SlimCache: Exploiting Data Compression Opportunities in Flash-based Key-value Caching

Yichen Jia
Computer Science and Engineering
Louisiana State University
yjia@csc.lsu.edu

Zili Shao
Computer Science and Engineering
The Chinese University of Hong Kong
shao@cse.cuhk.edu.hk

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

***Abstract*—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 paper, 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 leveraged the unique workload characteristics in key-value systems to efficiently identify and separate hot and cold data. In order to dynamically adapt to workload changes during runtime, we have designed an adaptive hot/cold area partitioning method based on a cost model. In order 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 125.9%, effectively increasing throughput and reducing average latency by up to 255.6% and 78.9%, respectively.**

I. Introduction

Today's data centers still heavily rely on hard disk drives (HDDs) as their main storage devices. To address the performance problem of disk drives, especially for handling random accesses, in-memory key-value cache systems, such as Memcached [37], become popular in data centers for serving various applications [20], [48]. 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 high 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 [20], as a replacement of the expensive Memcached servers. Twitter has a similar key-value cache solution, called Fatcache [48].

A. Motivations

The traditional focus on improving the caching efficiency is to develop sophisticated cache replacement algorithms [36], [26]. 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 [36] and CLOCK-Pro [26], 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 [7], [15]. 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 further 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) [48]. 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 paper 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

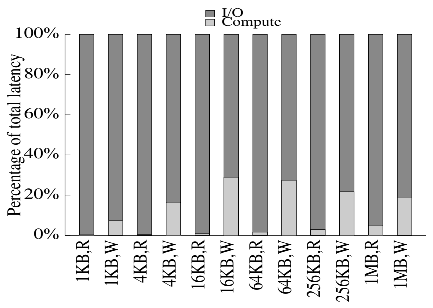


Fig. 1: I/O time v.s. computation time

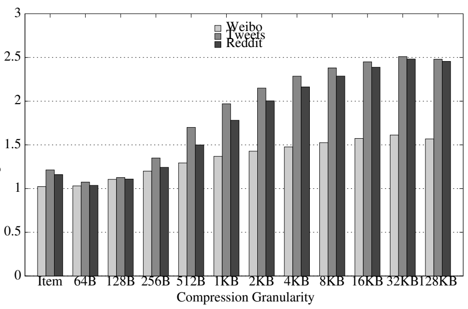


Fig. 2: Compr. ratio v.s. granularity.

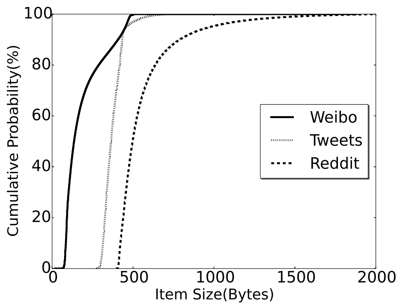


Fig. 3: Distribution of item sizes.

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 flash. Compared to schemes compressing data in memory, such as zExpander [54], 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.

B. 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 [3], [5], [32]. Lightweight algorithms, such as `lz4` [32] and `snappy` [3], are fast, but only provide moderate compression ratio (i.e., $\frac{uncompressed}{compressed}$); heavyweight schemes, such as the deflate algorithm used in `gzip` [2] and `zlib` [5], 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 paper 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.

C. Our Solution: SlimCache

In this paper, we present an adaptive on-line compression scheme for key-value caching in flash, called *SlimCache*. SlimCache identifies the key-value items that are suitable for compression, applies a compression and decompression algorithm at a proper granularity, and expands 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 [8], 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 the runtime workloads. In order 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 [48]. Our experimental evaluations on an Intel 910 PCI-E SSD have shown that SlimCache can accommodate more key-value items in the cache by up to 125.9%, effectively increasing throughput and reducing average latency by up to 255.6% and 78.9%, respectively. Such an improvement is essential for data-intensive applications in data centers.

The rest of this paper is organized as follows. Section II gives the background and related work. Section III introduces the design of SlimCache. Section IV gives our experimental results. The other related work is presented in Section V. The final section concludes this paper.

II. Background and related work

A. 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 [7], [14], [15], [16].

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 [48] 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.

B. Related Work

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 [1], [57], storage controller [25], and operating systems [9], [19], [33], [47], [53]. Much prior works have also be done in database systems (e.g., [4], [6], [18], [28], [38], [41]). Our work focuses on applying data compression to improve the hit ratio of caching key-value data in flash. To our best knowledge, SlimCache is the first work introducing data compression into flash-based key-value caching.

Among the related work, zExpander [54], which applies compression in memory-based key-value caches, is the closest to our work. SlimCache is particularly designed for key-value caching in flash, which brings several different and unique challenges. First, small random writes are particularly harmful for the lifetime and performance of flash devices, so storing and querying an item using a small-size (2KB) 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, in order 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.

III. Design of SlimCache

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

A. Overview

We present a comprehensive on-line compression scheme for flash-based key-value caching, called *SlimCache*. As shown in Figure 4, 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,

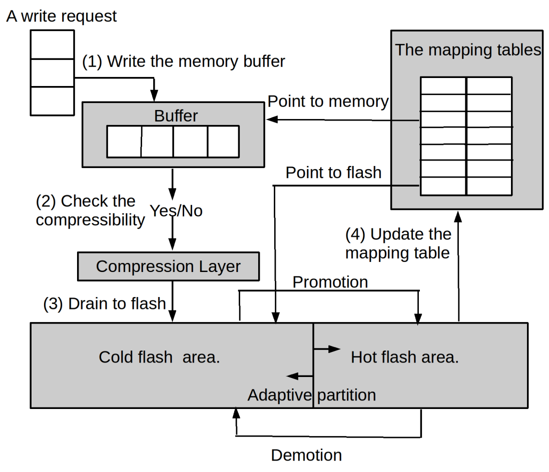


Fig. 4: An illustration of SlimCache architecture.

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 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.

B. 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 a 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. The slab I/O management is unnecessary 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 runtime, while disregarding the details of storing and transferring data.

C. 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 [8], 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 3, around 80% of items in the three datasets, Weibo [50], [51], Twitter [49] and Reddit [42], are under 288 bytes, 418 bytes 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` [32], 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 increases, considering that a flash page read is typically 25-100 μs. So, compressing data in slabs would cause concerns on the overhead issues.

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 at runtime, and calculate an average compression ratio, $avg\_compression\_ratio$. The estimated compression container size is calculated as $flash\_page\_size \times avg\_compression\_ratio$, where $flash\_page\_size$ is the known flash page size (typically 4-16 KB), and must be no smaller than a memory page size (4KB as default). The rationale behind this is that we desire to keep the compression algorithm having 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). In our experiments, we have particularly studied the effect of compression granularity on the performance of SlimCache in Section IV-C1.

D. Hot/Cold Data Separation

In order 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 adaptively in Section III-E.

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. In order 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 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 III-F, 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. We will study this effect in Section IV-C3.

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—in order 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 released. 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, random writes would be generated to flash.

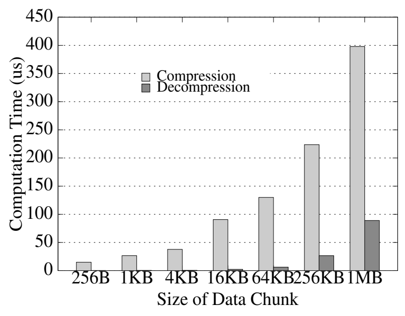


Fig. 5: Compression time vs. unit size.

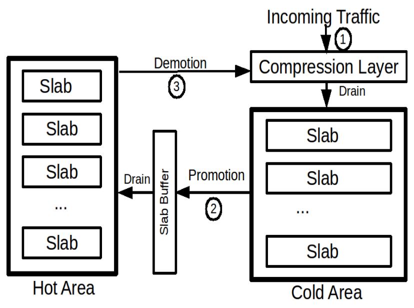


Fig. 6: Hot and cold data separation

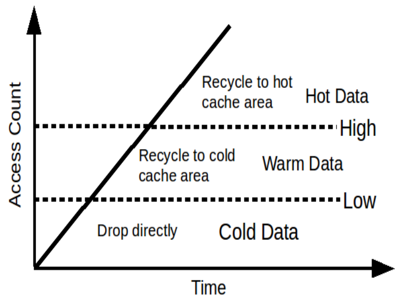


Fig. 7: Data recycling in garbage collection.

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 III-F), 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.

For demotion, when the hot area is full, the Least Recently Used (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 IV-C2).

E. Adaptive Partitioning

As mentioned above, the partitioning of flash space determines the portion of key-value items being stored in the compressed or uncompressed format. The larger the cold region is, the more flash space could be saved, and the higher hit ratio would be, but the more I/Os have to experience a time-consuming decompression. Thus, we need to provide a partitioning scheme being adaptive to the change of workloads. Here we present a solution based on a simple cost model to achieve such an adaptive partitioning.

Initializing partitions. If we assume the workload distribution follows the true Zipf's law [56], [12], which has $\alpha = 1$, for a system with 100 million items, a 5% cache can handle approximately about 85% of the requests, according to prior cache analysis [11], [44]. Thus in our prototype, we optimistically set the hot area initially as 5% of the flash space and 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.

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

H_{hot} and H_{cold} are the ratios of hits contributed by the hot (uncompressed) key-value items and the cold (compressed) 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. C_{miss} is the cost of fetching data from the backend data store. These parameters can be obtained through runtime measurement. 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, 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, 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, we keep the current partitioning unchanged. If the predicted cost after increasing or decreasing the hot area is lower, we enlarge or reduce the hot area size, accordingly.

The above-said model is simple yet effective. Other models, such as miss ratio curve [55], 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.

F. Garbage Collection

Garbage collection is a must-have process. 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 through garbage collection.

Traditional garbage collection directly drops all the items, including the valid data, to reclaim free space. SlimCache deploys a recycling mechanism in garbage collection as shown in Figure 7. Based on the access_count, the key-value items can be divided into three categories: *hot*, *warm* and *cold*. Accordingly, we can 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. After modifying the hash table mappings, the whole slab is placed back to the free cold slab list. This garbage collection procedure collects and promotes valuable items for retaining a high hit ratio. We will study the effect of threshold settings for hot, warm, and cold data identification in Section IV-C3.

Our proposed garbage collection process is integrated with the hot/cold data management. The data promotion and demotion happen only when the garbage collection process is triggered, which effectively removes the undesirable waste of cache space, as discussed in Section III-D.

G. 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 [45], 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 [23], [27]. Entropy has been widely used for testing data compressibility in various scenarios, such as primary storage [23], memory cache [17], device firmware [43], image compression [35], and many others. We use *normalized entropy* [52], which is the entropy divided by the maximum entropy ($\log_b n$), to quickly filter out the incompressible data, which are implied by a high entropy value (greater than 0.95). The items that are detected incompressible are directly written to the cold area in their original uncompressed format. Thus 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 compressibility recognition in Section IV-C5.

H. 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 of SlimCache.

IV. Evaluation

To evaluate the proposed schemes, we have implemented a prototype of SlimCache based on Twitter's Fatcache [48]. Our implementation accounts for about 2,700 lines of code in C. In this section, we will evaluate the SlimCache design on a real SSD hardware platform.

A. Experimental Setup

Our experiments are conducted on three Lenovo ThinkServers. All the three servers feature an Intel Xeon(R) 3.40GHz CPU and 16GB memory. In the key-value cache server, an 800GB Intel 910 PCI-E SSD is used as the storage device for key-value caching. Note that for a fair comparison, only 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 page cache. Our backend data store is MongoDB v3.4.4 running on a separate server with 1TB Seagate 7200RPM hard drive. The clients run on another ThinkServer to generate traffic to drive the experiments. The three servers are connected via a 10Gbps Ethernet switch. For all the three servers, we use Ubuntu 14.04 with Linux kernel 4.4.0-31 and Ext4 file system in the experiments.

We use Yahoo's YCSB benchmark suite [21] to generate workloads to access the key-value items, following three different distributions (Zipfian, Normal, and Hotspot) as described in prior work [13], [54] to simulate typical traffic in cloud services [8]. Since the YCSB workloads do not contain actual data, we use the datasets from Twitter [49] and Flickr [24] to emulate two typical types of key-value data with different compressibility. The Twitter dataset has a high compression ratio (about 2), while the Flickr dataset has a low compression ratio, nearly 1 (incompressible). In order to generate fixed-size compressible values (Section IV-C1), we use the text generator [22] based on Markov chain provided by Python to generate the pseudo-random fixed-size values. We use `lz4` [32] and the deflate method in `zlib` [5] for compression in comparison.

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.

B. 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 (480 million and 2 million records for Twitter and Flicker, respectively), so note that only part of the 800GB SSD capacity is used as cache (12-24 GB). 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 30 minutes 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.

1) Performance for Twitter Dataset: Our on-line compression solution can "virtually" enlarge the size of the cache space. Figure 8, 10, and 9 show the number of items cached in SlimCache compared to the stock Fatcache with the same amount of flash space. As shown in Figure 8, 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. Figure 11, 12, and 13 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 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 Figure 14, 15, and 16, 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 Figure 17, 18, and 19.

TABLE I: 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%

To further understand the reason of the performance gains, we repeated the experiments with compression disabled. Table I 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 in 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.

2) 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 dataset. Figure 20 shows that `zlib` performs the best among the three, while `lz4` and `snappy` are almost identical. 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%. 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.

3) 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 21 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.

C. 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 30GB data to populate the cache server, and generate 10 million `GET`/`SET` requests for the test. All the experiments use 8 key-value cache servers and 32 clients.

1) Compression Granularity: We first study the effect of compression granularity. Figure 22 and Figure 23 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 KB to 16 KB, as comparison to our dynamically adjusted approach (see Section III). 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, the throughput is 34K ops/sec, and it increases to 51K ops/sec by using our dynamic method. The improvement is as high as

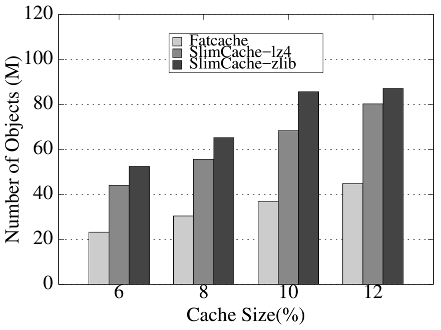


Fig. 8: Num. of objects, Zipfian

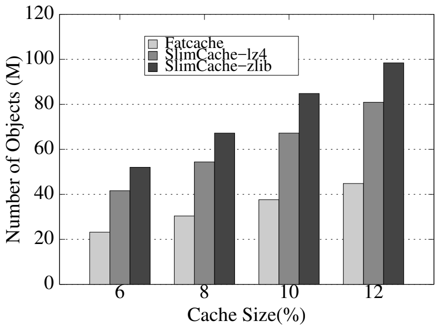


Fig. 9: Num. of objects, Hotspot

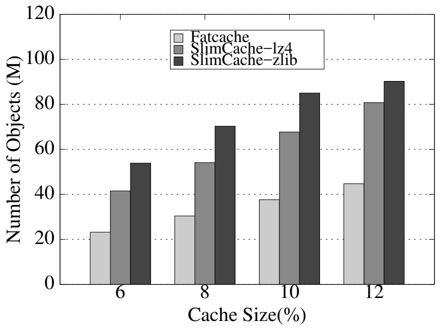


Fig. 10: Num. of objects, Normal

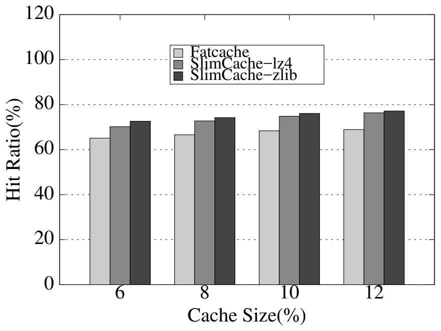


Fig. 11: Hit Ratio (%), Zipfian

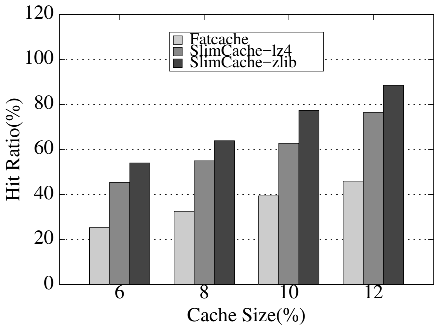


Fig. 12: Hit Ratio (%), Hotspot

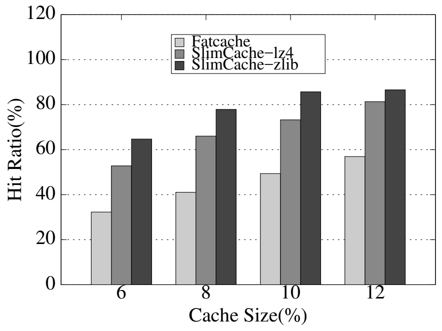


Fig. 13: Hit Ratio (%), Normal

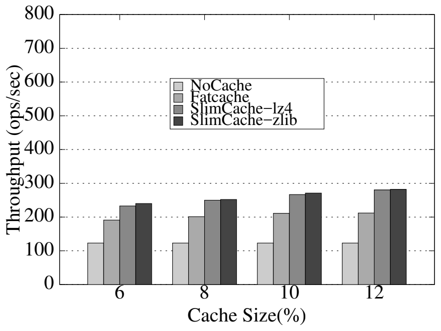


Fig. 14: Throughput (OPS), Zipfian

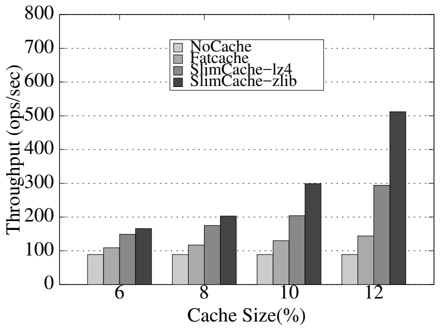


Fig. 15: Throughput (OPS), Hotspot

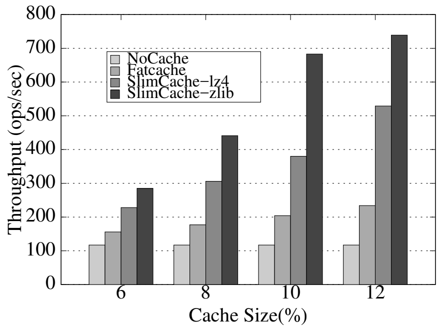


Fig. 16: Throughput (OPS), Normal

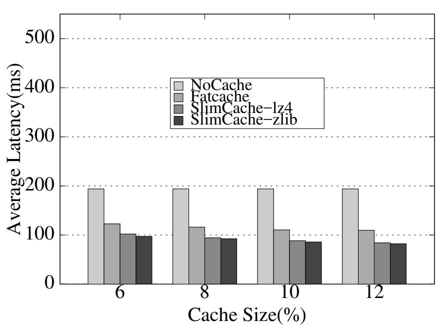


Fig. 17: Average latency (ms), Zipfian

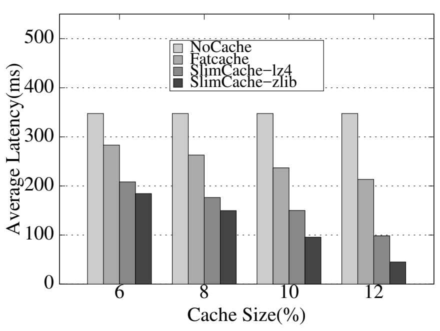


Fig. 18: Average latency (ms), Hotspot

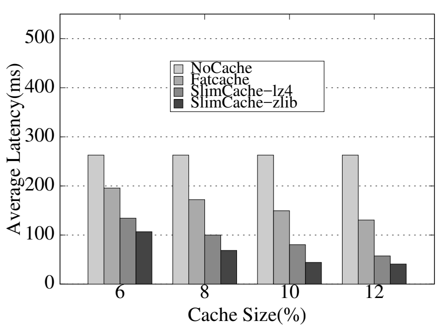
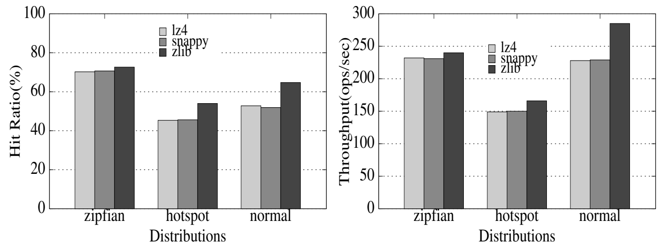


Fig. 19: Average latency (ms), Normal

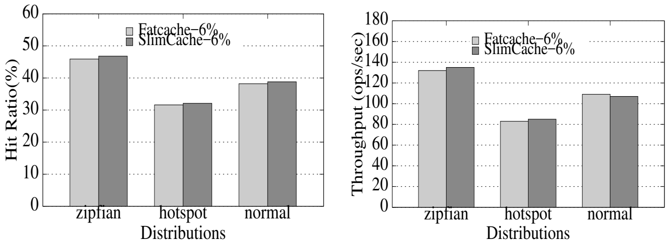
50%. Figure 22 also shows that the throughput of the dynamic mechanism is always among the top two and is close to the highest static setting. Figure 23 shows a similar trend.

2) Hot/Cold Data Separation: Figure 24 compares the throughput with and without the hot area for the Zipfian-distributed Twitter dataset. As shown in the figure, the throughput of `GET` operations is 39K ops/sec and 65K ops/sec for SlimCache without and with hot/cold data separation, respectively (66.7% improvement). 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 throughputs. 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



(a) Hit ratio (%) (b) Throughput (IOPS)

Fig. 20: Effect of different compression algorithms.



(a) Hit ratio (%) (b) Throughput (OPS)

Fig. 21: Hit ratio and throughput with Flickr dataset.

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 for the performance and also the lifetime of flash. In our experiments, we find that the write amplification factor (WAF) is 4.2% in SlimCache, meaning that only 4.2% of the traffic 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 24.

3) Garbage Collection: We investigate the effect of threshold setting for hot, warm and cold data identification during garbage collection, with 300 million requests following Zipfian distribution. The cache size is set 6% of the workload dataset size. Figure 25 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 II 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 II: 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%

4) 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 III-F). 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 26 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 duplicate copies would incur a waste of cache space. Table III shows the effect of such a space waste on the hit ratio. We have repeated the Twitter experiments in Section IV-B1 and set the cache 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 system, where it shows significant performance improvement.

TABLE III: 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%

5) Compressibility Recognition (CR): The compressibility recognition (CR) can bring both benefits and overhead. For incompressible data, it can reduce significant overhead by skipping the compression process. However, for compressible

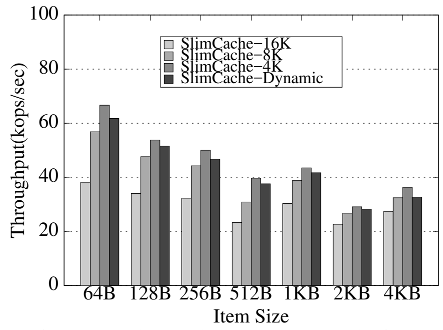


Fig. 22: Throughput vs. granularity

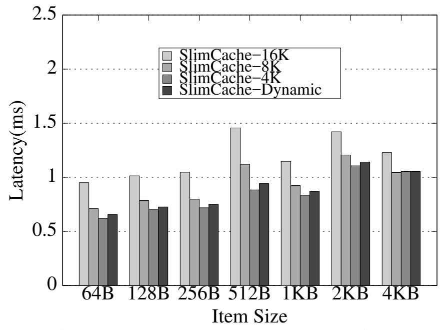


Fig. 23: Latency vs. granularity

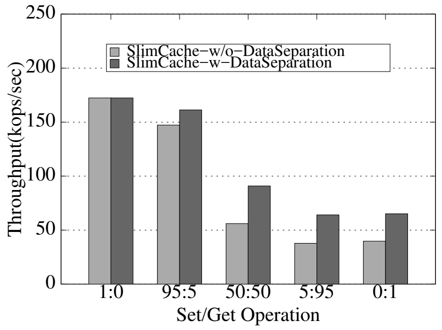


Fig. 24: Hot/cold data separation

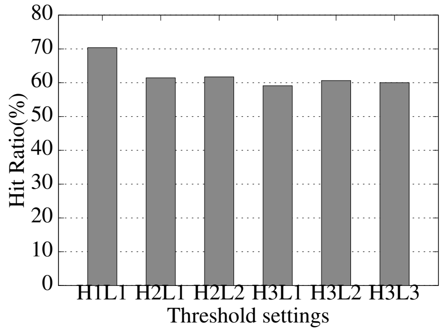


Fig. 25: Threshold settings in GC

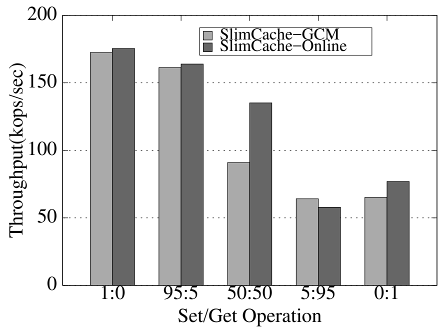


Fig. 26: Online vs. GCM promotion.

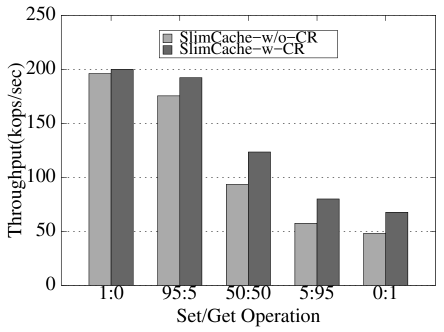


Fig. 27: CR with incompressible data

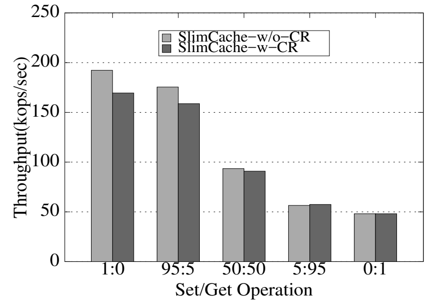


Fig. 28: CR with compressible data

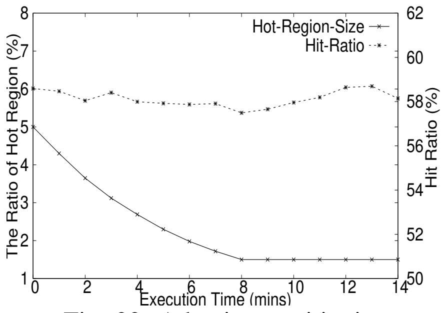


Fig. 29: Adaptive partitioning

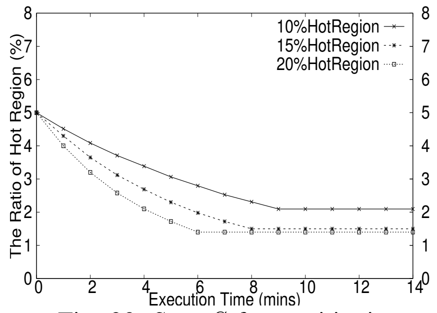


Fig. 30: Step S for partitioning

data, the compressibility check incurs additional overhead. Figure 27 shows the benefit of applying compressibility recognition to the incompressible Flickr dataset. In particular, compressibility recognition improves the throughput by up to 40.5%. In contrast, the CR mechanism adds overhead for the compressible Twitter dataset, as shown in Figure 28. We also can 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. In our prototype, SlimCache provides an interface for users to decide whether to enable compressibility recognition or not, according to the workload. Our results show that the CR mechanism is generally more efficient than compressing all the data indistinctively.

D. Adaptive Partitioning

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. Figure 29 shows the runtime hot area size and the hit ratio when dynamic partitioning happens. 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 29 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 minutes for all the step settings as Figure 30 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.

E. Overhead Analysis

SlimCache introduces on-line compression in flash-based key-value cache, which could increase the consumption of CPU, memory and flash resources on the sever 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 1TB 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. In our experiments, we find that the actual memory consumption of SlimCache and Fatcache is similar at runtime.

• **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 and decompression operations also consume CPU cycles. As shown in Table IV, 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.

TABLE IV: 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%

• **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.

V. Other Related Work

This section discusses the other prior studies related to this work. Key-value caching has attracted high interests in both academia and industry. Recent research on key-value cache focuses mostly on performance improvement [29], [30], [34], such as network request handling, OS kernel involvement, data structure design, and concurrency control, etc. Recently hardware-centric studies [31], such as FPGA-based design [10] and Open-Channel SSD [46], began to explore the hardware features. In particular, DIDACache [46] provides a holistic flash-based key-value cache using Open-Channel SSD through a deep integration between hardware and software. Besides the performance, some other studies deal with the scalability problem [20], [39], [40], which results from hardware cost and power/thermal problems. For example, Nishtala et al. aim to scale Memcached to handle large amount of Internet traffic in Facebook [39]. Ouyang et al. design an SSD-assisted hybrid memory for Memcached to achieve high performance and low cost [40]. McDipper [20] is a flash-based key-value cache solution to replace Memcached in Facebook. Our SlimCache is a general-purpose software-level solution without relying on any special hardware. It enhances cache performance through data compression and is orthogonal to these prior optimizations.

VI. Conclusions

In this paper, 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 optimizations, 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 our design 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. The work described in this paper was supported in part by the Research Grants Council of the Hong Kong Special Administrative Region, China under Grants GRF 152223/15E, GRF 152736/16E, and GRF 152066/17E, in part by Louisiana Board of Regents under Grant LEQSF(2014-17)-RD-A-01, and in part by the U.S. National Science Foundation under Grants CCF-1453705 and CCF-1629291.

References

[1] https://www.intel.com/content/www/us/en/support/articles/000006354/memory-and-storage.html.
[2] Gzip. http://www.gzip.org/.
[3] Snappy. https://github.com/google/snappy.
[4] WiredTiger Storage Engine. https://docs.mongodb.com/manual/core/wiredtiger/.
[5] Zlib. https://zlib.net/.
[6] D. Abadi, S. Madden, and M. Ferreira. Integrating Compression and Execution in Column-oriented Database Systems. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD'06)*, Chicago, IL, USA, June 26-29 2006.
[7] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design Tradeoffs for SSD Performance. In *Proceedings of the 2008 USENIX Annual Technical Conference (USENIX ATC'08)*, Boston, MA, June 22-27 2008.
[8] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of a Large-Scale Key-Value Store. In *Proceedings of 2012 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'12)*, London, UK, June 11-15 2012.
[9] V. Beltran, J. Torres, and E. Ayguad. Improving Web Server Performance Through Main Memory Compression. In *Proceeding of the 14th IEEE International Conference on Parallel and Distributed Systems (ICPADS'08)*, Melbourne, VIC, Australia, December 8-10 2008.
[10] M. Blott, K. Karras, L. Liu, K. Vissers, J. Baer, and Z. Istvan. Achieving 10Gbps Line-rate Key-value Stores with FPGAs. In *Proceeding of the 5th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'13)*, San Jose, CA, June 25-26 2013.

[11] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. On the Implications of Zipf's Law for Web Caching. In *Proceedings of the 3rd International WWW Caching Workshop*, Manchester, England, June 15-17 1998.

[12] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. 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)*, New York, NY, March 21-25 1999.

[13] D. Carra and P. Michiard. Memory Partitioning in Memcached: An Experimental Performance Analysis. In *Proceedings of 2014 IEEE International Conference on Communications (ICC'14)*, Sydney, Austrilia, June 2014 10-14.

[14] F. Chen, B. Hou, and R. Lee. Internal Parallelism of Flash Memory-Based Solid-State Drives. *ACM Transactions on Storage*, 12(3):13:1–13:39, May 2016.

[15] F. Chen, D. A. Koufaty, and X. Zhang. Understanding Intrinsic Characteristics and System Implications of Flash Memory based Solid State Drives. In *Proceedings of 2009 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS/Performance'09)*, Seattle, WA, June 15-19 2009.

[16] F. Chen, R. Lee, and X. Zhang. Essential Roles of Exploiting Internal Parallelism of Flash Memory based Solid State Drives in High-speed Data Processing. In *Proceedings of the 17th International Symposium on High Performance Computer Architecture (HPCA'11)*, San Antonio, Texas, February 12-16 2011.

[17] S. Choi and E. Seo. A Selective Compression Scheme for In-Memory Cache of Large-Scale File systems. In *Proceedings of 2017 International Conference on Electronics, Information, and Communication (ICEIC'17)*, Phuket, Thailand, January 11 2017.

[18] G. V. Cormack. Data Compression on a Database System. *Communications of the ACM*, 28(12):1336–1342, Dec. 1985.

[19] DoromNakar and S. Weiss. Selective Main Memory Compression by Identifying Program Phase Changes. In *Proceedings of the 23rd IEEE Convention of Electrical and Electronics Engineers in Israel*, Tel-Aviv, Isreal, September 6-7 2004.

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

[21] B. F.Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing (SoCC'10)*, Indianapolis, Indiana, June 10-11 2010.

[22] P. S. Foundation. Text Generator based on Markov Chain. https://pypi.python.org/pypi/markovgen/0.5.

[23] D. Harnik, R. Kat, O. Margalit, D. Sotnikov, and A. Traeger. 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)*, San Jose, February 12-15 2013.

[24] M. J. Huiskes and M. S. Lew. The MIR Flickr Retrieval Evaluation. In *Proceedings of the 2008 ACM International Conference on Multimedia Information Retrieval (MIR'08)*, Vancouver, Canada, October 30-31 2008.

[25] IBM. IBM Real-time Compression in IBM SAN Volume Controller and IBM Storwize V7000. http://www.redbooks.ibm.com/redpapers/pdfs/redp4859.pdf.

[26] S. Jiang, F. Chen, and X. Zhang. CLOCK-Pro: An Effective Improvement of the CLOCK Replacement. In *Proceedings of 2005 USENIX Annual Technical Conference (USENIX ATC'05)*, Anaheim, CA, April 10-15 2005.

[27] S. R. Kulkarni. *Information, Entropy, and Coding*. Lecture Notes for ELE201 Introduction to Electrical Signals and Systems, Princeton University, 2002.

[28] H. Lang, T. Mhlbauer, F. Funke, P. Boncz, T. Neumann, and A. Kemper. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD'16)*, San Francisco, CA, June 26-July 01 2016.

[29] S. Li, H. Lim, V. W. Lee, J. H. Ahn, A. Kalia, M. Kaminsky, D. G. Andersen, S. O, S. Lee, and P. Dubey. 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)*, Portland, OR, June 13-17 2015.

[30] H. Lim, D. Han, D. G.Andersen, and M. Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI'14)*, Seattle, WA, April 2-4 2014.

[31] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch. Thin Servers with Smart Pipes: Designing SoC Accelerators for Memcached. In *Proceeding of the 40th Annual International Symposium on Computer Architecture (ISCA'13)*, Tel Aviv, Israel, June 23-27 2013.

[32] lz4. Extremely fast compression. http://lz4.github.io/lz4/.

[33] T. Makatos, Y. Klonatos, M. Marazakis, M. D. Flouris, and A. Bilas. Using Transparent Compression to Improve SSD-based I/O Caches. In *Proceeding of the 5th European conference on Computer systems (EuroSys'10)*, Paris, France, April 13-16 2010.

[34] Y. Mao, E. Kohler, and R. Morris. Cache Craftiness for Fast Multicore Key-Value Storage. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys'12)*, Bern, Switzerland, April 10-13 2012.

[35] V. L. Marripudi and P. Yakaiah. Image Compression based on Multilevel Thresholding Image Using Shannon Entropy for Enhanced Image. *Global Journal of Advanced Engineering Technologies*, 4:271–274, 2015.

[36] N. Megiddo and D. Modha. ARC: A Self-tuning, Low Overhead Replacement Cache. In *Proceedings of the 2nd USENIX Conference on File and Storage (FAST'03)*, San Francisco, CA, February 12-15 2003.

[37] Memcached. A Distributed Memory Object Caching System. https://memcached.org/.

[38] I. Mller, C. Ratsch, and F. Faerber. Adaptive String Dictionary Compression in In-Memory Column-Store Database Systems. In *Proceedings of the 17th International Conference on Extending Database Technology (EDBT'14)*, Athens, Greece, March 24-28 2014.

[39] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *Proceeding of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)*, Lombard, IL, April 2-5 2013.

[40] X. Ouyang, N. S. Islam, R. Rajachandrasekar, J. Jose, M. Luo, H. Wang, and D. K. Panda. SSD-Assisted Hybrid Memory to Accelerate Memcached over High Performance Networks. In *Proceeding of the 41st International Conference on Parallel Processing (ICPP'12)*, Pittsburgh, PA, September 10-13 2012.

[41] M. Poess and D. Potapov. Data Compression in Oracle. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB'03)*, Berlin, Germany, September 9-12 2003.

[42] Reddit. Reddit Comments. https://www.reddit.com/r/datasets/comments/3bxlg7/i_have_every_publicly_available_reddit_comment/.

[43] B.-K. Seo, S. Maeng, J. Lee, and E. Seo. DRACO: A Deduplicating FTL for Tangible Extra Capacity. *IEEE Computer Architecture Letters*, 14:123–126, July-December 2015.

[44] D. N. Serpanos and W. H. Wolf. Caching Web objects using Zipf's law. *Proceedings of the SPIE*, 3527:320–326, October 1998.

[45] C. E. Shannon. A Mathematical Theory of Communication. *Bell System Technical Journal*, 27:379–423, 1948.

[46] Z. Shen, F. Chen, Y. Jia, and Z. Shao. DIDACache: A Deep Integration of Device and Application for Flash-based Key-value Caching. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*, Santa Clara, CA, February 27-March 2 2017.

[47] I. C. Tuduce and T. Gross. Adaptive Main Memory Compression. In *Proceedings of 2005 USENIX Annual Technical Conference (USENIX ATC'05)*, Anaheim, CA, April 10-15 2005.

[48] Twitter. Fatcache. https://github.com/twitter/fatcache.

[49] Twitter. Tweets2011. http://trec.nist.gov/data/tweets/.

[50] K. wa Fu. Weiboscope Open Data. https://hub.hku.hk/cris/dataset/dataset107483.

[51] K. wa Fu, C. Chan, and M. Chau. Assessing Censorship on Microblogs in China: Discriminatory Keyword Analysis and Impact Evaluation of the Real Name Registration Policy. *IEEE Internet Computing*, 17(3):42–50, 2013.

[52] Wikipedia. Entropy (Information Theory). https://en.wikipedia.org/wiki/Entropy_(information_theory).

[53] P. R. Wilson, S. F. Kaplan, and Y. Smaragdakis. The Case for Compressed Caching in Virtual Memory Systems. In *Proceedings of the 1999 USENIX Annual Technical Conference (USENIX ATC'99)*, Monterey, CA, June 6-11 1999.

[54] X. Wu, L. Zhang, Y. Wang, Y. Ren, M. HHack, and S. Jiang. zExpander: a Key-Value Cache with Both High Performance and Fewer Misses. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys'16)*, London, UK, April 18-21 2016.

[55] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. 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)*, Boston, MA, October 7-13 2004.

[56] G. K. Zipf. Relative Frequency as a Determinant of Phonetic Change. *Harvard Studies in Classical Philology*, 40:1–95, 1929.

[57] A. Zuck, S. Toledo, D. Sotnikov, and D. Harnik. Compression and SSD: Where and How? In *Proceedings of the 2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW'14)*, Broomfield, CO, Oct 2104.