiple keys could map to the same physical location (i.e., a compressed slot in the slab). In the items stored on flash, we add a 1-bit attribute, called *compressed bit*, to differentiate the two situations. Upon a `GET` request, SlimCache first queries the mapping table, loads the corresponding slot from the flash, and depending on its status, returns the key-value item (if uncompressed) or decompresses the compressed key-value unit first and then returns the demanded key-value item.

The above design has two advantages. First, we maximize the reuse of the existing well-designed key-to-slab mapping structure. A compressed key-value unit is treated exactly the same as a regular key-value item—select the best-fit slab slot, append it to the slab, and update the mapping table. Second, it detaches the slab management from the on-line compression module, which is only responsible for deciding whether and how to compress a key-value item. This makes the management more flexible. For example, we can adaptively use different container sizes at run-time, while disregarding the details of storing and transferring data.

3.3 Compression Granularity

Deciding a proper compression container size is crucial, because the compression unit size directly impacts the compression ratio and the computational overhead. Two straightforward considerations are compressing data in slot granularity or compressing data in slab granularity. Here, we discuss the two options and explain our decision.

- *Option 1: Compressing data in slot granularity.* A simple method is to directly compress each key-value item individually. However, such a small compression unit would result in a low compression ratio. As reported in prior work [5], in Facebook's Memcached workload, the size of most (about 90%) values is under 500 bytes, which is unfriendly to compression. As shown is Figure 4, around 80% of items in the three datasets, Weibo [27, 28], Twitter [77], and Reddit [65], are under 288, 418, and 637 bytes, respectively. Compressing such small-size values individually suffers from the low-compression-ratio problems (see Figure 2), and the space saving by compression would be limited.
- *Option 2: Compressing data in slab granularity.* Another natural consideration is to compress the in-memory slab, which is typically large (1 MB in Fatcache as default). However, upon a request to a key-value item in a compressed slab, the entire compressed slab has to be loaded into memory, decompressed, and then the corresponding item is retrieved from the decompressed slab. This *read amplification* problem incurs two kinds of overhead. (1) *I/O overhead.* Irrelevant data have to be transferred over the I/O bus, no matter they are needed or not. (2) *Computational overhead.* We apply `lz4` [52], an efficient compression algorithm, on data chunks of different sizes, generated from `/dev/urandom`. As shown in Figure 5, the computational overhead becomes non-negligible when the compressed data chunk size increases, considering that a flash page read is typically only about 25–100 μs. So, compressing data in slabs would cause concerns on the overhead issues.

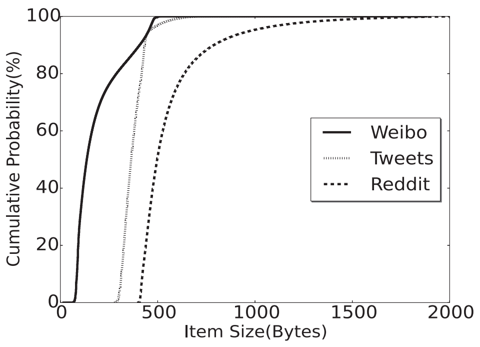


Fig. 4. Distribution of item sizes.

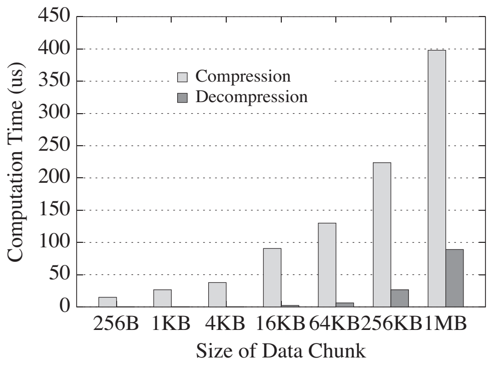


Fig. 5. Compression time vs. unit size.

The above analysis indicates that we must carefully balance between two design goals, achieving a high compression ratio and reducing the overhead. Directly applying compression in either slab or slot granularity would be unsatisfactory.

SlimCache attempts to make a GET operation completed in no more than one flash page read. We keep track of the compression ratio after each compression operation at runtime, and calculate the estimated compression ratio, $est\_compression\_ratio$, by calculating the arithmetic mean of the measured compression ratios. The estimated compression container size is calculated as $k \times flash\_page\_size \times est\_compression\_ratio$, where $flash\_page\_size$ is the known flash page size (typically 4–16 KB), and must be no smaller than a memory page size (4 KB as default). The coefficient k is the multiplier of $flash\_page\_size$ when multiple flash pages are needed. The rationale behind this is that we desire to provide the compression algorithm a sufficient amount of data for compression (at least one memory page), and also minimize the extra I/Os of loading irrelevant data (at least one flash page has to be loaded anyway). It is worth noting that the purpose is not to guarantee that the amount of data after being compressed will surely fit into one flash page but to estimate a proper granularity to meet the goal as best efforts. Also, one can adjust the coefficient k according to the properties of the target workloads to achieve the best performance. We set $k = 1$ in the prototype, which works well in our experiments. We will study the effect of compression granularity on the performance in Section 4.3.1.

3.4 Hot/Cold Data Separation

To mitigate the computational overhead, it is important to selectively compress the infrequently accessed data, *cold data*, while leaving the frequently accessed data, *hot data*, in their original format to avoid the read amplification problem and unnecessary decompression overhead. For this purpose, we logically partition the flash space into two regions: The *hot area* contains frequently accessed key-value items in the uncompressed format; the *cold area* contains relatively infrequently accessed key-value items in the compressed format, if compressible (see Figure 6). We will present a model-based approach to automatically tune the sizes of the two areas in Section 3.5.

Identifying hot/cold data. SlimCache labels the "hotness" at the fine-grained key-value item level rather than the slab level, considering that a slab could contain a random collection of key-value items that have completely different localities (hotness). Identifying the hot key-value items rather than hot slabs would provide more accuracy and efficiency. To identify the hot key-value items, we add an attribute, called *access_count*, in each entry of the mapping table. When updating a key-value item, its access_count is reset to 0. When the key-value item is accessed, its access_count

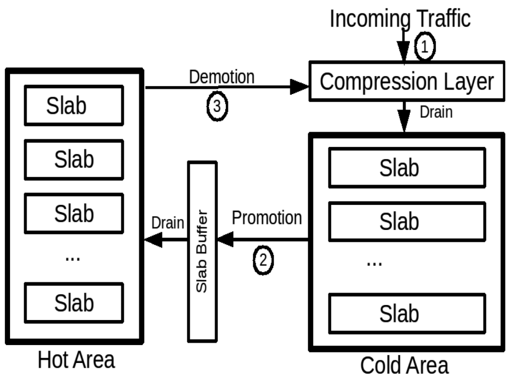


Fig. 6. Hot and cold data separation.

is incremented by 1. During garbage collection, if a compressed key-value item's access_count is greater than zero, it means that this key-value item has been accessed at least once in a compressed format and could be a candidate for promotion to the hot area or continue to stay in the cold area. In Section 3.6, we will discuss these two polices. Another issue is how many bits should be reserved for an *access_count*. Intuitively, the more bits, the more precisely we can tell the hotness of a key-value item, but more overhead is involved. We will study this effect in Section 4.3.4.

Admitting key-value items in cache. Two options are possible for handling new key-value items. The first one is to insert the newly admitted key-value item into the hot area, and when the hot area runs out of space, we demote the cold items (access_count is 0) into the cold area, compress and "archive" them there. The second method is to first admit the key-value item into the cold area, and when the garbage collection process happens, we decompress and promote the hot items to the hot area. Both approaches have advantages and disadvantages. The former has to write most key-value data at least twice (one to the hot area and the other to the cold area), causing *write amplification*; the latter applies compression in the front, which could cause the decompression overhead if a promotion happens later. Considering the high locality in key-value caches, only a small set of key-value items is hot and most are cold, the latter solution would remove unnecessary flash writes and thus be more efficient. We choose the second solution in SlimCache.

Promotion and demotion. Key-value items can be promoted from the cold area to the hot area, and vice verse. Our initial implementation adopts a typical promotion approach, which immediately promotes a key-value item upon access, if its access_count is non-zero. However, we soon found a severe problem with this approach—to create a log-like access pattern on flash, when a key-value item is promoted into the hot area, its original copy in the cold area cannot be promptly freed. Instead, it has to be simply marked as "obsolete" and waits for the garbage collection process to recycle at a later time. During this time window, the occupied space cannot be reused. In our experiments, we have observed a hit ratio loss of 5–10 percentage points (p.p.) caused by this space waste. If we enforce a direct reuse of the flash space occupied by the obsolete key-value items, then random writes would be generated to flash, which is not desirable either.

SlimCache solves this challenging problem in a novel way. Upon a repeated access to a key-value item, we do not immediately promote it to the hot area; rather, we postpone the promotion until the garbage collector scans the slab. In the victim slab, if a key-value item has an access_count greater than the threshold (see Section 3.6), we promote it to the hot area and its original space is reclaimed then. In this way, we can ensure that hot data be promoted without causing any space loss, and in the meantime, we still can preserve the sequential write pattern.

To determine the coldest slab for demotion from the hot area, the slabs are organized in a linked list, and we use the standard Least Recently Used (LRU) replacement algorithm in the slab granularity for eviction. Every time a slab is accessed, it is regarded as the Most Recently Used (MRU) one and moved to the head of the list. When the hot area is full, the LRU hot slab is selected for demotion. Instead of directly dropping all the key-value items, SlimCache compresses the items with a non-zero access_count and demotes them into the cold area, which offers the items that have been accessed a second chance to stay in cache. For the items that have never been accessed, SlimCache directly drops them, since they are unlikely to be accessed again.

In both promotion and demotion, we simply place the compressed/uncompressed key-value items back to the slab buffer, and the slab buffer flushing process is responsible for writing them to flash later. Such a hot/cold data separation scheme is highly effective. In our experiments, the write amplification caused by data movement between the two areas is found rather low (see Section 4.3.2).

3.5 Adaptive Partitioning

As mentioned above, the partitioning of flash space effectively determines the portion of key-value items being stored in compressed or uncompressed format. The larger the cold region is, the more flash space could be saved, and the higher hit ratio would be; however, the more I/Os have to experience a time-consuming decompression. Thus, we need to first identify a reasonable initial partitioning plan and also provide a dynamic partitioning scheme to reflect the change of workload patterns. We use a simple model-based solution for such adaptive partitioning.

Initializing partitions. If we assume the workload distribution follows the Zipf's law [8, 70, 88], then a small portion of records will serve most of the requests. The Zipfian distribution has been extensively studied. In the following, we adopt the expressions defined in prior work [45] to explain how we determine the initial partition ratio. As defined in the work [45], the Zipfian distribution has the random variable X and parameters α and N, and the probability is

$$f(x) = \frac{1}{x^{\alpha} \sum_{i=1}^{N} (\frac{1}{i})^{\alpha}}, x = 1, 2, \ldots, N, \tag{1}$$

where N is a positive integer and $\alpha \geq 0$. The true Zipf's law [88] has $\alpha = 1$, and a broader class of Zipf-like distributions [8] has $0 < \alpha < 1$ and close to 1. If we represent the summation in the denominator as

$$H_{N,\alpha} = \sum_{i=1}^{N} \left(\frac{1}{i}\right)^{\alpha}, \tag{2}$$

then the cumulative distribution function on the support of X becomes

$$F(x) = P(X \leq x) = \frac{H_{x,\alpha}}{H_{N,\alpha}}. \tag{3}$$

In the case that $\alpha = 1$, asymptotically $F(x) \approx \frac{\ln x}{\ln N}$. If we assume the distribution follows a true Zipf's law, where $\alpha = 1$, then according to Reference [45], the population mean of the true Zipf's law is

$$E[X] = \frac{H_{N,\alpha-1}}{H_{N,\alpha}}. \tag{4}$$

When we set x to $E[X]$, the cumulative distribution function becomes

$$F(E[X]) \approx \frac{\ln(\frac{N}{\ln N})}{\ln N} = 1 - \frac{\ln \ln N}{\ln N}. \tag{5}$$

For more details, one may refer to prior studies [8, 45, 70, 88]. In this article, we use Equations (4) and (5) to determine the initial partition ratio. Two examples are shown as below.

(1) When the number of records in the system is 100 million (i.e., $N = 100M$), $E[X] = \frac{N}{\ln N} = 100M/18.42 = 5.43M$. This means that the update frequency of 5.43M records (i.e., approximately 5.4%) is above the average frequency. The hit ratio of all the 5.43M records is $F(E[X]) \approx 1 - \frac{\ln \ln N}{\ln N} = 1 - 2.91/18.42 = 84.2\%$.
(2) When the number of records in the system is 10 billion (i.e., $N = 10B$), $E[X] = \frac{N}{\ln N} = 10B/23.03 = 434M$. This means that the update frequency of 434M records (i.e., approximately 4.3%) is above the average frequency. The hit ratio of all the 434M records is $F(E[X]) \approx 1 - \frac{\ln \ln N}{\ln N} = 1 - 3.14/23.03 = 86.4\%$.

In our prototype, we set the hot area initially as 5% of the flash space. According to our analysis above, it is expected to satisfy about 85% of service requests to the cache server. Then, we use a model-based on-line partitioning method to adaptively adjust the sizes of the two areas at runtime.

Cost model-based partitioning. As mentioned above, there is a tradeoff between the decompression overhead and the cache hit ratio. We propose a simple cost model to estimate the effect of area partitioning:

$$\begin{aligned} Cost = H_{hot} \times C_{hot} + H_{cold} \times C_{cold} \\ + (1 - H_{hot} - H_{cold}) \times C_{miss}, \end{aligned} \tag{6}$$

where H_{hot} and H_{cold} are the ratios of hits contributed by the hot key-value items and the cold key-value items on the flash, respectively, C_{hot} and C_{cold} are the costs when the data is retrieved from the hot and cold areas, respectively, and C_{miss} is the cost of fetching data from the backend data store. These parameters can be obtained through measurement during runtime.

As shown in Algorithm 1, our model needs to consider two possible partitioning decisions, increasing or decreasing the hot area size:

- *Option #1: Increasing hot area size.* If the size of the hot area is increased by S, then more data could be cached in the uncompressed format. The hit ratio contributed by the head S space of the cold area is denoted as H_{c_head}. The hit ratio H'_{hot} provided by the hot area after increasing by S becomes $H_{hot} + H_{c\_head}/compression\_ratio$. The hit ratio H'_{cold} provided by the cold area after decreasing by S becomes $H_{cold} - H_{c\_head}$.
- *Option #2: Decreasing hot area size.* If the size of the hot area is decreased by S, then there will be less uncompressed data cached. The hit ratio contributed by the tail S space of the hot area is denoted as H_{h_tail}. The hit ratio H'_{hot} provided by the hot area after decreasing by S becomes $H_{hot} - H_{h\_tail}$. Correspondingly, the cold area will grow by S, so the hit ratio H'_{cold} provided by the cold area will be increased to $H_{cold} + H_{h\_tail} \times compression\_ratio$.

We compare the current cost with the predicted cost after the possible adjustments. If the current cost is lower, then we keep the current partitioning unchanged. If the predicted cost after increasing or decreasing the hot area is lower, then we enlarge or reduce the hot area size, accordingly.

The above-said model is simple yet effective. Other models, such as miss ratio curve [87], could achieve a more precise prediction but is more complex and costly. In our scenario, since multiple factors vary at runtime anyway and the step S is relatively small, the cost estimation based on this simple model works well in our experiments.

ALGORITHM 1: DYNAMIC PARTITIONING

```
1: Data: compression_ratio, init_hot_area
2: Result: The partition of flash space
3: // $H_{hot}$ and $H_{cold}$ mean the hit ratios in the hot and cold areas, respectively.
4: // $init\_hot\_area$, $curr\_hot\_area$ and $opt\_hot\_area$ mean
5: // the initialized, current, and optimal hot area sizes, respectively.
6: // $Hit()$ calculates the estimated hit ratios in the hot and cold areas.
7: // $Cost()$ calculates the estimated overall cost with the estimated hit ratios.
8: $H_{hot}, H_{cold} \leftarrow Hit(init\_hot\_area, compression\_ratio)$;
9: $opt\_hot\_area \leftarrow init\_hot\_area$;
10: $opt\_cost \leftarrow Cost(H_{hot}, H_{cold})$;
11: $curr\_hot\_area \leftarrow init\_hot\_area$;
12: $step \leftarrow \{+S, -S\}$
13: $max\_hot\_area \leftarrow predefined\_threshold$
14: procedure $DYNAMIC\_PARTITIONING$
15:     for $i \leftarrow 0; i \leq 1; i \leftarrow i + 1$ do
16:         $new\_hot\_area \leftarrow curr\_hot\_area + step[i]$;
17:         $H_{hot}, H_{cold} \leftarrow Hit(new\_hot\_area, compression\_ratio)$;
18:         $new\_cost \leftarrow Cost(H_{hot}, H_{cold})$;
19:         if $new\_cost \leq opt\_cost$ and $|opt\_cost - new\_cost| > \epsilon$ then
20:             if $new\_hot\_area \leq max\_hot\_area$ then
21:                 $opt\_cost = new\_cost$;
22:                 $opt\_hot\_area \leftarrow new\_hot\_area$;
23:             end if
24:         end if
25:     end for
26:     $curr\_hot\_area \leftarrow opt\_hot\_area$
27:     $ADJUST\_PARTITION(opt\_hot\_area)$
28: end procedure
```

3.6 Garbage Collection

Garbage collection is a must-have process in flash-based key-value cache systems. Since flash memory favors large and sequential writes, when certain operations (e.g., SET and DELETE) create obsolete value items in slabs, we need to write the updated content to a new slab and recycle the obsolete or deleted key-value items at a later time. When the system runs out of free slabs, we need to reclaim their space on flash.

As Figure 7 shows, SlimCache deploys a *Two-stage Garbage Collection* similar to our prior work [73]. When the number of free slabs in the cold area of SSD drops to the start watermark (W_{start}), *Space-based Eviction* is triggered and quickly cleans slabs. It switches to *Locality-based Recycling*, when the free slab number is brought back to the low watermark (W_{low}). The GC process continues until the number of free slab reaches the high watermark (W_{high}).

- **Space-based eviction:** When the number of the free slabs in the cold area drops to below the start watermark, W_{start}, the space-based eviction process is triggered to release the high space pressure. All the data in the LRU slab, including the valid data, will be dropped directly to reclaim the free space quickly. This is safe, since the backend data store still contains the most recent version. After updating the hash table mapping, the whole slab

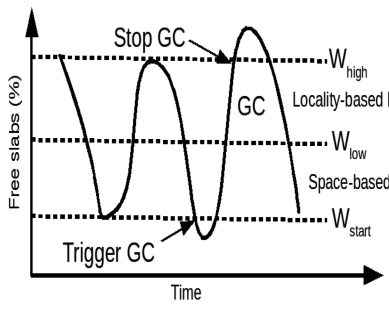


Fig. 7. An illustration of the two-stage GC.

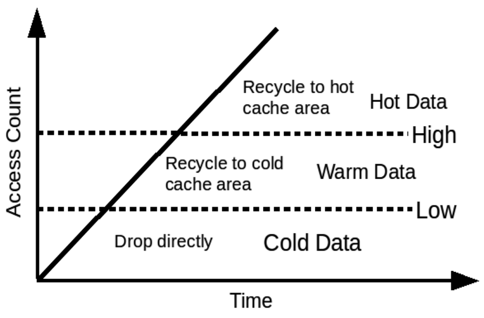


Fig. 8. Data recycling in garbage collection.

is put into the free cold slab list. This GC policy aims to reclaim the free space as fast as possible. When the number of free slabs reaches the low watermark, W_{low}, the GC process switches to locality-based recycling.

- **Locality-based recycling:** When the number of the free slabs in the cold area is between the low watermark, W_{low}, and the high watermark, W_{high}, the locality-based recycling is triggered. We search the slab queue of the cold area to identify the slab that is most frequently accessed. The whole slab is read and based on the access_count, the key-value items can be divided into three possible categories: *hot*, *warm*, and *cold*. Accordingly, as illustrated in Figure 8, we could apply different recycling policies for them—the cold or invalid (obsolete or deleted) key-value items are dropped directly; the warm items continue to stay in the cold area in the compressed format; the hot items are decompressed and promoted into the hot area. Note that we may also make a coarser differentiation by dividing the items into only two categories, hot and cold. In fact, in our experiments, we find that using a 1-bit counter to differentiate hot and cold items generally satisfies our needs in most cases. After updating the hash table mappings, the whole slab is cleaned and placed back to the free cold slab list. Unlike the space-based eviction, this garbage collection procedure takes more time, and collects and promotes valuable items for the purpose of retaining a high hit ratio. When the number of free slabs reaches the high watermark, W_{high}, the GC process stops.

These two GC policies are designed for different situations. The space-based eviction is responsible for evicting cold items and aims to reclaim the free space as quickly as possible. So it is used when SlimCache runs out of free slabs to a severe degree. The locality-based recycling is mainly responsible for collecting and promoting the hot items to retain the hit ratio.

The demotion process in the hot area is similar. When the free space is below the low watermark, W_{low}, the LRU slab is selected and all the valid items are compressed and demoted into the cold area. The process repeats until the number of free slabs reaches to the high watermark, W_{high}.

3.7 Dynamic Compressibility Recognition

Some key-value data are incompressible by nature, such as encrypted or already-compressed data, e.g., JPEG images. Compressing them would not bring any benefit but incurs unnecessary overhead. We need to quickly estimate data compressibility and selectively apply compression.

A natural indicator of data compressibility is the *entropy* of the data [72], which is defined as $H = -\sum_{i=1}^{n} p_i \times \log_b p_i$. Entropy quantitatively measures the information density of a data stream based on the appearing probability (p_i) of the n unique symbols. It provides a predictive method to estimate the amount of redundant information that could be removed by compression, such as the Huffman encoding [31, 43]. Entropy has been widely used for testing data compressibility in various scenarios, such as primary storage [31], memory cache [14], device firmware [69], image compression [56], and many others. We use *normalized entropy* [80], which is the entropy divided by the maximum entropy ($\log_b n$), to quickly filter out the incompressible data, which are indicated by a high entropy value.

ALGORITHM 2: DYNAMIC COMPRESSION RECOGNITION

```
1:  Result: threshold for the entropy
2:  $init\_entropy\_value \leftarrow entropy(randomstrings)$;
3:  $entropy\_threshold \leftarrow init\_entropy\_value$;
4:  $global\_min\_cmpr \leftarrow predefined\_value$;
5:  $curr\_min\_cmpr \leftarrow max\_init\_cmpr$;
6:  for each $curr\_blk$ do
7:      if $global\_min\_cmpr < compression\_ratio(curr\_blk) < curr\_min\_cmpr$ then
8:          $entropy\_threshold = entropy(curr\_blk)$;
9:          $curr\_min\_cmpr = compression\_ratio(curr\_blk)$;
10:     end if
11: end for
12: $UPDATE\_THRESHOLD(entropy\_threshold)$
```

We initialize the threshold to be the average entropy value of randomly generated strings. Since randomly generated strings are mostly incompressible [75], we can effectively skip compression operations for strings whose normalized entropy is larger than that of random strings (i.e., incompressible) to remove the unnecessary computational overhead.

We have developed a Dynamic Compressibility Recognition (DCR) algorithm to adaptively adjust the entropy threshold during runtime based on the real-time compression ratio. It works as shown in Algorithm 2. The *global_min_cmpr* is a predefined minimum compression ratio that is acceptable for SlimCache to apply compression operations to gain performance benefits. The *curr_min_cmpr* is the minimum compression ratio found in the current workload. If a current data block's compression ratio is found smaller than *curr_min_cmpr* and greater than *global_min_cmpr*, then the *entropy_threshold* and the *curr_min_cmpr* are updated as the entropy value and the compression ratio of the current data block, respectively. The rationale behind this algorithm is that we first ensure that using the entropy threshold would not result in a compression ratio lower than the acceptable ratio (defined by *global_min_cmpr*), and we initially set a high entropy threshold to ensure a high compression ratio, and gradually tune down this entropy threshold if we observe an acceptable compression ratio during runtime. In this way, we can find the best cutoff thresholds for different workloads.

The items that are detected incompressible are directly written to the cold area in their original uncompressed format. Thus, note that the cold area could hold a mix of compressed and uncompressed data. This entropy-based estimation fits well in our caching system, especially for its simplicity, low computation cost, and time efficiency. We will study the effect of dynamic compressibility recognition in Section 4.3.6.

3.8 Summary

SlimCache shares the basic architecture design with regular flash-based key-value caches, such as the slab/slot structure, the mapping table, the in-memory slab buffer, and the garbage collection. However, SlimCache also has several unique designs to realize efficient data compression.

First, we add a compression layer that applies compression algorithms on the suitable items at a proper granularity. The compressed unit is placed back to the slab-based cache structure as regular key-value items, so that the cache space can be consistently allocated and managed. Accordingly, the mapping structure is also modified to point to either compressed or uncompressed items. Second, SlimCache dynamically divides the flash cache space into two separate regions, a hot area and a cold area, to store data in different formats for minimizing the computational overhead caused by compression. Third, SlimCache also enhances the garbage collection process by integrating it with the hot/cold data separation mechanism to avoid the cache space waste caused by data movement between the two areas. Finally, we add compressibility recognition mechanism to identify the data suitable for compression. These differences between SlimCache and a regular flash-based key-value cache, such as Fatcache, contribute to the significant performance gain.

4 EVALUATION

To evaluate the proposed schemes, we have implemented a prototype of SlimCache based on Twitter's Fatcache [78], which has been used in academic works [4, 25, 42, 71, 73] and commercial product benchmarking [67, 68]. Our implementation accounts for about 2,700 lines of code in C. In this section, we present our evaluation results for the SlimCache design on a real SSD hardware platform.

4.1 Experimental Setup

Our experiments are conducted on three Lenovo ThinkServers. All three servers feature an Intel Xeon(R) 3.40 GHz CPU and 16 GB DRAM memory. In the key-value cache server, an 800 GB Intel 910 PCIe SSD is used as the storage device for key-value caching. Note that for a fair comparison, only a part of the SSD space (12–24 GB) is used for caching in our experiments, proportionally to the workload dataset size. All the experiments use `direct_io` to minimize the effect of the operating system page cache. In Fatcache and SlimCache, the consumed memory space is mainly for holding the hash mapping structure in memory. Each mapping entry consumes 44 bytes, and the memory consumption is largely proportional to the number of key-value items in cache. For example, in our experiments with the Twitter data set, SlimCache consumes up to about 4 GB memory for indexing 98 million key-value items. Fatcache, due to the less amount of key-value items being cached, consumes proportionally less memory. Our backend data store is MongoDB v3.4.4 running on a separate server with 1 TB Seagate 7,200 RPM hard drive. The clients run on another ThinkServer to generate traffic to drive the experiments. The three servers are connected via a 10 Gbps Ethernet switch. For all the three servers, we use Ubuntu 14.04 with Linux kernel 4.4.0-31 and Ext4 file system.

We use Yahoo's YCSB benchmark suite [24] to generate workloads to access key-value items, following three different distributions, Zipfian, Normal, and Hotspot,[1] as described in prior work [9, 84] to simulate typical traffic in cloud services [5]. Since the YCSB workloads do not contain actual data, we use the datasets from Twitter [77], Flickr [33], and Reddit [65] to emulate three typical types of key-value data with different compressibility. The Twitter and Reddit datasets have a high compression ratio (about 2–4), while the Flickr dataset has a low compression ratio, near to 1 (incompressible). To generate fixed-size compressible values (Section 4.3.1), we use the text generator [26] based on Markov chain provided by Python to generate the pseudo-random fixed-size values. We use the lightweight `lz4` [52] and the heavyweight deflate method in `zlib` [66] for compression in comparison.

[1]Hotspot is a distribution in which 80% of the operations access 20% of the data items and the rest 20% of the operations access the rest 80% items. Elements for the hot set and cold set are chosen in an uniform manner.

In the following, our first set of experiments evaluates the overall system performance with a complete setup, including both the cache server and the backend database. Then, we focus on the cache server and study each design component individually. Finally, we study the cache partitioning and further give the overhead analysis.

4.2 Overall Performance

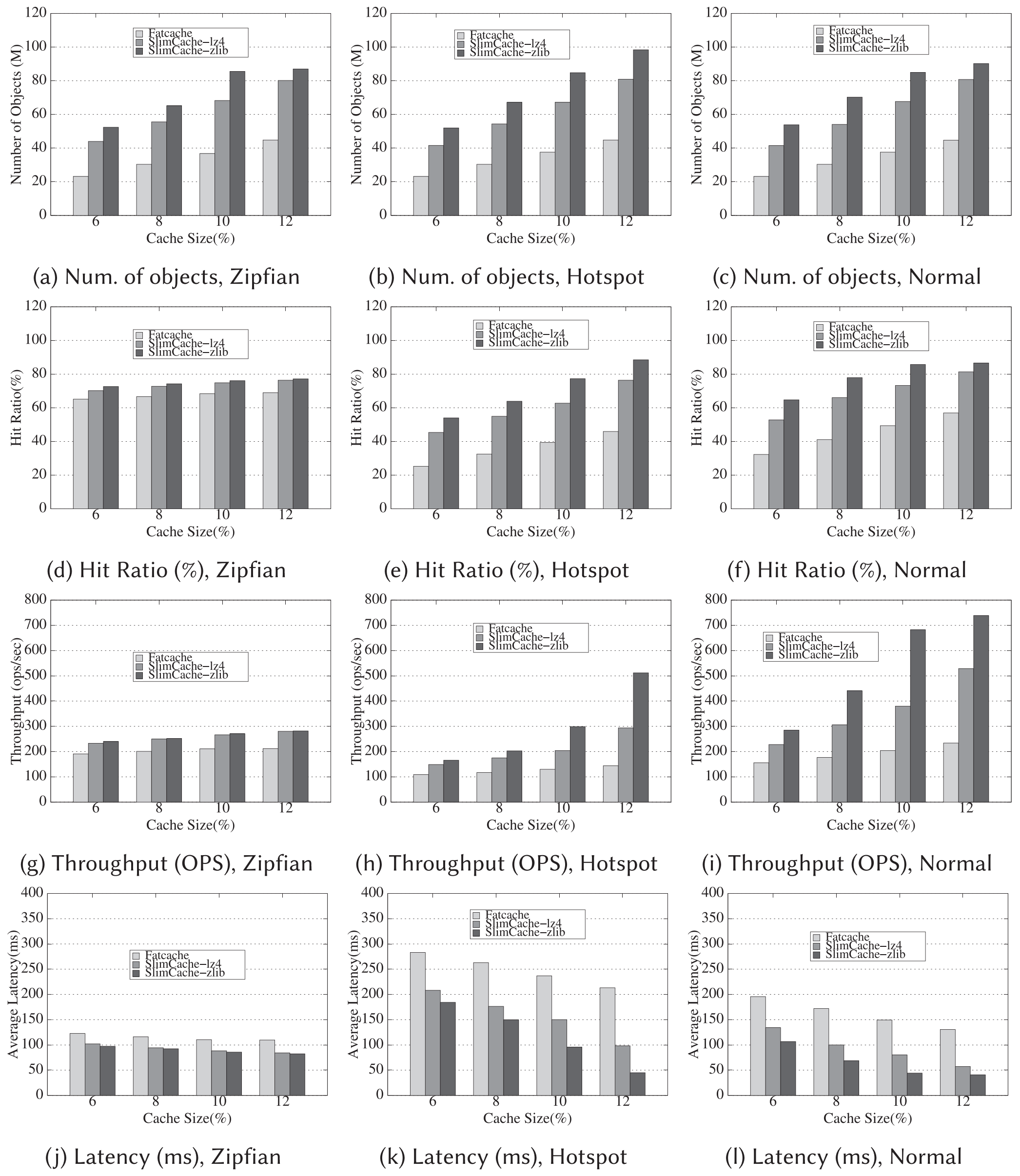
In this section, our experimental system simulates a typical key-value caching environment, which consists of clients, key-value cache servers, and a database server in the backend. We test the system performance by varying the cache size from 6% to 12% of the dataset size, which is about 200 GB in total (480M, 300M, and 2M records for Twitter, Reddit and Flicker, respectively). Thus, only part of the 800 GB SSD capacity is used as cache (12–24 GB) for fair comparison. For each test, we first generate the dataset to populate the database, and then generate 300 million `GET` requests. We only collect the data for the last 400K requests in the trace replaying to ensure that the cache server has been warmed up. All the experiments use 8 key-value cache servers and 32 clients.

4.2.1 Performance for Twitter Dataset. Our on-line compression solution can "virtually" enlarge the size of the cache space. Figures 9(a), 9(b), and 9(c) show the number of items cached in SlimCache compared to the stock Fatcache with the same amount of flash space.

As shown in Figure 9(a), the number of items in cache increases substantially by up to 125.9%. Such an effect can also be observed in other distributions. Having more items cached in SlimCache means a higher hit ratio. Figures 9(d), 9(e), and 9(f) show the hit ratio difference between Fatcache and SlimCache. In particular, when the cache size is 6% of the dataset, the hit ratio (54%) of SlimCache-zlib for the hotspot distribution is 2.1 times of the hit ratio provided by Fatcache. For the Zipfian and normal distributions, the hit ratio of SlimCache-zlib reaches 72.6% and 64.7%, respectively. A higher hit ratio further results in a higher throughput. As the backend database server runs on a disk drive, the increase of hit ratio in the flash cache can significantly improve the overall system throughput and reduce the latencies. As we can see from Figures 9(g), 9(h), and 9(i), compared to Fatcache, the throughput improvement provided by SlimCache-zlib ranges from 25.7% to 255.6%, and the latency decrease ranges from 20.7% to 78.9%, as shown in Figures 9(j), 9(k), and 9(l).

To further understand the reason of the performance gains, we repeated the experiments with compression disabled. Table 1 shows the results with a cache size as 6% of the dataset. We can see that without data compression, solely relying on the two-area (hot and cold area) cache design, SlimCache only provides a slight hit ratio increase (1.1–1.5 p.p.) over the stock Fatcache. In contrast, SlimCache with compression provides a more significant hit ratio improvement (5.1–20.8 p.p.). It indicates that the performance gain is mainly a result of the virtually enlarged cache space by on-line compression rather than the two-area cache design.

4.2.2 Performance for Reddit Dataset. We further conduct experiments with Reddit on SlimCache to illustrate the effectiveness of our proposed approaches. Figures 10(a)–10(c) show the number of key-value items cached in SlimCache compared to Fatcache with the same amount of cache space. We can see from Figure 10(a) that the number of items cached in SlimCache-zlib increases significantly by up to 223.4%. Such an increase can also be found with other distributions. More key-value items cached by SlimCache result in a higher hit ratio. We can observe the hit ratio difference between Fatcache and SlimCache in Figures 10(d)–10(f). For the Zipfian distribution, when the cache size is 6% of the working set, the hit ratio provided by SlimCache-zlib is about 69.3%, which is about 7 p.p. higher than Fatcache. A higher hit ratio further helps improve the throughput. As Figures 10(g)–10(i) show, the throughput improvement provided



(a) Num. of objects, Zipfian (b) Num. of objects, Hotspot (c) Num. of objects, Normal

(d) Hit Ratio (%), Zipfian (e) Hit Ratio (%), Hotspot (f) Hit Ratio (%), Normal

(g) Throughput (OPS), Zipfian (h) Throughput (OPS), Hotspot (i) Throughput (OPS), Normal

(j) Latency (ms), Zipfian (k) Latency (ms), Hotspot (l) Latency (ms), Normal

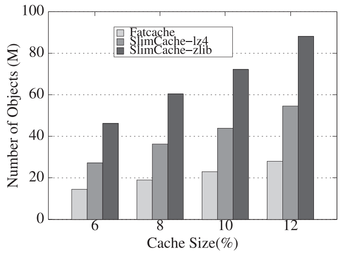
Fig. 9. Performance of Twitter dataset.

by SlimCache-zlib ranges from 18.9% to 380.1%. We can also observe in Figures 10(j)–10(l) that, as the cache size increases, the latency decrease ranges from 16.4% to 80.7%. It well shows that SlimCache can gain significant performance improvement for both Twitter and Reddit datasets.

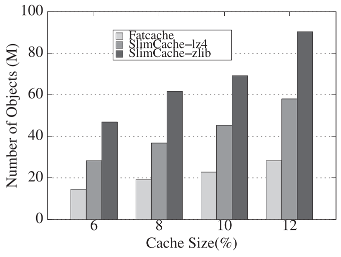
4.2.3 Effect of Replacement Algorithms. Figure 11 shows the effect of different replacement algorithms on the performance of the system for the Reddit workload with Zipfian distribution. We compare Fatcache with the FIFO algorithm (default) and the LRU algorithm with our proposed SlimCache-zlib. Figure 11(a) shows that the hit ratio increases from 66.39% to 68.72%, if we

Table 1. Hit Ratio Gain of Compression in SlimCache

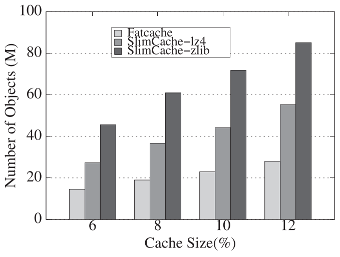
Scheme	Zipfian	Hotspot	Normal
Fatcache	65.1%	25.2%	32%
SlimCache w/o Compression	66.2%	26.4%	33.5%
SlimCache with lz4	70.2%	45.4%	52.8%



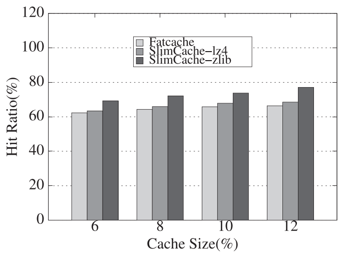
(a) Num. of objects, Zipfian



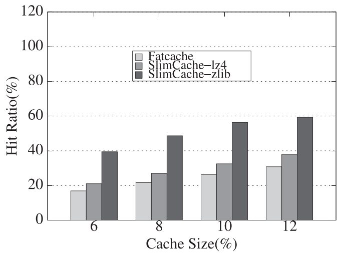
(b) Num. of objects, Hotspot



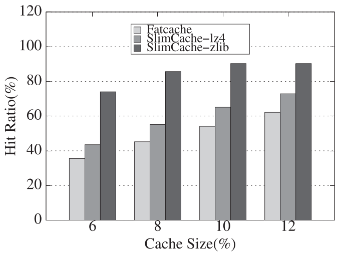
(c) Num. of objects, Normal



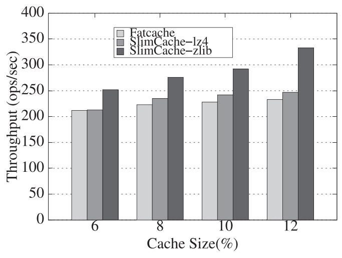
(d) Hit Ratio (%), Zipfian



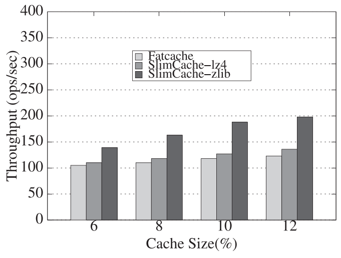
(e) Hit Ratio (%), Hotspot



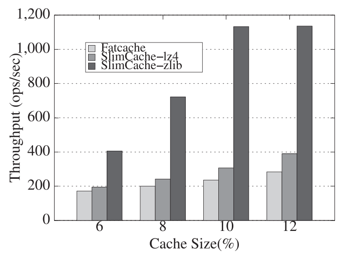
(f) Hit Ratio (%), Normal



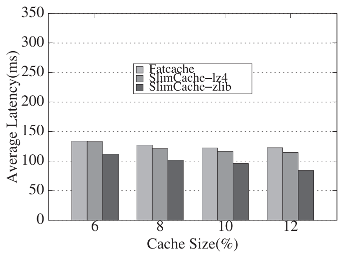
(g) Throughput (OPS), Zipfian



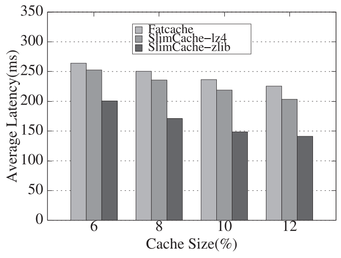
(h) Throughput (OPS), Hotspot



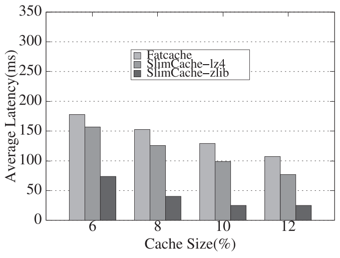
(i) Throughput (OPS), Normal



(j) Latency (ms), Zipfian

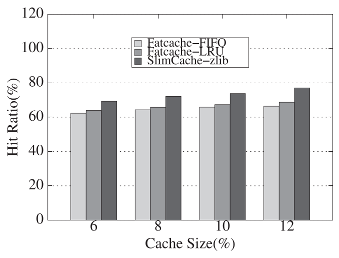


(k) Latency (ms), Hotspot

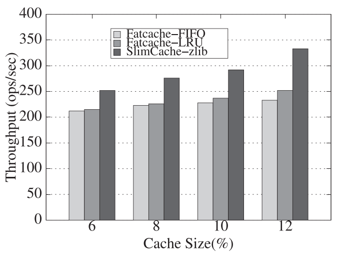


(l) Latency (ms), Normal

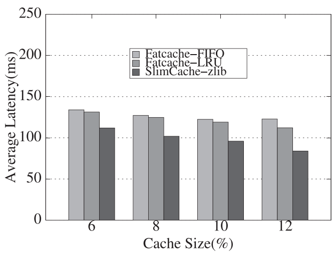
Fig. 10. Performance of Reddit dataset.



(a) Hit ratio (%)

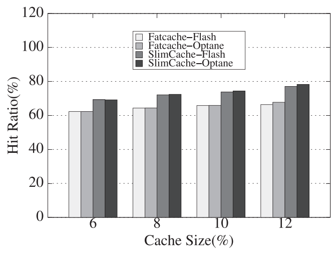


(b) Throughput (OPS)

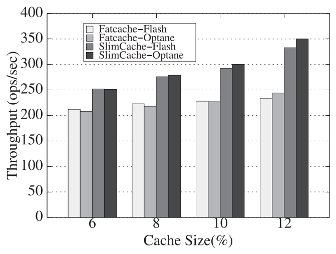


(c) Latency (ms)

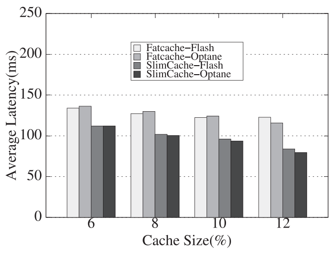
Fig. 11. Effect of cache replacement algorithms.



(a) Hit ratio (%)



(b) Throughput (OPS)

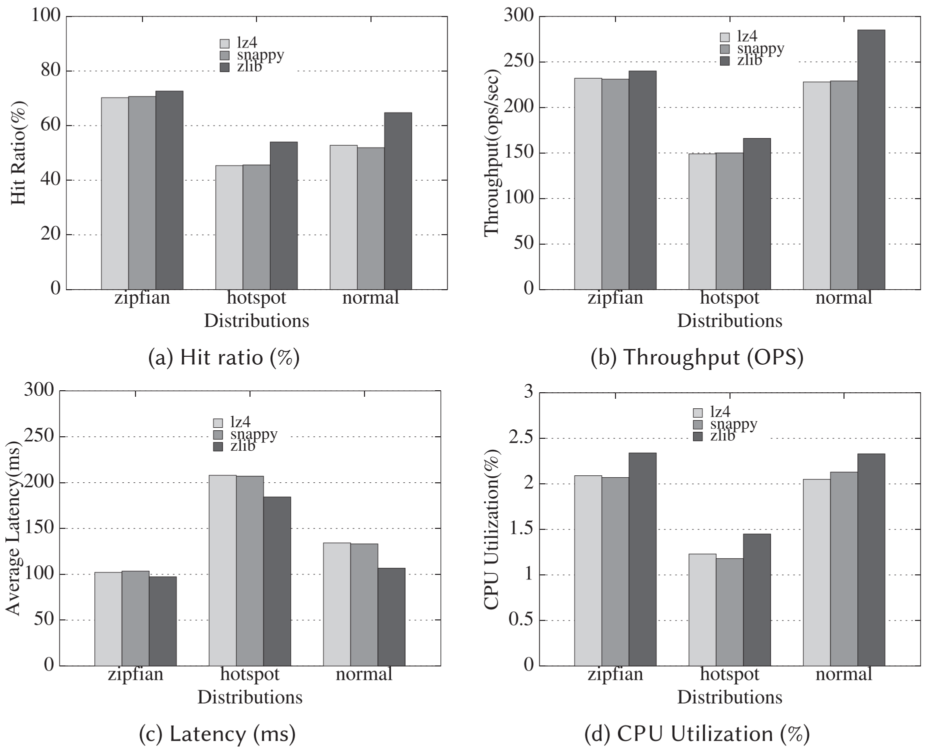


(c) Latency (ms)

Fig. 12. Effect of caching devices.

change the replacement algorithm from FIFO to LRU for Fatcache when the cache size is 12% of the working set. Accordingly, Figure 11(b) shows that the throughput increases from 233 to 252 ops/s and Figure 11(c) shows that the average latency decreases from 122.78 to 112.17 ms. These experimental results illustrate that the LRU replacement algorithm only slightly improves performance over FIFO. In contrast, SlimCache-zlib outperforms Fatcache-LRU significantly. For example, the throughput of SlimCache-zlib is 32.1% higher than that of Fatcache-LRU when the cache size is 12%. It clearly shows that most of the performance gain of SlimCache-zlib is due to the efficient data compression mechanism, which significantly increases the cache hit ratio.

4.2.4 *Effect of Caching Devices.* Figure 12 shows the effect of caching devices on the performance of the system for the Reddit workloads with Zipfian distribution. In this experiment set, we replace the caching device with a 280 GB Intel 900P Optane SSD [35], which is built on 3D XPoint non-volatile memory, while keeping the other configurations unchanged. Figure 12(a) shows that the conventional NAND flash-based SSD and the new 3D XPoint-based SSD provide nearly identical hit ratios. If we compare the two devices, then as we can see in Figure 12(b), when the cache size is 12% of the workload, Fatcache-Optane can provide 4.7% higher throughput than Fatcache-Flash, and SlimCache-Optane can provide 5.1% higher throughput than SlimCache-Flash. Correspondingly, as Figure 12(c) shows, Fatcache-Optane reduces 5.7% average latency than Fatcache-Flash, and SlimCache-Optane reduces 5.3% average latency than SlimCache-Flash. Together with the experimental results shown in Section 4.2.2, we can find that when the speeds of caching devices (SSDs) are on the same order of magnitude, it only incurs slight performance difference, since the backend data store, which stores data in hard disk drive, is much slower than the caching device. As a consequence, increasing the cache hit ratio, which means fewer accesses being generated to the slow backend data store, would improve system performance more significantly than simply using a faster and more expensive caching device. This observation further illustrates that SlimCache,



(a) Hit ratio (%)

(b) Throughput (OPS)

(c) Latency (ms)

(d) CPU Utilization (%)

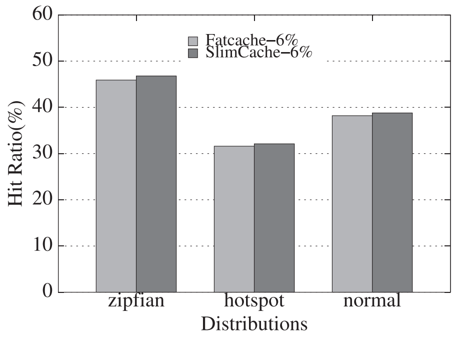
Fig. 13. Effect of different compression algorithms.

which aims to improve hit ratio by caching more key-value items with compression techniques, is practically a more effective and cost-efficient approach.

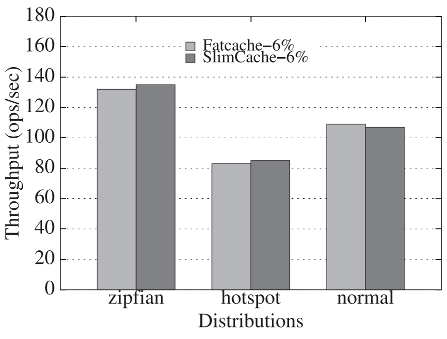
4.2.5 Effect of the Compression Algorithms. We compare the performance of applying three different compression algorithms, the lightweight `lz4`, `snappy`, and heavyweight deflate in `zlib`, when the cache size is 6% of the Twitter dataset.

Figure 13 shows that `zlib` performs the best among the three, while `lz4` and `snappy` are very similar. In particular, `zlib` provides a hit ratio gain of 2.4–11.9 p.p. over `lz4` and `snappy`, which results in a throughput increase of 3.4%–25% and a latency decrease of 6%–20.6%. Meanwhile, the CPU utilization ratio is up to 2.34% in all the cases as shown in Figure 13(d). This indicates that heavyweight compression algorithms, such as the deflate method in `zlib`, work fine with flash-based caches, since the benefit of increasing the hit ratio significantly outweighs the incurred computational overhead in most of our experiments.

4.2.6 Performance for Flickr Dataset. We have also studied the performance of SlimCache when handling incompressible data. SlimCache can estimate the compressibility of the cache data and skip the compression process for the items that are not suitable for compression, such as already-compressed images. We have tested SlimCache with the Flickr dataset, and Figure 14 shows that for workloads with little compression opportunities, SlimCache can effectively identify and skip such incompressible data and avoid unnecessary overhead, showing nearly identical performance as the stock Fatcache.



(a) Hit ratio (%)



(b) Throughput (OPS)

Fig. 14. Hit ratio and throughput with Flickr dataset.

Table 2. Compression Ratios of the Key-value Pairs of Different Sizes

Item Size	**64 B**	**128 B**	**256 B**	**512 B**	**1 KB**	**2 KB**	**4 KB**	**8 KB**	**16 KB**
Compression Ratio	0.98	1.00	1.03	1.07	1.12	1.13	1.19	1.37	1.37

4.3 Cache Server Performance

In this section, we study the performance details of the cache server by generating GET/SET requests directly to the cache server. Since we focus on testing the raw cache server capabilities, there is no backend database server in this set of experiments, if not otherwise specified, and we load about 30 GB data using the Twitter dataset to populate the cache server, and generate 10M GET/SET requests with the Zipfian distribution for the test. All the experiments use 8 key-value cache servers and 32 clients.

4.3.1 Compression Granularity. We first study the effect of compression granularity. Table 2 shows the average compression ratio of fixed-size key-value pairs generated by Markov text generator [26] when compressed individually with lz4. In the following experiments, we compare our proposed dynamic compression granularity with static compression in three large granularities, 4, 8, and 16 KB, which achieve the highest compression ratios as shown in the table.

Figures 15 and 16 show the throughput and the average latency of the workload with a GET/SET ratio of 95:5. We vary the fixed-size compression granularity from 4 to 16 KB, as comparison to our dynamically adjusted approach (see Section 3). It shows that by limiting the size of the compressed items in one flash page, the throughput can be significantly higher than those spreading over multiple flash pages. For example, when the value size is 128 Bytes, if the compression granularity is 16 KB, then the throughput is 34K ops/s, and it increases to 51K ops/s by using our dynamic method. The improvement is as high as 50%. Figure 15 also shows that the throughput of the dynamic mechanism is always among the top two and is close to the highest static setting. Figure 16 shows a similar trend. Using dynamic compression granularity, we can achieve both high compression ratio and high throughput simultaneously. Compared to static setting, our dynamic configuration approach achieves a similar performance, and is more flexible and adapts to the workloads during runtime.

4.3.2 Hot/Cold Data Separation. Figure 17 compares the throughput with and without the hot area for the Twitter dataset with the Zipfian distribution. As shown in the figure, the throughput when SET/GET ratio is 0:100 is 39K and 65K ops/s for SlimCache without and with hot/cold data separation, respectively. Thus, a 66.7% improvement can be achieved with hot/cold separation.

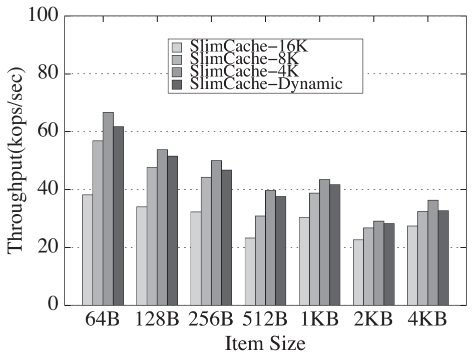


Fig. 15. Throughput vs. granularity.

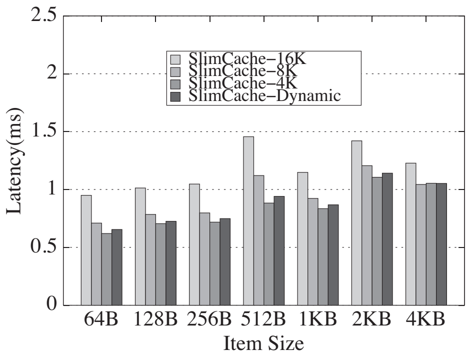


Fig. 16. Latency vs. granularity.

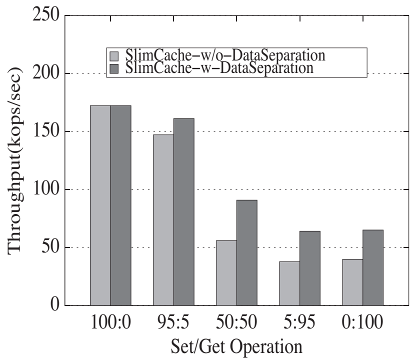


Fig. 17. Hot/cold data separation.

Such an improvement can also be seen with other SET/GET ratios, but when all the requests are SET operations, the two mechanisms achieve almost the same throughput. That is because the SET path in SlimCache is identical, no matter the data separation is enabled or not—The items are all batched together and written to the cold area in the compressed format. However, the difference emerges when GET operations are involved, because the hot items are promoted to the hot area in uncompressed format, and the following GET requests to this item can avoid the unnecessary overhead. Although the hot area only accounts for a small percentage of the cache space, it improves the performance significantly compared to that without hot/cold separation.

We note that such a great performance improvement is not for free. Frequent data movement between the hot and cold areas may cause a write amplification problem, which is harmful to the performance and also the lifetime of flash. In our experiments, we find that the Write Amplification Factor (WAF) is up to 4.2% in SlimCache, meaning that only 4.2% of the write requests is caused by the switch between the two areas. Since the WAF is quite low and the hot/cold data switch is a background operation, the benefit introduced by hot/cold data separation clearly outweighs its overhead, as shown in Figure 17.

4.3.3 Two-stage Garbage Collection. We test the effect of the high watermark W_{high} and low watermark W_{low} to the performance by setting the high watermark from 8 to 128 free slabs, and the low watermark half of the high watermark. For the Twitter dataset following different distributions, the performance is insensitive to the watermark settings. This is because the reserved free space only accounts for a very small portion of the entire cache space as shown in Figure 18 and

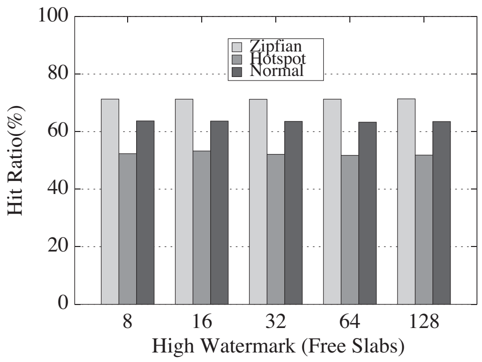


Fig. 18. Hit ratio vs. watermark.

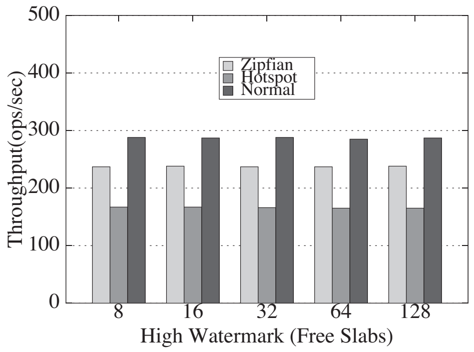


Fig. 19. Throughput vs. watermark.

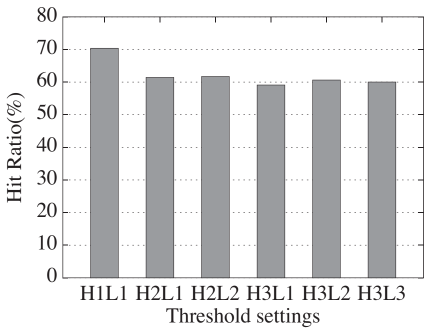


Fig. 20. Threshold settings in GC.

Figure 19. In our experiments, we set $W_{high} = 16$, $W_{low} = 8$ and $W_{start} = 2$, which is only about 1% of the overall cache space.

4.3.4 Data Recycling. We investigate the effect of threshold setting for hot, warm and cold data identification during garbage collection, with 300 million requests following the Zipfian distribution. The cache size is set 6% of the workload dataset size.

Figure 20 shows the hit ratio change by setting different thresholds. When the high threshold and the low threshold are both 1 (denoted as H1L1 in the figure), which means that the items will be promoted to the hot area when they are reaccessed at least once and all the rest are dropped directly, the hit ratio reaches the highest, 70.4%, among all the settings. When we vary the threshold settings, the hit ratio drops to about 60%. It indicates that recycling hot data to the hot area is very effective to identify the most valuable data. However, recycling warm data to the cold area incurs inefficient recollection, since many of the recollected warm data are not frequently reaccessed but occupy the cache space that could be used for other valuable items. Based on the experimental results, we simplify the garbage collection process without recycling warm data to the cold area. Instead, only hot items are promoted to the hot area.

Table 3 shows the percentage of `GET` requests that are served from the hot area when the high threshold and the low threshold are both 1. With a `SET/GET` ratio of 5:95, 56.7% of the `GET` requests fall in the hot area, whose size is only 5% of the entire cache space. These results show that the hot/cold data separation can effectively alleviate the read amplification problem caused by on-line compression.

Table 3. Ratio of GET Requests Served in the Hot Area

SET/GET	**95:5**	**50:50**	**5:95**	**0:100**
SlimCache	79.1%	87.3%	56.7%	55%

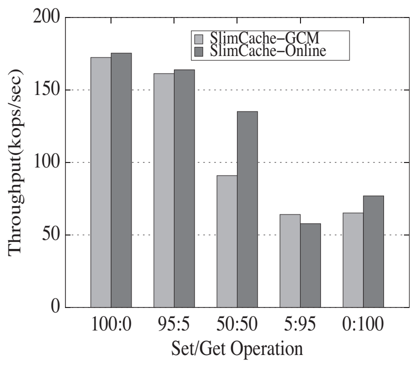


Fig. 21. Online vs. GCM promotion.

Table 4. Hit Ratio of Online and GCM Promotion

Scheme	**Zipfian**	**Hotspot**	**Normal**
Fatcache	65.1%	25.2%	32%
SlimCache-Online	69.5%	38.2%	47%
SlimCache-GCM	70.2%	45.4%	52.8%

4.3.5 Garbage-Collection-Merged Promotion. We compare two different promotion approaches. The first one is on-line promotion, which moves the items to the hot area in the uncompressed format immediately after this item is re-accessed. The second one is called Garbage Collection Merged (GCM) promotion, which is used in GC in SlimCache (see Section 3.6). In the GCM promotion, re-accessed items are promoted to the hot area during the GC period. Neither of the two approaches causes extra read overhead, since the on-demand read requests or the embedded GC process needs to read the items or the slab anyway. However, these two methods have both advantages and disadvantages. On-line promotion is prompt, but it wastes extra space, because the original copy of the promoted items would not be recycled until the slab is reclaimed, reducing the usable cache space and harming the hit ratio. On the contrary, the GCM promotion postpones the promotion until the GC process happens, but it does not cause space waste, which is crucial for caching.

As Figure 21 shows, when we test the server without considering the backend database server, the on-line promotion shows a relatively better performance than the GCM promotion, because the on-line compression can timely promote a frequently accessed item into the hot area, reducing the decompression overhead.

However, the on-line promotion approach could create duplicate copies in cache. It would incur a waste of cache space, reducing the hit ratio and causing a negative performance impact. In contrast, GCM removes such issues and maintains a higher hit ratio. Table 4 shows the effect of such a space waste on the hit ratio. We have repeated the Twitter experiments in Section 4.2.1 and set the cache

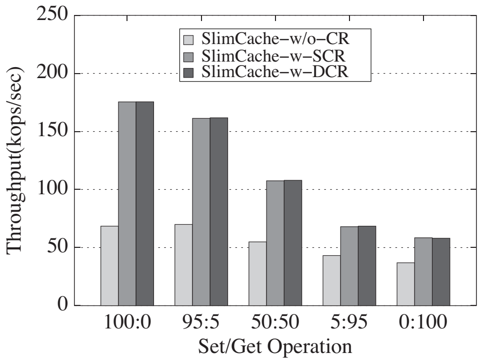


Fig. 22. DCR with incompressible data.

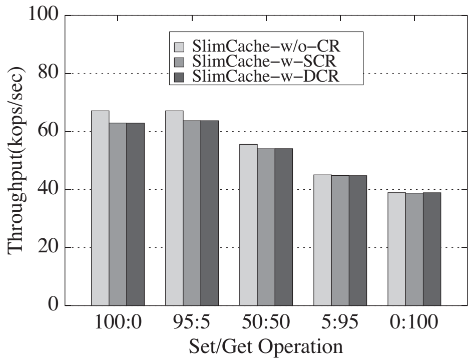


Fig. 23. DCR with Twitter data.

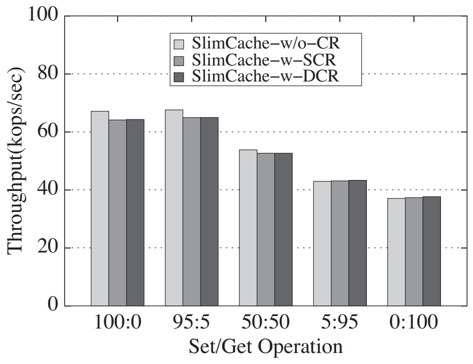


Fig. 24. DCR with hybrid data.

size as 6% of the dataset size. It shows that SlimCache-GCM provides a hit ratio increase of 0.7–7.2 p.p. over SlimCache-Online, which would correspondingly translate into performance gains in cases when a backend database is involved. As space saving for hit ratio improvement is the main goal of SlimCache, we choose GCM in SlimCache. This highly integrated garbage collection and hot/cold data switch process is specifically customized for flash-based caching systems, with significant performance improvement.

4.3.6 Dynamic Compressibility Recognition (DCR). The dynamic compressibility recognition (DCR) can bring both benefits and overhead. For incompressible data, it can reduce significant overhead by skipping the compression process. However, for compressible data, the compressibility check incurs additional overhead.

We have benchmarked the effect of DCR with Zipfian workloads. Figure 22 shows the benefit of applying compressibility recognition to the incompressible dataset, which is composed of randomly generated characters. In particular, compressibility recognition improves the throughput by up to 156.1%. In contrast, the DCR mechanism adds overhead for the compressible Twitter dataset, as shown in Figure 23. We can also see that the overhead is mainly associated with SET operations. When the GET operations are dominant, which is typical in key-value cache systems, the overhead is minimal. Figure 24 shows the effect of DCR when the workload is a hybrid compressible (Twitter) and incompressible (random) dataset at the ratio of 1:1. The overhead introduced by DCR is negligible when the dataset is hybrid. Compared to Static Compressibility Recognition (SCR),

Table 5. Parameters Used in Dynamic Partition Mechanism I

Scheme	Latency (μs)
Hot area read	400
Cold area read	900
Back-end fetch	300,000

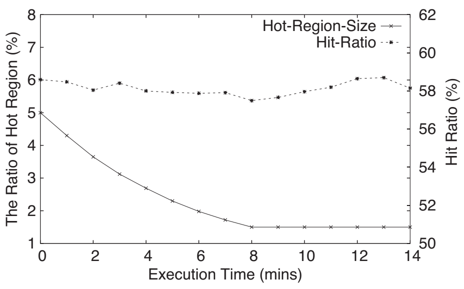


Fig. 25. Adaptive partitioning.

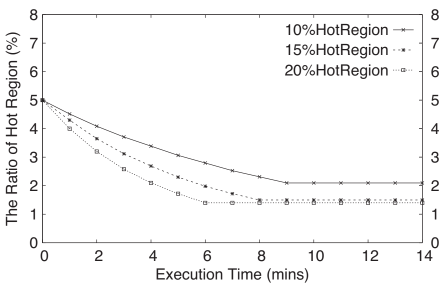


Fig. 26. Step S for partitioning.

DCR provides very close performance as shown in Figures 22, 23, and 24. However, DCR provides a more user-friendly interface than SCR, since determining the proper compression ratio is more straightforward than determining the proper entropy. Our results show that the DCR mechanism generally incurs little overhead for the read-intensive compressible data and improves throughput significantly for incompressible data. It is also worth noting that such an automatic approach avoids the need for involving human efforts in determining whether to apply data compression or not, which is particularly important for an online system handling key-value requests at a high throughput.

4.4 Adaptive Partitioning

4.4.1 High Miss Cost. To illustrate the adaptive partitioning, we collect the average read latency to configure our proposed cost model. The hot area cache read is measured 400 μs, the cold area cache read is 900 μs, and the backend fetch is 300 ms. The parameters are listed in Table 5.

Figure 25 shows the runtime hot area size and the hit ratio when dynamic partitioning happens when the miss cost is high. As the speed of our backend database is slow, SlimCache tends to keep a larger cold area and attempts to reduce the number of cache misses until the convergence condition is reached. Figure 25 shows that the hit ratio keeps stable when data migration happens in SlimCache. We have also studied the effect of step S by setting it to 10%, 15%, and 20% of the hot area size. SlimCache can reach a stable cache partitioning within 9 min for all the step settings as Figure 26 shows. Considering that the up-time of a real server is often long (days to months), such a short time for reaching a stable cache partitioning means that our adaptive partitioning approach is reasonably responsive and effective.

4.4.2 Low Miss Cost. In this section, we configure the hot area cache read time to be 400 μs, the cold area cache read time to be 900 μs, and the backend access read time to be 1.2 ms to illustrate the adaptive partitioning when the backend access cost is low. The parameters are listed in Table 6.

Figure 27 shows the runtime hot area size and the hit ratio, when dynamic partitioning happens in the situation of dealing with a fast backend data store. In this case, as the speed of our backend

Table 6. Parameters Used in Dynamic Partition Mechanism II

Scheme	Latency (μs)
Hot area read	400
Cold area read	900
Back-end fetch	1,200

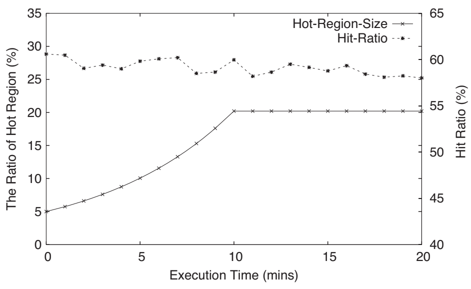


Fig. 27. Adaptive partitioning.

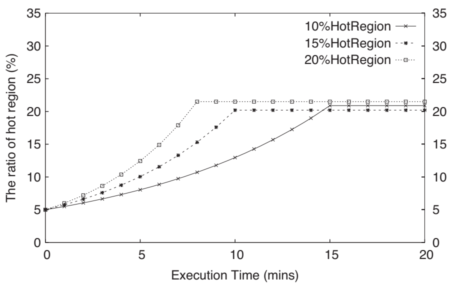


Fig. 28. Step *S* for partitioning.

database is only 33% slower than the cache server, SlimCache gradually expands the hot area (to about 20%) to reduce the incurred compression overhead until the convergence condition is reached. This is because with a fast backend data store, the relative benefit of maintaining a large amount of compressed data in the cold area decreases. Figure 27 also shows that the hit ratio keeps stable, when data migration happens in SlimCache. Similar to the high-miss cost case, we also study the effect of step *S* by setting it to 10%, 15%, and 20% of the hot area size. It takes less than 15 min for SlimCache to reach a stable cache partitioning for all the step settings as Figure 28 shows. Together with the results from Section 4.4.1, we can see that no matter the backend database is relatively fast or slow, our adaptive partitioning approach can reach a reasonably good partition quickly.

4.5 Overhead Analysis

SlimCache introduces on-line compression in flash-based key-value caching, which could increase the consumption of CPU, memory, and flash resources on the server side.

- **Memory utilization.** In SlimCache, memory is mainly used for three purposes. (1) In-memory hash table. SlimCache adds a 1-bit *access_count* attribute to record the access count of the item since stored in the system. (2) Slab buffer. SlimCache performance is not sensitive to the memory buffer. We maintain a 128 MB memory for slab buffer, which is identical to Fatcache. (3) Slab metadata. We add a 1-bit attribute for each slab, called *hotslab*. This bit indicates whether the slab belongs to the hot area or not. In total, for a 1 TB SSD that stores 1 billion records, SlimCache consumes about 128 MB (128 MB for hash table entry metadata, 128 KB for slab metadata) more memory than Fatcache, which is about 0.3% of the overall memory consumption. In our experiments, we find that the actual memory consumption of SlimCache and Fatcache is similar, when caching the same amount of key-value items.
- **CPU utilization.** SlimCache is multi-threaded. In particular, we maintain one thread for the drain operation, one thread for garbage collection, one thread for data movement between the hot and the cold areas, and one thread for dynamic partitioning. Compression

Table 7. CPU Utilization of SlimCache

Scheme	Zipfian		Hotspot		Normal	
Cache	6%	12%	6%	12%	6%	12%
Fatcache	1.93%	2.08%	1.07%	1.19%	1.84%	2.25%
SlimCache	2.09%	2.14%	1.23%	2.21%	2.05%	3.37%

and decompression operations also consume CPU cycles. As shown in Table 7, the CPU utilization of SlimCache is less than 3.5% in all our experiments. The main bottleneck is the backend database for the whole system. Computation resource is sufficient on the cache server to complete the demanded work.

- **Flash utilization.** We add a 1-bit *compressed* attribute to each key-value item to indicate whether the item is in compressed format or not. This attribute is used to determine if a decompression process should be applied when the slot is read upon a `GET` operation. Storing 1 billion records will consume 128 MB more flash space, which is a small storage overhead.

5 LIMITATIONS

Although SlimCache can achieve significantly better performance than Fatcache, there are still several limitations that are out of the scope of this work and worth studies in the future.

5.1 Data Persistence

As a replacement of Memcached, Twitter's Fatcache is not designed to guarantee data persistence in cache. In Fatcache, the mapping structure is completely stored in volatile memory rather than the flash SSD. This design choice removes the related performance overhead and also simplifies the system design, but when a system crash or power failure happens, the key-value data hosted in the flash cache would become invalid. It is worth noting that it is still safe, because the client can always find a copy of the most up-to-date data in the backend data store. However, the cache system has to warm up again after restart, which could take a long duration. Similar to the stock Fatcache, SlimCache shares the same limitation in terms of cache data persistence. A potential solution is to keep the mapping structure is byte-addressable persistent memory, such as Intel's Optane Memory [37]. Our main goal in this work is to improve the caching performance and efficiency by adopting data compression techniques in cache management, as demonstrated in SlimCache. As an orthogonal challenge, the cache data persistence issue is worth further studies in the future.

5.2 Flash Durability

Since SlimCache divides the logical space of the flash SSD into a hot area and a cold area, the data swapping between the two areas could potentially result in *write amplification*, causing additional amount of writes to the flash SSD and accelerating the wear-out process of the caching device. We find that because of the strict yet effective data swapping policy, the write amplification factor is observed to be about 4.2% in our experiments. Considering the significant performance gain of SlimCache and the continuously decreasing price of flash SSDs, such a relatively small write amplification is considered to be fairly acceptable to most users. Measures, such as using a flash-based RAID, can also be adopted to reduce the concerns on the durability. It is worth future studies on further reducing the impact of flash durability in such scenarios.

6 RELATED WORK

This article and its earlier version [38] present a highly efficient on-line data compression scheme for enhancing flash-based key-value caching. The two topics, data compression [1, 6, 17, 19, 64, 76, 81, 84] and key-value systems [5, 18, 46, 48, 51, 73, 83, 86], have been extensively researched. This section discusses prior studies that are most related to these components.

Data compression is a popular technique. In prior works, extensive studies have been conducted on compressing memory and storage at both architecture and system levels, such as device firmware [36, 89], storage controller [34], and operating systems [2, 6, 19, 53, 76, 81]. Many prior works have also be done in database systems (e.g., References [59, 1, 16, 44, 60, 64]). Our work focuses on applying data compression to improve the hit ratio by caching more key-value items in flash. To our best knowledge, SlimCache is the first work introducing data compression into flash-based key-value caching.

Recent research on key-value cache focuses mostly on performance improvement [47, 49, 54], such as network request handling, OS kernel involvement, data structure design, and concurrency control, and so on. Recently, hardware-centric studies [50], such as FPGA-based design [7], Open-Channel SSD [73], and programmable NIC [46], began to explore the hardware features. In particular, DIDACache [73] provides a holistic flash-based key-value cache using Open-Channel SSD through a deep integration between hardware and software. KV-Direct [46] presents a high performance key-value system through remote direct key-value access to the host memory by extending the RDMA primitives based on programmable NIC. Similarly, NetCache [40] optimizes the queries to hot key-value items and attempts to balance the load across storage nodes by leveraging the flexibility of new programmable switches. Memshare [15] gives a DRAM-based key-value cache system with a dynamic memory management across applications. To reduce small random writes in photo caching, RIPQ [74] provides a framework to support advanced cache replacement algorithms with optimized writes on flash devices by collecting small writes, flushing updates lazily, and grouping similar data together. For a similar purpose, Flashield [20, 21] gives a hybrid solution by using DRAM to filter and reduce writes to flash, which addresses the write amplification problem on flash SSDs. In comparison, SlimCache adopts a largely orthogonal approach, data compression, to improve the flash-based key-value cache performance and efficiency.

Besides performance, some other studies deal with the scalability problem [23, 61, 62, 82], which results from hardware cost and power/thermal issues. For example, Nishtala et al. aim to scale Memcached to handle large amount of Internet traffic in Facebook [61]. Ouyang et al. design an SSD-assisted hybrid memory for Memcached to achieve high performance and low cost [62]. McDipper [23] is a flash-based key-value cache solution to replace Memcached in Facebook. BlueCache [85] proposes to address the scalability challenges by implementing all the key-value operations including the flash controller operations directly in hardware. Anna [82] is a partitioned, multi-mastered key-value system that can effectively scale from a single core to multi-core to the distributed system via wait-free execution and coordination-free consistency. As a scale-up solution, Tucana [63] presents an efficient and high-speed key-value store design for achieving both high performance and low CPU overhead. Cascade Mapping [79] provides a new index mapping structure in key-value caches to address the scalability challenge caused by limited memory resources.

With the popularity of persistent memory, a number of studies [10, 22, 32, 41, 42, 55, 79] have been proposed to integrate non-volatile memory (NVM) within key-value systems. Huang et al. [32] propose to use cross-referencing logs to close the performance gap between the key-value stores in volatile DRAM and persistent NVM. NVMKV [55] optimizes flash-based key-value stores through techniques, such as alleviating dynamic mapping, providing transaction support, and leveraging parallelization. NoveLSM [41] proposes an LSM-tree-based design of persistent key-value store by taking advantage of the byte addressability and persistence features of non-volatile memories, such as creating a byte-addressable skiplist, directly manipulating persistent state, and exploiting opportunistic read parallelism, and so on. HashKV [10] is designed to achieve high update performance based on KV separation and using hash-based data grouping. MyNVM [22] reduces the DRAM footprint of Facebook's key-value store by replacing DRAM with NVM. uDepot [42] presents a key-value design, which exploits the performance of NVM devices with a two-level indexing structure and a new task-based IO run-time system. In contrast, the data compression scheme presented in this article and its earlier version [38] is a general-purpose software-level solution without relying on any special hardware.

Among the prior works, zExpander [84], which applies compression in memory-based key-value caches, is the closest to our work. However, SlimCache is particularly designed for key-value caching in flash, which brings several different and unique challenges. First, small random writes are particularly harmful to the lifetime and performance of flash devices, so storing and querying an item using a small-size (2 KB) block on SSD as what zExpander does would be sub-optimal in our scenario. Second, as the amount of key-value items stored in flash-based key-value cache is much larger than that in a memory-based cache, the organization unit has to be much coarser and the metadata overhead brought by each item must be minimized. Third, choosing a proper compression granularity on flash needs to consider the flash page size to minimize the extra I/Os caused by loading irrelevant data. Finally, to guarantee that all the writes are sequential in flash, the space occupied by the obsolete values in one slab cannot be freed until the whole slab is dropped. A special mechanism is needed to handle such situations to avoid the loss of hit ratio caused by data promotion and demotion while preserving the sequential write pattern. All these distinctions and new challenges have motivated us to design an efficient, on-line data compression scheme, customized for caching key-value data in flash.

7 CONCLUSION

In this article, we present an on-line compression mechanism for flash-based key-value cache systems, called SlimCache, which expands the effectively usable cache space, increases the hit ratio, and improves the cache performance. For optimization, SlimCache introduces a number of techniques, such as unified management for compressed and uncompressed data, dynamically determining compression granularity, efficient hot/cold data separation, optimized garbage collection, and adaptive cache partitioning. Our experiments show that SlimCache can effectively accommodate more key-value data in cache, which in turn significantly increases the cache hit ratio and improves the system performance.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive feedback and insightful comments.

REFERENCES

[1] Daniel Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating compression and execution in column-oriented database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'06).*

[2] Bulent Abali, Mohammad Banikazemi, Xiawei Shen, Hubertus Franke, Dan E. Poff, and T. Basil Smith. 2001. Hardware compressed main memory: Operating system support and performance evaluation. *IEEE Trans. Comput.* 50, 11 (Nov. 2001), 1219–1233. DOI: https://doi.org/10.1109/12.966496

[3] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. 2008. Design tradeoffs for SSD performance. In *Proceedings of the USENIX Annual Technical Conference (USENIXATC'08).*

[4] Zhongqi An, Zhengyu Zhang, Qiang Li, Jing Xing, Hao Du, Zhan Wang, Zhigang Huo, and Jie Ma. 2017. Optimizing the datapath for key-value middleware with NVMe SSDs over RDMA interconnects. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER'17).* 582–586. DOI: https://doi.org/10.1109/CLUSTER.2017.69

[5] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'12)*.

[6] Vicenc Beltran, Jordi Torres, and Eduard Ayguad. 2008. Improving web server performance through main memory compression. In *Proceeding of the 14th IEEE International Conference on Parallel and Distributed Systems (ICPADS'08)*.

[7] Michaela Blott, Kimon Karras, Ling Liu, Kees Vissers, Jeremia Baer, and Zsolt Istvan. 2013. Achieving 10Gbps line-rate key-value stores with FPGAs. In *Proceedings of the 5th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'13)*.

[8] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. 1999. Web caching and zipf-like distributions: Evidence and implications. In *Proceedings of the 18th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOMM'99)*.

[9] Damiano Carra and Pietro Michiard. 2014. Memory partitioning in memcached: An experimental performance analysis. In *Proceedings of the IEEE International Conference on Communications (ICC'14)*.

[10] Helen H. W. Chan, Yongkun Li, Patrick P. C. Lee, and Yinlong Xu. 2018. HashKV: Enabling efficient updates in KV storage via hashing. In *Proceedings of the USENIX Annual Technical Conference (USENIXATC'18)*. USENIX Association, 1007–1019. Retrieved from https://www.usenix.org/conference/atc18/presentation/chan.

[11] Feng Chen, Binbing Hou, and Rubao Lee. 2016. Internal parallelism of flash memory-based solid-state drives. *ACM Trans. Stor.* 12, 3 (May 2016), 13:1–13:39.

[12] Feng Chen, David Koufaty, and Xiaodong Zhang. 2009. Understanding intrinsic characteristics and system implications of flash memory-based solid state drives. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS/Performance'09)*.

[13] Feng Chen, Rubao Lee, and Xiaodong Zhang. 2011. 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)*.

[14] Seonghyeog Choi and Euiseong Seo. 2017. A selective compression scheme for in-memory cache of large-scale file systems. In *Proceedings of the International Conference on Electronics, Information, and Communication (ICEIC'17)*.

[15] Asaf Cidon, Daniel Rushton, Stephen M. Rumble, and Ryan Stutsman. 2017. Memshare: A dynamic multi-tenant key-value cache. In *Proceedings of 2017 USENIX Annual Technical Conference (USENIXATC'17)*. USENIX Association, 321–334. Retrieved from https://www.usenix.org/conference/atc17/technical-sessions/presentation/cidon.

[16] Gordon V. Cormack. 1985. Data compression on a database system. *Commun. ACM* 28, 12 (Dec. 1985), 1336–1342.

[17] Rodrigo S. de Castro, Alair Pereira do Lago, and Dilma Da Silva. 2003. Adaptive compressed caching: Design and implementation. In *Proceedings of the 15th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'03)*.

[18] Biplob Debnath, Sudipta Sengupta, and Jin Li. 2011. SkimpyStash: RAM space skimpy key-value store on flash-based storage. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'11)*.

[19] DoromNakar and Shlomo Weiss. 2004. Selective main memory compression by identifying program phase changes. In *Proceedings of the 23rd IEEE Convention of Electrical and Electronics Engineers in Israel*.

[20] Assaf Eisenman, Asaf Cidon, Evgenya Pergament, Or Haimovich, Ryan Stutsman, Mohammad Alizadeh, and Sachin Katti. 2017. Flashield: A key-value cache that minimizes writes to flash. Retrieved from http://arxiv.org/abs/1702.02588.

[21] Assaf Eisenman, Asaf Cidon, Evgenya Pergament, Or Haimovich, Ryan Stutsman, Mohammad Alizadeh, and Sachin Katti. 2019. Flashield: A hybrid key-value cache that controls flash write amplification. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI'19)*. USENIX Association, 65–78. Retrieved from https://www.usenix.org/conference/nsdi19/presentation/eisenman.

[22] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. 2018. Reducing DRAM footprint with NVM in Facebook. In *Proceedings of the 13th EuroSys Conference (EuroSys'18)*. ACM, New York, NY. DOI: https://doi.org/10.1145/3190508.3190524

[23] Facebook. 2013. McDipper: A Key-value Cache for Flash Storage. Retrieved from https://www.facebook.com/notes/facebook-engineering/mcdipper-a-key-value-cache-for-flash-storage/10151347090423920/.

[24] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud computing (SoCC'10)*.

[25] Annie Foong and Frank Hady. 2016. Storage as fast as rest of the system. In *Proceedings of the IEEE 8th International Memory Workshop (IMW'16)*. DOI: https://doi.org/10.1109/IMW.2016.7495289

[26] Python Software Foundation. 2019. Text Generator based on Markov Chain. Retrieved from https://pypi.python.org/pypi/markovgen/0.5.

[27] Kingwa Fu. 2017. Weiboscope Open Data. Retrieved from https://hub.hku.hk/cris/dataset/dataset107483.

[28] Kingwa Fu, C. H. Chan, and Michael Chau.2013. Assessing censorship on microblogs in China: Discriminatory keyword analysis and impact evaluation of the real name registration policy. *IEEE Internet Comput.* 17, 3 (2013), 42–50.
[29] GNU. 2018. Gzip. Retrieved from https://www.gnu.org/software/gzip/.
[30] Google. 2019. Snappy. Retrieved from https://github.com/google/snappy.
[31] Danny Harnik, Ronen Kat, Oded Margalit, Dmitry Sotnikov, and Avishay Traeger. 2013. To zip or not to zip: Effective resource usage for real-time compression. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*.
[32] Yihe Huang, Matej Pavlovic, Virendra Marathe, Margo Seltzer, Tim Harris, and Steve Byan. 2018. Closing the performance gap between volatile and persistent key-value stores using cross-referencing logs. In *Proceedings of the USENIX Annual Technical Conference (USENIXATC'18)*. USENIX Association, 967–979. Retrieved from https://www.usenix.org/conference/atc18/presentation/huang.
[33] Mark J. Huiskes and Michael S. Lew. 2008. The MIR flickr retrieval evaluation. In *Proceedings of the ACM International Conference on Multimedia Information Retrieval (MIR'08)*.
[34] IBM. 2015. IBM Real-time Compression in IBM SAN Volume Controller and IBM Storwize V7000. Retrieved from http://www.redbooks.ibm.com/redpapers/pdfs/redp4859.pdf.
[35] Intel. 2012. Optane SSD. Retrieved from https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/consumer-ssds/optane-ssd-9-series/optane-ssd-900p-series/900p-280gb-aic-20nm.html.
[36] Intel. 2018. Intel SSD. Retrieved from https://www.intel.com/content/www/us/en/support/articles/000006354/memory-and-storage.html.
[37] Intel. 2020. Intel Optane Memory. Retrieved from https://www.intel.com/content/www/us/en/architecture-and-technology/optane-memory.html.
[38] Yichen Jia, Zili Shao, and Feng Chen. 2018. SlimCache: Exploiting data compression opportunities in flash-based key-value caching. In *Proceedings of the IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'18)*. IEEE, 209–222.
[39] Song Jiang, Feng Chen, and Xiaodong Zhang. 2005. CLOCK-Pro: An effective improvement of the CLOCK replacement. In *Proceedings of the USENIX Annual Technical Conference (USENIXATC'05)*.
[40] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*. ACM, New York, NY, 121–136. DOI: https://doi.org/10.1145/3132747.3132764
[41] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. Redesigning LSMs for nonvolatile memory with NoveLSM. In *Proceedings of the USENIX Annual Technical Conference (USENIXATC'18)*. USENIX Association, 993–1005. Retrieved from https://www.usenix.org/conference/atc18/presentation/kannan.
[42] Kornilios Kourtis, Nikolas Ioannou, and Ioannis Koltsidas. 2019. Reaping the performance of fast NVM storage with uDepot. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*. USENIX Association, 1–15. Retrieved from https://www.usenix.org/conference/fast19/presentation/kourtis.
[43] Sanjeev R. Kulkarni. 2002. *Information, Entropy, and Coding*. Lecture Notes for ELE201 Introduction to Electrical Signals and Systems, Princeton University, 2002.
[44] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data blocks: Hybrid OLTP and OLAP on compressed storage using both vectorization and compilation. In *Proceedings of the International Conference on Management of Data (SIGMOD'16)*.
[45] Larry Leemis. 2019. Zipf. Retrieved from http://www.math.wm.edu/ leemis/chart/UDR/PDFs/Zipf.pdf.
[46] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. 2017. KV-direct: High-performance in-memory key-value store with programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*. ACM, New York, NY, 137–152. DOI: https://doi.org/10.1145/3132747.3132756
[47] Sheng Li, Hyeontaek Lim, Victor W. Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, David G. Andersen, Seongil O, Sukhan Lee, and Pradeep Dubey. 2015. Architecting to achieve a billion requests per second throughput on a single key-value store server platform. In *Proceeding of the 42nd ACM/IEEE Annual International Symposium on Computer Architecture (ISCA'15)*.
[48] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. 2011. SILT: A memory-efficient, high-performance key-value store. In *Proceedings of the 23th Symposium on Operating Systems Principles (SOSP'11)*.
[49] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: A holistic approach to fast in-memory key-value storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI'14)*.
[50] Kevin Lim, David Meisner, Ali G. Saidi, Parthasarathy Ranganathan, and Thomas F. Wenisch. 2013. Thin servers with smart pipes: Designing SoC accelerators for memcached. In *Proceeding of the 40th Annual International Symposium on Computer Architecture (ISCA'13)*.

[51] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. WiscKey: Separating keys from values in SSD-conscious storage. In *Proceedings of 14th USENIX Conference on File and Storage Technologies (FAST'16)*.

[52] lz4. 2019. Extremely Fast Compression. Retrieved from http://lz4.github.io/lz4/.

[53] Thanos Makatos, Yannis Klonatos, Manolis Marazakis, Michail D. Flouris, and Angelos Bilas. 2010. Using transparent compression to improve SSD-based I/O caches. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys'10)*.

[54] Yandong Mao, Eddie Kohler, and Robert Morris. 2012. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys'12)*.

[55] Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, and Raju Rangaswami. 2015. NVMKV: A scalable, lightweight, FTL-aware key-value store. In *Proceedings of the USENIX Annual Technical Conference (USENIX-ATC'15)*. USENIX Association, 207–219. Retrieved from https://www.usenix.org/conference/atc15/technical-session/presentation/marmol.

[56] Venkata Lakshmi Marripudi and P. Yakaiah. 2015. Image compression based on multilevel thresholding image using shannon entropy for enhanced image. *Global J. Adv. Eng. Technol.* 4, 3 (2015), 271–274.

[57] N. Megiddo and D. Modha. 2003. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of the 2nd USENIX Conference on File and Storage (FAST'03)*.

[58] Memcached. 2018. A Distributed Memory Object Caching System. *Retrieved from https:// memcached.org/*.

[59] MongoDB. 2019. WiredTiger Storage Engine. Retrieved from https://docs.mongodb.com/manual/core/wiredtiger/.

[60] Ingo Muller, Cornelius Ratsch, and Franz Faerber. 2014. Adaptive string dictionary compression in in-memory column-store database systems. In *Proceedings of the 17th International Conference on Extending Database Technology (EDBT'14)*.

[61] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013. Scaling memcache at Facebook. In *Proceeding of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)*.

[62] Xiangyong Ouyang, Nusrat S. Islam, Raghunath Rajachandrasekar, Jithin Jose, Miao Luo, Hao Wang, and Dhabaleswar K. Panda. 2012. SSD-assisted hybrid memory to accelerate memcached over high performance networks. In *Proceeding of the 41st International Conference on Parallel Processing (ICPP'12)*.

[63] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. 2016. Tucana: Design and implementation of a fast and efficient scale-up key-value store. In *Proceedings of the USENIX Annual Technical Conference (USENIXATC'16)*. USENIX Association, 537–550. Retrieved from https://www.usenix.org/conference/atc16/technical-sessions/presentation/papagiannis.

[64] Meikel Poess and Dmitry Potapov. 2003. Data compression in Oracle. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB'03)*.

[65] Reddit. 2015. Reddit Comments. Retrieved from https://www.reddit.com/r/datasets/comments/3bxlg7/i_have_every_publicly_available_reddit_comment/.

[66] Greg Roelofs, Jean-loup Gailly, and Mark Adler. 2017. Zlib. Retrieved from https://zlib.net/.

[67] Samsung. 2017. Ultra-Low Latency with Samsung Z-NAND SSD—Breakthrough Storage for a New Generation of Enterprise and Data Center Infrastructure. Retrieved from https://www.samsung.com/semiconductor/global.semi.static/Brochure_Samsung_S-ZZD_SZ985_1804.pdf.

[68] Samsung. 2018. Samsung Z-SSD SZ985 - Ultra-low Latency SSD for Enterprise and Data Centers—Brochure. Retrieved from https://www.samsung.com/semiconductor/global.semi.static/Brochure_Samsung_S-ZZD_SZ985_1804.pdf.

[69] Bon-Keun Seo, Seungryoul Maeng, Joonwon Lee, and Euiseong Seo. 2015. DRACO: A deduplicating FTL for tangible extra capacity. *IEEE Comput. Architect. Lett.* 14, 2 (July 2015), 123–126.

[70] Dimitrios N. Serpanos and Wayne H. Wolf. 1998. Caching web objects using Zipf's law. *Proc. SPIE* 3527 (Oct. 1998), 320–326.

[71] Dipti Shankar, Xiaoyi Lu, Md Rahman, Nusrat Islam, and D. K. Panda. 2015. Benchmarking key-value stores on high-performance storage and interconnects for web-scale workloads. In *Proceedings of the IEEE International Conference on Big Data (BigData'15)*. 539–544. DOI: https://doi.org/10.1109/BigData.2015.7363797

[72] C. E. Shannon. 1948. A mathematical theory of communication. *Bell Syst. Tech. J.* 27 (1948), 379–423.

[73] Zhaoyan Shen, Feng Chen, Yichen Jia, and Zili Shao. 2017. 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)*.

[74] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. 2015. RIPQ: Advanced photo caching on flash for Facebook. In *Proceedings of 13th USENIX Conference on File and Storage Technologies (FAST'15)*. USENIX Association, 373–386. Retrieved from https://www.usenix.org/conference/fast15/technical-sessions/presentation/tang.

[75] Luca Trevisan. 2012. Kolmogorov Complexity. Retrieved from http://cs.stanford.edu/~trevisan/cs154-12/kolcomplexity-rev.pdf.

[76] Irina Chihaia Tuduce and Thomas Gross. 2005. Adaptive main memory compression. In *Proceedings of the USENIX Annual Technical Conference (USENIXATC'05)*.

[77] Twitter. 2011. Tweets2011. Retrieved from http://trec.nist.gov/data/tweets/.

[78] Twitter. 2013. Fatcache. Retrieved from https://github.com/twitter/fatcache.

[79] Kefei Wang and Feng Chen. 2018. Cascade mapping: Optimizing memory efficiency for flash-based key-value caching. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC'18)*. ACM, New York, NY, 464–476. DOI: https://doi.org/10.1145/3267809.3267847

[80] Wikipedia. 2019. Entropy (Information Theory). Retrieved from https://en.wikipedia.org/wiki/Entropy_(information_theory).

[81] Paul R. Wilson, Scott F. Kaplan, and Yannis Smaragdakis. 1999. The case for compressed caching in virtual memory systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'99)*.

[82] Chenggang Wu, Jose M. Faleiro, Yihan Lin, and Joseph M. Hellerstein. 2018. Anna: A KVS for any scale. In *Proceedings of IEEE 34th International Conference on Data Engineering (ICDE'18)*. 401–412. DOI: https://doi.org/10.1109/ICDE.2018.00044

[83] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. 2015. LSM-trie: An LSM-tree-based ultra-large key-value store for small data items. In *Proceedings of the USENIX Annual Technical Conference (USENIXATC'15)*. USENIX Association, 71–82. https://www.usenix.org/conference/atc15/technical-session/presentation/wu.

[84] Xingbo Wu, Li Zhang, Yandong Wang, Yufei Ren, Michel H. Hack, and Song Jiang. 2016. zExpander: A key-value cache with both high performance and fewer misses. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys'16)*.

[85] Shuotao Xu, Sungjin Lee, Sang-Woo Jun, Ming Liu, Jamey Hicks, and Arvind. 2016. Bluecache: A scalable distributed flash-based key-value store. *Proc. VLDB Endow.* 10, 4 (Nov. 2016), 301–312. DOI: https://doi.org/10.14778/3025111.3025113

[86] Heng Zhang, Mingkai Dong, and Haibo Chen. 2016. Efficient and available in-memory KV-store with hybrid erasure coding and replication. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*.

[87] Pin Zhou, Vivek Pandey, Jagadeesan Sundaresan, Anand Raghuraman, Yuanyuan Zhou, and Sanjeev Kumar. 2004. Dynamically tracking miss-ratio-curve for memory management. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'04)*.

[88] George Kingsley Zipf. 1929. Relative frequency as a determinant of phonetic change. *Reprinted from the Harvard Studies in Classical Philology* XL (1929).

[89] Aviad Zuck, Sivan Toledo, Dmitry Sotnikov, and Danny Harnik. 2104. Compression and SSD: Where and how? In *Proceedings of the 2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW'14)*.

Received March 2019; revised July 2019; accepted February 2020