zExpander: a Key-Value Cache with both High Performance and Fewer Misses

Xingbo Wu, Li Zhang*, Yandong Wang*, Yufei Ren*, Michel Hack*, Song Jiang
Wayne State University, *IBM T. J. Watson Research Center
{wuxb,sjiang}@wayne.edu, {zhangli,yandong,yren,hack}@us.ibm.com

Abstract

While key-value (KV) cache, such as `memcached`, dedicates a large volume of expensive memory to holding performance-critical data, it is important to improve memory efficiency, or to reduce cache miss ratio without adding more memory. As we find that optimizing replacement algorithms is of limited effect for this purpose, a promising approach is to use a compact data organization and data compression to increase effective cache size. However, this approach has the risk of degrading the cache's performance due to additional computation cost. A common perception is that a high-performance KV cache is not compatible with use of data compacting techniques.

In this paper, we show that, by leveraging highly skewed data access pattern common in real-world KV cache workloads, we can both reduce miss ratio through improved memory efficiency and maintain high performance for a KV cache. Specifically, we design and implement a KV cache system, named `zExpander`, which dynamically partitions the cache into two sub-caches. One serves frequently accessed data for high performance, and the other compacts data and metadata for high memory efficiency to reduce misses. Experiments show that `zExpander` can increase `memcached`'s effective cache size by up to 2× and reduce miss ratio by up to 46%. When integrated with a cache of a higher performance, its advantages remain. For example, with 24 threads on a YCSB workload `zExpander` can achieve throughput of 32 million RPS with 36% of its cache misses removed.

1. Introduction

As an indispensable component of data center infrastructures, key-value cache, such as `memcached` [6], provides fast access to data that can be slow to re-fetch or re-compute in the back-end storage or database systems. Presented in the form of key-value pairs, the data is cached in the memory of a cluster of servers, and is accessed with a simple interface, such as `GET(key)` for reading data, `SET(key,value)` for writing data, and `DEL(key)` for removing data. With its rapidly increasing importance on entire storage and/or database systems' service quality, the cache's performance and its improvement have received extensive studies. These optimization efforts include reducing network stack cost, alleviating contention with concurrent accesses, reducing memory accesses, and optimizing memory management [22, 28, 30, 37, 40]. Some of the efforts leverage new hardware features, including direct cache access [25, 28] and RDMA [20, 27, 34, 38], or specialized hardware, such as GPU and FPGA [14, 23, 41], to speed up KV caches.

While a KV cache's performance, either in request latency or in throughput, is important, the significance of reducing its misses cannot be downplayed. In many usage scenarios, each miss represents a request to a back-end database system. In a busy KV cache system these misses, which may represent a small miss ratio, can impose significant workload on the database system. As reported in a study on Facebook's `memcached` workloads, eviction misses produced by one server can turn into over 120 million requests sent to the database per day, though they represent only a few percentage points of the server's miss ratio [40]. In other words, seemingly minor reduction of miss ratio can lead to significant workload reduction on the database system. As the miss penalty can be as high as a few milliseconds or even seconds [36], miss-ratio reduction is also important to reducing effective request response time. Significant efforts are demanded to reduce it.

Without adding DRAM to the server, there are two approaches to reduce misses, and unfortunately both approaches could potentially compromise the cache's performance. One approach is to apply advanced cache replacement algorithms. Over years a large number of replacement algorithms have been proposed, including LRU and advanced algorithms aiming to improve it, such as LIRS [26], ARC [33], and MQ [42]. However, in KV cache systems,

EuroSys '16 April 18–21, 2016, London, United Kingdom

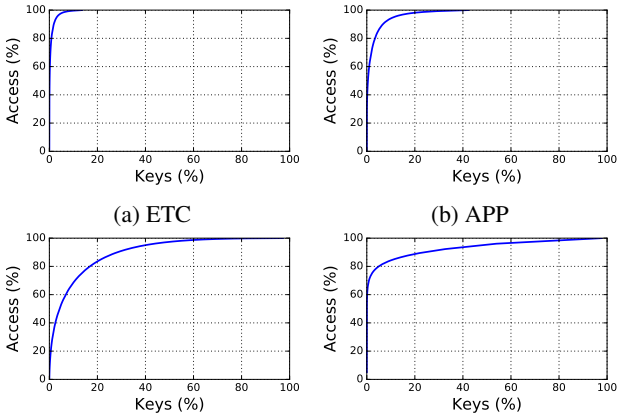
ACM 978-1-4503-4240-7/16/04…$15.00
DOI: http://dx.doi.org/10.1145/2901318.2901332

only LRU or its approximations are adopted for their low cost. As examples, `memcached` uses LRU, and MemC3 chooses a lower-cost LRU-approximation scheme (similar to CLOCK) for higher space efficiency and better concurrency support. The advanced algorithms expected to produce lower miss ratios are often (much) more expensive, requiring more space to track access history for data that have been evicted. As we will show in Section 2, their limited improvements on miss ratio are less interesting when the cost is considered.

Another approach is to use data compression technique to improve memory efficiency. It increases effective cache size and reduces miss ratio. However, adoption of this technique would require compression for `SET` requests and decompression for `GET` requests. For a KV cache system designed for high performance this additional cost seems to be unbearable. However, in the next section, we will show that KV cache's accesses are highly skewed and a majority of its `GET` requests are for a relatively small portion of the entire data set. This access pattern allows us to decouple the effort on providing high performance from that on reducing cache misses, and to achieve both goals (high performance and low miss ratio) simultaneously. The method is to partition the cache into two sub-caches, each managed by a different scheme and dedicated to achieving one of the goals. The first sub-cache is named *N-zone*, whose data are those frequently accessed and will not be compressed. Because of highly skewed access pattern, N-zone can have a relatively small space to serve a majority of requests with high performance. This sub-cache can be managed by a state-of-the-art KV cache scheme designed for high performance. The second sub-cache is named *Z-zone*, which is expected to hold a (much) larger portion of the cache space but serve a smaller portion of requests. Therefore, with small impact on the KV cache's performance Z-zone prioritizes its effort on memory efficiency to reduce misses.

In this paper we propose `zExpander`, a KV cache system applying this method in the management of a KV cache. In the following sections we will show that increasing cache size is an effective way to reduce misses. In the meantime, there are several challenges to address in the design of `zExpander`:

- First, in real-world KV workloads [9, 35, 40], most values in KV items are very small. Individually compressing them cannot produce substantial space saving.
- Second, metadata for indexing KV items can consume a substantial portion of memory when the items are small. A compact data organization is required to reduce the cost.
- Third, the space allocation between N-zone and Z-zone has to be dynamically adjusted to ensure the majority of accesses can be processed in N-zone to maintain performance and to ensure that Z-zone's space is effectively utilized to reduce misses.



(a) ETC (b) APP

(c) USR (d) YCSB

Figure 1: CDF curves for percentage of accesses associated with certain percentage of the most frequently accessed KV items.

2. Motivation of Increasing Effective Cache Size and Using Batched Data Compression

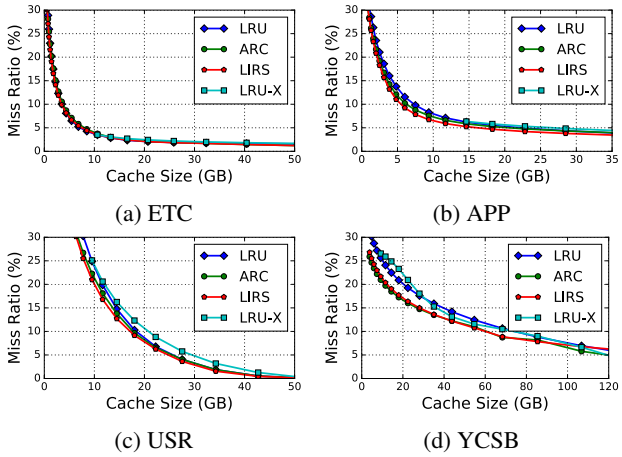
In this section we use workloads collected at Facebook's production `memcached` system and synthetic workload of representative access pattern to show that (1) accesses to KV cache are highly skewed, (2) increasing cache size is a necessary and effective means to reduce misses, and (3) batched data compression can achieve a much higher compression ratio than individually compressing KV items, facilitating the effort of increasing cache size.

2.1 Long-tail distribution and Impact of Larger Cache Size

To understand access pattern of representative KV cache workloads, we select three Facebook's `memcached` traces (USR, APP, and ETC) [13].[1] We also use Yahoo's YCSB benchmark suite [18] to generate a trace to access KV items. The trace covers a data set of about 128 GB and the popularity of keys follows a Zipfian distribution with a skewness parameter of 0.99, which is YCSB's default Zipfian parameter. The Zipfian distribution has been widely assumed on the KV cache's workloads in prior works on KV caches [20, 22, 28, 30, 31].

Figure 1 plots the access CDF (Cumulative Distribution Function) curves of the four workloads, or the percentage of accesses associated with most frequently accessed KV items. As shown, all workloads exhibit distributions with long tails, and a relatively small cache can cover a majority of accesses. For ETC, APP, USR, and YCSB workloads, the 3.6%, 6.9%, 17.0%, and 5.9% most frequently accessed items receive 80% of total accesses, respectively. This obser-

[1] In the paper for characterizing Facebook's memcached workload [13], there are five traces collected and analyzed. We choose three of them for this investigation. Trace VAR is not selected because it is write-dominated and reads only a few distinct keys. Trace SYS has a very small data set, and a cache of a few Gigabytes can produce almost a 100% hit ratio.



(a) ETC (b) APP

(c) USR (d) YCSB

Figure 2: Miss ratios of the KV cache workloads with different replacement algorithms. In the simulation, cache space used for metadata tracking access history, such as pointers in their linked lists, is not counted in the reported cache size.

Trace	Base Size	Algo.	Cache Size (×Base Size)				
			×1.0	×1.5	×2.0	×2.5	×3.0
ETC	9.5 (GB)	LRU-X	968M	-24.15%	-34.65%	-40.69%	-45.16%
		LRU	-0.00%	-30.99%	-43.42%	-49.49%	-52.76%
		LIRS	-2.55%	-30.68%	-43.53%	-50.35%	-54.19%
		ARC	-8.50%	-28.26%	-42.01%	-49.28%	-53.40%
APP	12.3 (GB)	LRU-X	2,273M	-16.18%	-25.22%	-31.95%	-37,32%
		LRU	-0.00%	-20.46%	-31.56%	-39.19%	-44.91%
		LIRS	-16.70%	-32.32%	-41.16%	-46.64%	-51.18%
		ARC	-7.27%	-23.96%	-33.28%	-39.74%	-44.09%
USR	9.2 (GB)	LRU-X	17,417M	-33.62%	-53.93%	-67.81%	-77.64%
		LRU	-0.00%	-38.60%	-61.61%	-75.58%	-84.39%
		LIRS	-15.83%	-47.47%	-65.74%	-77.42%	-85.97%
		ARC	-10.81%	-42.89%	-63.97%	-76.21%	-84.38%
YCSB	7.7 (GB)	LRU-X	4,905M	-3.78%	-9.09%	-15.92%	-23.53%
		LRU	-0.00%	-10.65%	-18.22%	-24.30%	-29.24%
		LIRS	-15.51%	-24.44%	-30.75%	-35.77%	-39.86%
		ARC	-18.29%	-26.77%	-32.69%	-37.37%	-41.13%

Table 1: Reference miss counts with base cache size and LRU-X replacement (such as 968 millions of misses for ETC at 9.5 GB (×1) and LRU-X), and percentages of the misses that are removed with use of larger cache (such as 1.5× or 2.0× of base cache size) and/or other replacement algorithms (LRU, LIRS, and ARC).

vation is echoed by miss ratio curves with different replacement algorithms, including LRU, LIRS, and ARC, shown in Figure 2. This suggests that a relatively small cache holding these hot items can achieve low miss ratios.[2] Let us define such a small cache that can accommodate the set of most frequently accessed KV items serving 80% of a workload's total accesses (see Figure 1) as the workload's base cache, and its size as the workload's base cache size. As long as the base cache can efficiently serve its requests, the cache's performance is mostly warranted. However, this cannot lead to the conclusion that a small cache is sufficient.

Though miss ratios with a cache larger than a workload's base cache are low, a busy KV cache can still produce a very large number of misses, or equivalently heavy load on the database system. A small improvement on miss ratio can make a substantial difference on the system's performance. It is necessary to make major efforts to keep removing the misses, including using optimized replacement algorithms and/or increasing cache size. To estimate the contribution of locality-aware replacement algorithms on miss ratio improvement, we apply a hypothetical replacement algorithm, named LRU-X, in which the base cache uses LRU, and data out of the base cache but still in the memory are managed by the random replacement policy.

To clearly observe impacts of replacement algorithm and cache size on miss ratio, we list miss count with base cache size and the LRU-X replacement, which is considered as a reference value, as well as percentage of the misses that are removed with use of larger cache and/or optimized replacement algorithms in Table 1. We have several interesting observations. First, with a cache larger, or even several times larger, than the base cache, increasing cache size can still substantially reduce misses. For example, for every 50% increase of the base cache size, the miss count is reduced by 8% to 31% for ETC, and by 30% to 38% for USR. Second, the locality-aware replacement algorithms moderately perform better than LRU-X, a scheme that simply selects random items in the long tail for replacement. This seems to suggest that for accesses in the long tail of very weak locality the room for further improvement by more carefully exploiting locality is limited. In addition, advanced algorithms, such as LIRS and ARC, need to spend substantial cache space to track accesses of KV items that have been evicted out of the cache, which essentially reduces effective cache size and offsets their advantages on miss ratio. Third, the benefit from increasing cache size is consistent across various replacement algorithms and workloads. In particular, this benefit does not disappear or even become smaller with advanced replacement algorithms producing lower miss ratio. While data compression can increase effective cache size, this observation suggests that it is a potentially effective technique for substantially reducing misses.

2.2 Batched Data Compression

A convenient approach to increasing KV-cache's effective size is to individually compress KV items in the cache. This approach can be effective for items with large values. However, small values are common in KV cache workloads. As reported in the study on Facebook's `memcached` workloads [13], except for one workload (ETC) 90% of cache space is occupied by values of under 500 B. In ETC, requests with values smaller than 16 bytes account for 40% of the total requests. Another workload USR virtually has only one value size, which is 2 bytes. In another example, a study on

[2] A `SET` request is always considered as a hit on the corresponding KV item in the calculation of the miss ratio.

Container Size:	Individual	256 B	512 B	1 KB	2 KB	4 KB
Tweets	0.99	1.10	1.21	1.30	1.34	1.41
Places	1.28	1.28	1.45	1.60	1.70	1.77

Table 2: Average compression ratios with compression applied on containers of different sizes. "Individual" is for compression of individual values of KV items. "Tweets" and "Places" represent the data sets for Twitter's tweets and location records, respectively. The ratio is between sizes of an uncompressed data object and its corresponding compressed object.

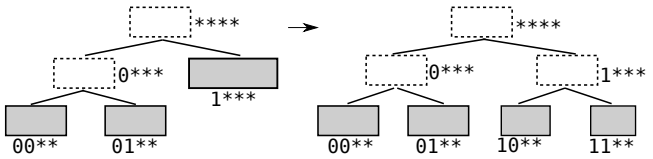


Figure 3: KV items in a Z-zone are compacted into blocks, which are organized into a binary trie. Only a leaf block (shaded) stores KV items (in the left), and grows two child blocks when it is overloaded (in the right).

Twitter's workloads finds that average value size of tweets is only 92 B [17].

To understand how compression ratio is affected by value size, we place values into containers of various sizes and apply `LZ4` [1], a high-speed compression algorithm, on the containers. We test two value sets, one is a collection of about 10 million tweets that have been collected in Twitter's service from September 2009 to January 2010 to study geolocation of twittering [17]. Average value size of the tweets is 92 B. Another value set contains records in a format named *Places* defined by Twitter to describe geographic locations with coordinates. We fill fields of the records with random data, and then encode the records using Google's Protocol Buffers, which is a widely used method for serializing structured data [7]. The average record size is 100.9 B.

Table 2 shows average compression ratios of the two data sets when each data items are compressed individually or collectively in a container of various sizes. It can be seen that compression with a larger set of data is more effective on reducing data size. The larger the container is, the better compression ratio is. For tweets, individual compression does not result in any reduction of their sizes. With such an observation, users do not have the incentive to compress their data beforehand. Therefore, we increase effective cache size by compacting small items into containers for batched compression.

3. Design of zExpander

`zExpander` is a KV cache designed for high performance and reduced misses, each objective is achieved by a dedicated cache partition (named as `N-zone` and `Z-zone`, respectively). While N-zone can be managed by any state-of-the-art high-performance KV cache scheme, we need to design a scheme for managing Z-zone for high memory- and access-efficiency, and a policy to dynamically adjust cache space allocation between the two zones to ensure that the KV-cache's performance is not (substantially) compromised with workloads of changing access patterns.

In `zExpander`, a new request is first processed at its N-zone. For a `GET` request, if it hits in the N-zone, the result can be immediately returned. Otherwise, the request is passed to the Z-zone. A `SET` request is always immediately admitted by the N-zone. Only when an item is evicted out of the N-zone, it is then admitted into the Z-zone. A `DELETE` is performed simultaneously at both zones. While an N-zone manager is almost a plug-in of any existing KV cache system, which is responsible for communicating with clients and serving frequent accesses, we focus on the design of Z-zone.

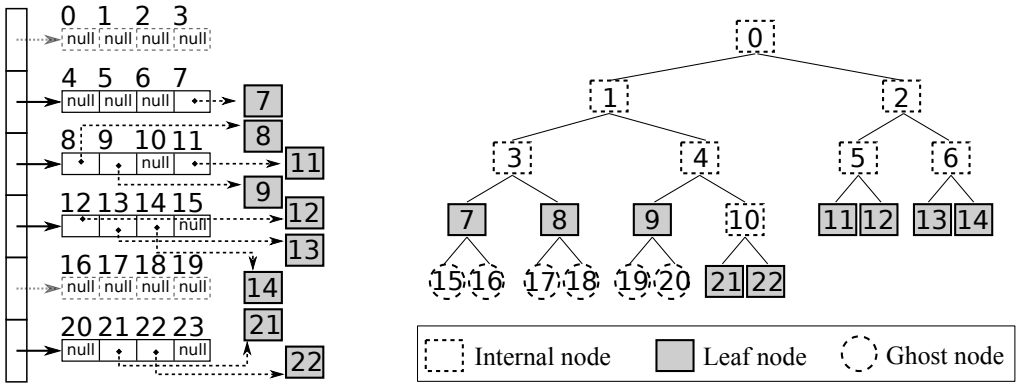
3.1 Data Organization of the Z-zone

As suggested in Section 2.2, KV items are placed into containers, which are named *blocks*, for effective compression[3]. Accordingly, indices are built on these blocks, rather than on individual KV items, to reduce use of pointers for space efficiency. In the design there are two objectives. One is that a block should be always well loaded for space efficiency. The other is that time to reach a block for a requested item should be minimized for CPU efficiency.

To this end, we organize the blocks into a binary trie, or a binary prefix tree. As shown in Figure 3, each tree node is labeled with a binary prefix. Each KV item's key is also a binary string. An item is stored in a leaf node whose label matches its key's prefix. In other words, internal nodes do not contain data. When a block is full and a new item needs to be inserted into the block, it grows two child nodes, and each of its items is moved to one of the new nodes according to next bit of its key's prefix. Accordingly the corresponding node becomes an internal tree node, the space held by the block is de-allocated. Without a physical presence to consume memory, the node is associated with a NULL pointer to facilitate key searching in the tree, which will be explained soon.

We choose binary tree, rather than regular tree allowing for more children from a node, so that new nodes (blocks) are more likely to be well loaded (at least half full). However, to this end we also need to make sure storage load on these two nodes is well balanced. In addition, the entire binary tree has to be well balanced to avoid excessively long paths leading to some leaf nodes, or to avoid long access time for walking from the root to a leaf node on the trie. To this end, we apply a hash function, such as MurmurHash [12], on keys, and then use the hashed keys to locate the corresponding KV

[3] For KV items larger than half of a block's capacity, each of the items is compressed and stored individually, and a pointer recording its address is stored in the block where it is supposed to stay. As zExpander is designed for workloads of small items, we expect such large items are rare in the cache.



(a) Two-level pointer array

(b) Complete binary tree storing KV items

Figure 4: A trie, where KV items are stored at its leaf nodes, is enhanced with ghost leaf nodes to form a complete binary tree (b). Nodes on the tree are accessed through two-level pointer arrays (a). Note that spaces for internal and ghost nodes in (b) and pointer segments, such as [0,1,2,3] and [16,17,18,19] in (a), are virtual, or not physically allocated, to save memory. Furthermore, the links between parent and children nodes in (b) do not have physical presence.

items in the trie. In this way, every block has an equal probability to receive KV items, and the chance of unbalanced data storage in the tree structure is minimized.

A conventional approach to locate an item in a tree is to chase pointers starting from the root node along a deterministic path leading to a leaf node according to the item's key. However, on a binary tree, the path can be substantially long. Due to processor cache misses during the pointer chasing, the access time can be unnecessarily high. For higher access efficiency, we add minimal number of *ghost* leaf nodes to the tree to make it a complete binary tree. In a complete binary tree, "every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible." [5]. Each node is pointed by a pointer. A ghost node does not have any physical presence, and is pointed by (or associated with) a NULL pointer. We linearize these pointers into an array ordered from the top level to the bottom level, and in each level from the left to the right. Now a complete binary tree is formed. From the last pointer's position in the array, we can directly compute the index of the last level (leaf or ghost) node corresponding to a given prefix of a hashed key. From this last level node, we can trace up the tree to identify the leaf node (with a non-NULL pointer) that possibly stores the item. This approach avoids traversing a (long) list of pointers from the root. In practice, to identify the non-NULL pointer, we only need to inspect a few (usually fewer than three) consecutive pointers up along the path starting from the last level node.

As shown in Figure 4, to avoid recording many pointers to the ghost nodes, or NULL pointers, we evenly partition the array into segments of fixed size, each with 128 4-byte-pointers. This array is then considered as the second level of pointers, and we maintain a so-called first-level array of pointers, each pointing to a segment in the second level. If all pointers in a segment are NULLs, the segment is not allocated to save memory space, and a NULL pointer is set in the corresponding slot of the first-level array.

3.2 Data Access and Replacement in the Z-zone

As KV items are compacted and compressed as a whole in a block with a default capacity of 2 KB, accessing the KV items involves decompression and/or compression of the block. Writing a new item into a block always leads to its reconstruction, including decompressing items in the block, re-compressing the existing items and the new item, allocating new memory space(s) to store the new block(s), and freeing the space held by the original block. Reading an item from a block requires a decompression of the block and searching the decompressed data for the item. For a quick search in the block, items are arranged in a block according to a hashed key order [29], and a small index consisting of offsets of up to eight items evenly spaced in the block is recorded so that only a few KV items have to be checked for a look-up in the block. In addition, to avoid unnecessary decompression operations on a block when looking for non-existing items, each block is associated with a Bloom filter, named as *Content Filter*, to remember items in the block. Expecting a block contains roughly 20 (for block size of 2 KB and small item size of 100 B) or fewer KV items, the Bloom filter is of 16-byte long. For each `GET` or `DELETE` request the *Content Filter* must be first checked to determine if the block needs to be accessed.

Unlike `memcached` who uses slab class to manage its own space (de)allocation for KV items, `zExpander` relies on the general-purpose memory allocator (`malloc`), usually provided by `Glibc`, for the allocation/de-allocation in the Z-zone. In this way, there is no internal fragmentation in the zone. Meanwhile, because the allocation size (a block) is large, space efficiency is less of a concern [37].

When the Z-zone has reached its allocated size, we need to determine an item for replacement. For this purpose, `memcached` organizes KV items in linked lists to find the least-recently-used (LRU) item. However, for a cache storing small items, the space overhead for so many pointers can be too high. In addition, `zExpander` cannot track access history of the blocks to make its replacement decision. As the block where a KV item is placed is determined by the hashed key, each key has an equal probability to be inserted into any block, and each block can usually contain a mix of popular and unpopular items. To address the issue we need to exploit access locality of items within individual blocks. This is much like what the replacement policy for a set-associative processor cache does, except that `zExpander` needs to determine an item in a block, rather than a cache line in a cache set, for replacement. To efficiently track access history of items in a block, we associate a Bloom filter, named *Access Filter*, to each block to record recently accessed items. Like a *Content Filter*, an *Access Filter* is also of 16-byte long. Whenever an item is accessed for `GET`, its key is recorded in the corresponding *Access Filter*.

In the replacement policy, all blocks, or the trie's leaf nodes containing KV items, are linked into a circular list. `zExpander` sweeps around the list. At each block it stops at, it tries to select victim items for replacement before moving to the next block. The victim items are selected by randomly choosing half of the items that are not recorded in the *Access Filter*. If all items are recorded in the filter, it skips the block. The *Access Filter* is cleared before `zExpander` moves to the next block, so that recent accesses can be recorded in the filter before this block is examined again.

3.3 Limiting Accesses in the Z-zone

While accessing KV items in the Z-zone is more expensive than that in the N-zone, especially for serving write requests, `zExpander` needs to control (relative) number of accesses on the zone. There are two potential issues that may cause a Z-zone to receive too many accesses. One issue is that the corresponding N-zone is not sufficiently large and causes actively accessed KV items to be spilled into the Z-zone. The other issue is frequent movements of items into and out of the Z-zone due to use of two-zone caching in `zExpander`.

3.3.1 Adaptive Cache Space Allocation

To address the first issue, we adaptively adjust cache space allocation between the two zones so that most requests can be processed at the N-zone and, if possible, the Z-zone has a substantially large size to help with the system's memory efficiency. An N-zone has its target size, and the gap between its actual size and the target size suggests an action to expand or shrink the N-zone. Accordingly, a Z-zone has an action status, which can be *expand*, *shrink*, or *stay*, which indicates size change is not necessary. `zExpander` periodically checks the fraction of requests serviced at the N-zone in the current time window (one minute by default). If it is non-trivially smaller than a target threshold (90% by default) and the current action status is not *expand*, `zExpander` increases the zone's target size by 3% of the cache space. Otherwise, if it is non-trivially larger than the target threshold and the current action is not *shrink*, the zone's target size is reduced by 3% of the cache space. We assume that the N-zone is managed by a KV cache system that supports resizing memory to a given size. Details on its implementation in `zExpander`'s prototype is described in Section 4.

When new items are added into the N-zone, `zExpander` will usually evict not-actively-accessed items into the Z-zone. Accordingly, to expand N-zone to its target size, `zExpander` simply keeps these items from being evicted. To shrink the N-zone, we leave a thread in the background and activate it for moving not-actively-accessed items into the Z-zone when the processors are not fully utilized.

In the calculation of requests serviced at a zone, we do not consider requests that do not require block (de)compression. These include missed `GET` requests and `DELETE` requests on non-existing items. Both types of the requests can be identified by Content Filters and the requests can be efficiently serviced. On the other hand, we consider items that are evicted from the N-zone into the Z-zone as requests serviced at the Z-zone. In this way, only expensive operations are counted and their impact on the system's performance can be effectively capped.

3.3.2 Minimizing Write Operations at Z-zone

To address the second issue, we need to identify *unnecessary* item movements and remove them. Due to existence of access locality, a KV item can be relatively either an actively accessed (`hot`) one or an inactively accessed (`cold`) one in a certain time period. A hot item's home zone is N-zone and a cold one's home zone is Z-zone. When access pattern changes and a cold (hot) item turns hot (cold), moving the item from Z-zone (N-zone) to N-zone (Z-zone) is necessary. In the meantime, there are two scenarios where the movements are not necessary.

One scenario is that when an item is read from the Z-zone when a `GET` request is serviced, we need to know if it should be removed from the zone and inserted into the N-zone. To do this effectively, we need to make sure the item has turned from cold to hot. Otherwise, it would quickly return from the N-zone and back to the Z-zone. An item's status, hot or cold, depends on its relative locality strength compared to that of items in the other zone. Therefore, the key to the answer on whether the item currently in the Z-zone has turned hot when it is read is to quantitatively compare locality strength of this item to that of any item currently in the N-zone. To this end, we need to measure an N-zone's locality strength, which is defined as the weakest strength of all of its cached items. It is desired to have an efficient approach that does not require modification of KV cache code managing the N-zone. To this end, we treat the zone as a black box and periodically issue a special `SET` request, named as *Marker* request, into

the N-zone. Each *Marker* request has a unique key[4] and will never be re-accessed. We then observe how long it will take for the item written by the *Marker* request to be evicted out of the zone. This duration represents the Z-zone's locality strength, and is considered as its *locality benchmark*. The shorter the benchmark, the stronger the Z-zone's locality strength. To be more effective, the benchmark we adopt is a weighted average of three most recent benchmarks. When an item in the Z-zone is accessed for the first time, `zExpander` records its access time without moving it into the N-zone. When it is re-accessed, the time gap, or re-use time, from its last access is calculated. If the re-use time is smaller than the N-zone's locality benchmark, the item is moved to the N-zone. Otherwise, it remains in the Z-zone. In this way, the item being moved into the N-zone is likely to be actively re-accessed and stay there, and the item remaining in the Z-zone is less likely to be re-accessed soon. As we only need to identify the items that are best qualified to be moved from Z-zone into N-zone, for each block we only maintain two records for its recent accesses, each containing a hashed key (4 bytes) and a access time (4 bytes). They are of only 16 bytes, less than 1% of each block's size.

The other scenario is that a `SET` request is received and a new KV item is first written into the N-zone. If at this time the old version of the item (of the same key) is in the Z-zone (by checking the *Content Filter* of the corresponding block), `zExpander` needs to decide if the version should immediately be removed from the Z-zone. For memory efficiency, we should keep an item from being doubly cached. However, an immediate removal may be followed soon with an eviction of the item's new version from the N-zone and insertion into the Z-zone if the item is a cold one. To avoid the unnecessary early removal, we postpone it for a time period at least equal to the N-zone's benchmark. If the item is evicted before or around when the time period expires, removal and write operations are merged into one at the Z-zone. Otherwise, the removal will be combined with the space reclamation in the Z-zone. When Z-zone's replacement algorithm runs, it will first execute the pending removal operations, if any, before looking for LRU items in the blocks for replacement.

4. Evaluation

`zExpander` has been implemented and extensively evaluated with different workloads and system configurations. In the evaluation, we will answer three questions. First, can `zExpander` substantially reduce misses with little or limited loss of performance? Second, if `zExpander`'s performance loss is minimal with using `memcached` to manage its N-zone, is this still true with a high-performance KV cache of very-low-cost networking, such as RDMA, and increasingly large number of threads? Third, can `zExpander` effectively respond to change of access pattern and opportunistically retain its advantage on miss reduction with its adaptive space allocation? In addition, we will also demonstrate why individually compressing KV items in off-the-shelf KV caches is not sufficient to achieve desirable miss reduction.

[4] It contains special characters so it cannot appear in real workloads.

4.1 Implementation of Two zExpander Prototypes

We have built two prototypes, a `memcached`-based `zExpander` and a high-performance-KV-cache-based `zExpander`.

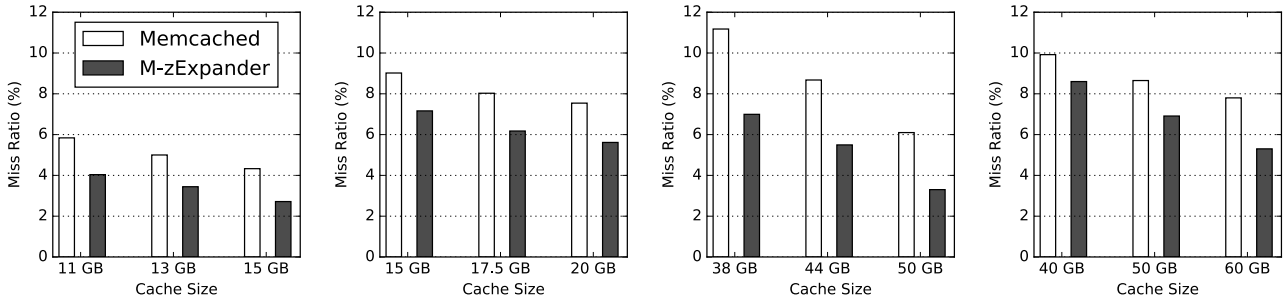
In the first one, the N-zone is managed by `memcached` (Ver. 1.4.24), which is also responsible for communication with clients. However, `memcached` does not support on-line change of cache size, a capability required by `zExpander` and we attempted to add into `memcached`. The challenge is that `memcached` maintains multiple slab classes, each for storing KV items at a certain size range. To add or remove cache space, we would have to decide which classes' allocations need to be increased or decreased and by how much. A decision on this has implication on what KV items are cached or replaced as well as on the cache's miss ratio [24]. To keep authenticity of `memcached`'s behaviors and performance in the evaluation, we choose not to include the mechanism for adaptively adjusting cache space allocation in this prototype. Instead, we manually determine the target sizes for N-zone and Z-zone and statically configure them. In this prototype, we add about 85 lines of code to integrate `memcached` into `zExpander`, mainly for catching events, such as item evictions, `GET` misses, and item writes, so that corresponding operations at the Z-zone can be triggered.

Because items in different classes are managed in different LRU queues in `memcached` according to their sizes, we maintain a locality strength benchmark for each class, and use them to decide item movements depending on item size.

It is known that `memcached`'s performance is seriously constrained by its networking cost [30, 35, 40]. Its use of linked list in the hash table and use of the LRU list can lead to substantial processor cache misses, and limits the KV-cache's performance. While these overheads can potentially overshadow the cost of Z-zone operations, we build a high-performance `zExpander` prototype to fully expose the Z-zone operation cost. In this prototype, network processing is removed by issuing requests at the user level of the server where the prototype KV cache runs. It also adopts optimistic Cuckoo hashing and CLOCK-based replacement suggested in MemC3 [22] to further improve the efficiency of N-zone operations.

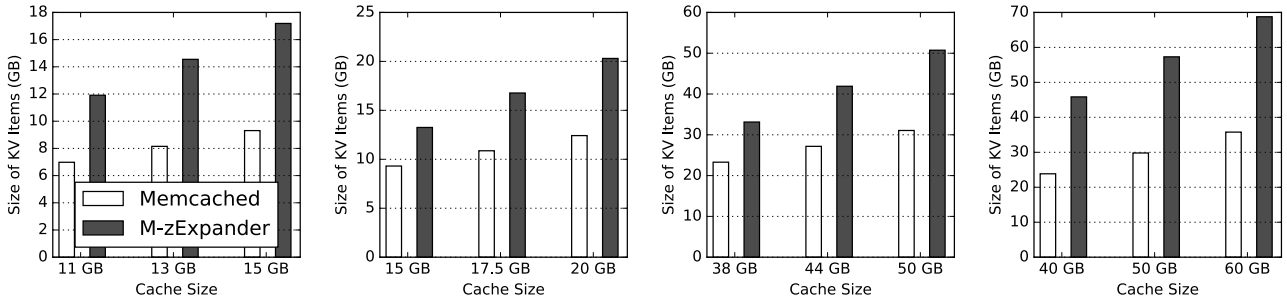
4.2 Experiment Setup

In this evaluation, we replay the four traces (three Facebook's `memcached` traces and one Zipfian trace generated by Yahoo's YCSB) used in Section 2 as the prototyped system's workloads. While the traces do not contain actual values, we use the data sets about Tweeter's location records to emulate the values (see Section 2.2). The value size is distributed in the range from 2 B to 327 B with an average of 100.9 B. We use LZ4 compression algorithm [1]. In the YCSB trace the



(a) ETC (b) APP (c) USR (d) YCSB

Figure 5: Miss ratios of systems using `memcached` with different workloads. One thread is employed for serving requests.



(a) ETC (b) APP (c) USR (d) YCSB

Figure 6: Size of (uncompressed) KV items cached in the systems using `memcached` with different workloads at the time when a trace has been replayed.

ratio of `GET` and `SET` requests are 95% vs. 5% if not otherwise specified. In the trace replaying, we use first 1/5 of a trace to warm up the cache before collecting measurements.

The server running the prototypes has two Xeon E5-2680 v3 CPU, each having twelve cores with hyper-threading disabled and 30 MB last-level cache. The server is equipped with 128 GB (8×16 GB) DDR4 memory. For the `memcached`-based `zExpander`, we use another server of the same configuration to generate and send requests over 10Gb Ethernet.

Each of the Facebook traces is a sequence of serialized requests. Concurrency relationship among requests is lost in the trace collection. To keep authenticity of the trace we do not artificially break it into multiple concurrent segments for replaying. Accordingly for these workloads, only one thread is employed to serve requests at the server.

In the below for brevity we name `memcached`-based `zExpander` as **M-zExpander**, the high-performance-KV-cache-based `zExpander` as **H-zExpander**. We also have **H-Cache** by removing the Z-zone from `H-zExpander` and running the high-performance KV-cache exclusively.

4.3 Results for `memcached`-based KV Caches

Figure 5 shows miss ratios for the four workloads (ETC, APP, USR, and YCSB) running on `memcached` and `M-zExpander` with various cache sizes. Cache sizes for different workloads are chosen according to their respective data set sizes. As shown, `M-zExpander` can substantially reduce miss ratio, by up to 46%. The actual reduction depends not only on the increase of effective size but also on the workload's demand on cache space in the increased cache size range. Figure 6 shows the size of KV items cached in the systems (in their uncompressed form) corresponding to each of the experiments in Figure 5. As an example, Figure 5 shows that USR achieves the largest miss ratio reduction (from 37% to 46%) with `zExpander`. However, its increase of amount of cached KV items with `zExpander` is moderate (from 42% to 63%) compared to the increases with other workloads. As suggested in Figure 2 and in Table 1, USR's miss ratio has the largest reduction in the range of cache size from 20 GB to 50 GB.[5] Another observation in Figure 5 is that the miss ratio reduction is pretty consistent across the selected cache sizes, suggesting that `zExpander` can be effective in a considerably large range of cache capacity. With `zExpander`, cache can be sized economically without compromising miss ratio. For example, for APP, a 15-GB cache with `zExpander` can have a miss ratio lower that a 20-GB cache with `memcached`.

[5] Note that cache sizes indicated in Figure 2 and in Table 1 include only (uncompressed) KV items, or those shown in Figure 6 as "Size of KV Items".

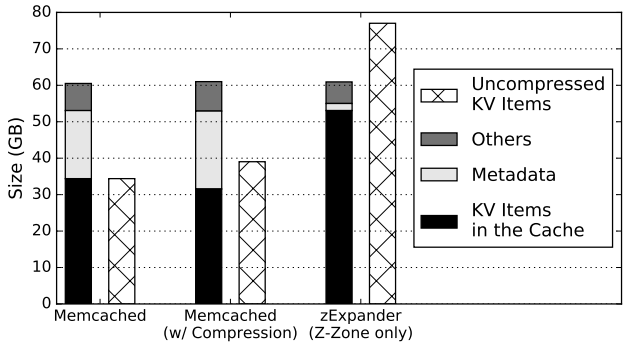


Figure 7: Comparison of memory usages of a 60-GB KV cache managed by `memcached`, `memcached` whose KV items are individually compressed, and `zExpander` that has only Z-zone. The usage includes spaces used for KV items, metadata, and others (e.g., allocation fragmentations). It also shows amount of KV items in a cache should they be not compressed.

To obtain insights on how `zExpander` allows substantially more KV items to be cached without adding DRAM, we first analyze the memory usage of `memcached`. The left two bars of Figure 7 show the usage for the YCSB workload at a 60-GB cache (corresponding to the right two bars of Figure 6d). As shown, in a `memcached` cache of 60 GB, only 56% of it (34 GB) is actually used to store KV items, and about 32% is for metadata, including three pointers to each item for hash table and its LRU replacement policy, and other cache management metadata specific to each individual item. The remaining cache space is mainly consumed by internal fragmentation in its slab allocation. `zExpander` relies on its Z-zone to increase effective cache size, or store more KV items. So we assume a `zExpander` with only Z-zone to cache the YCSB KV items. As shown in the right two bars in Figure 7, in the zExpander-managed cache, 88% of its space is used to store (compressed) KV items. Compared to its uncompressed form, the size of cached KV items is increased by 126% (34 GB vs. 77 GB). Because of its use of compact data structure (organizing blocks in the binary trie), the metadata holds only 3.3% of the cache space. Even the allocation fragmentation is reduced with the use of `Glibc`'s memory allocator. This is made possible by (de)allocating larger blocks, rather than individual KV items.

To understand how a `memcached` that simply compresses items individually would help with memory efficiency, we write compressed KV items into the cache. The result is shown in the middle two bars of Figure 7. Unfortunately, with this compression only 13.5% more KV items are cached, and metadata cannot be reduced at all without use of batched compression and compact data structure.

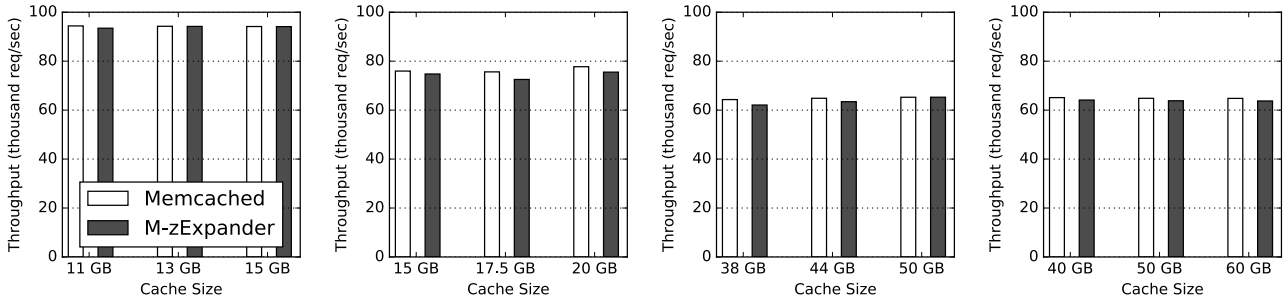
Figure 8 shows throughput of the KV caches with various workloads and cache sizes corresponding to each of the experiments in Figure 5. As we explained, only one thread is employed to obtain the results. For the YCSB workload, we increase the number of threads (up to 24), each exclusively on one core, and show the throughput with different cache sizes in Figure 9. As seen, in the experiments `M-zExpander`'s throughput is within 4% of that of `memcached`, though it serves about 10% of requests at its Z-zone. A major reason is that `memcached` has a serious bottleneck on its networking processing [30, 35, 40]. Its throughput is less than 100 K RPS (requests per second) with one thread, and less than 700 K RPS with 24 threads. The throughput is even much lower than that of `zExpander` that serves all requests at its Z-zone, which is around 1.3 M RPS with one thread and around 18.1 M RPS with 24 threads, if networking is excluded. To fully expose the potential performance impact we compare the high-performance cache without networking involved (`H-Cache`), and `zExpander` with `H-Cache` to manage its N-zone (`H-zExpander`).

4.4 Results for High-Performance KV Caches

Figure 10 shows throughput of `H-Cache` and `H-zExpander` using YCSB workload with different mixes of `GET` and `SET` requests. As seen, the peak throughput can reach as high as 33 M RPS with the all-`GET` workload. For each workload, `H-Cache`'s throughput keeps increasing with number of threads until the number arrives at about 15. Beyond this, it reaches a plateau and even slightly falls because lock contention intensifies. Before the number of threads becomes high (around 20), `H-zExpander`'s throughput is about 10%–15% lower than that of `H-Cache`. This is understandable as around 10% of requests are served at the Z-zone with a higher cost. However, when more threads are added, `H-Cache`'s throughput increase becomes slower or even stops. `H-zExpander`'s throughput keeps its increase longer and eventually (almost) catches up with `H-Cache`'s throughput. With the same number of threads, the lock contention is less severe in `H-zExpander` than that in `H-Cache` as some of the threads are diverted to perform more expensive operations at Z-zone. In other words, `H-zExpander` leverages some of the CPU cycles waiting for locks to meet the demand from Z-zone operations. In this way, for `H-zExpander` operations at its Z-zone does not compromise its N-zone's performance. Another scenario where `H-zExpander` almost does not suffer any performance loss is when the system does not run at its peak capacity and has spare CPU cycles. Because most KV cache systems are over provisioned, this scenario is common.

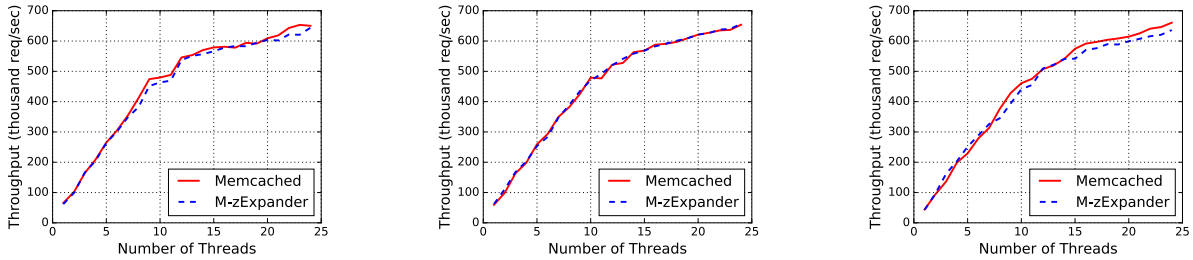
With more `SET`s, both systems' throughput reduces, as `SET`s intensifies `H-Cache`'s lock contention and they are (much) more expensive in `H-zExpander`'s Z-zone by involving compression. However, the relative throughput trend between the two systems stays.

Figure 11 compares the systems' performance for the YCSB workload with 24 threads using a different metric–request processing time, which is the turnaround time of a request in the system and will be part of the latency observed by clients in a networked setting. With a smaller percentile, `H-Cache` has smaller processing time. However, at high per-



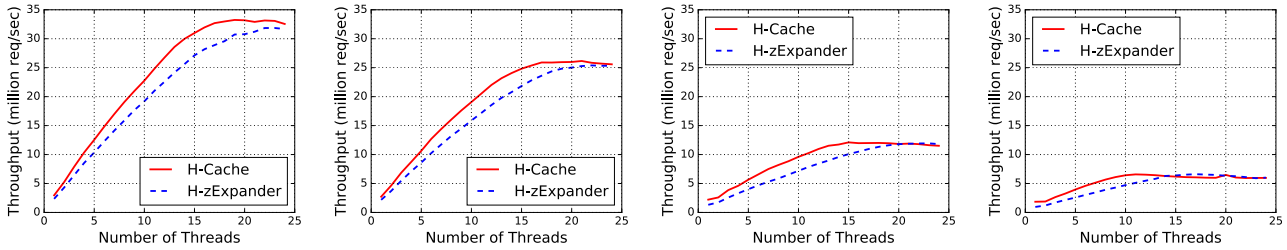
(a) ETC (b) APP (c) USR (d) YCSB

Figure 8: Throughput of systems using `memcached` with different workloads. One thread is employed for serving requests.



(a) Cache Size 40 GB (b) Cache Size 50 GB (c) Cache Size 60 GB

Figure 9: Throughput of systems using `memcached` with the YCSB workload and different number of threads. Caches of three different sizes are tested.



(a) 100% `GET`, 0% `SET` (b) 95% `GET`, 5% `SET` (c) 50% `GET`, 50% `SET` (d) 5% `GET`, 95% `SET`

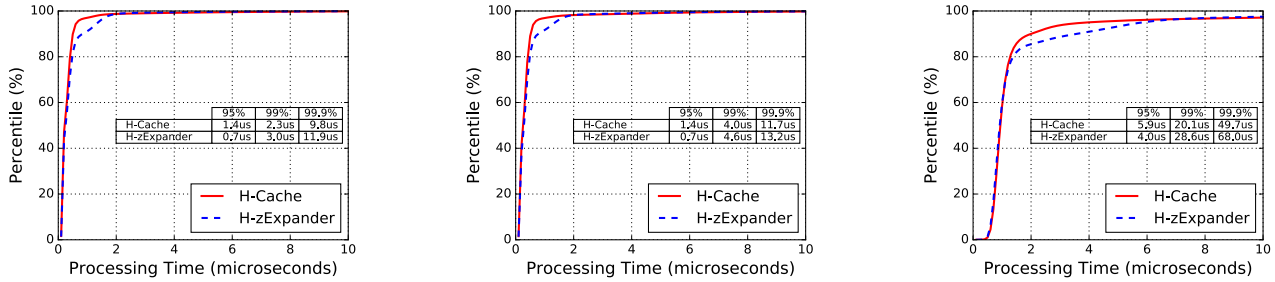
Figure 10: Throughput of systems using the high-performance cache with the YCSB workload and different number of threads. The cache size is 60 GB. Different mixes of `GET` and `SET` requests are tested.

centiles, the `H-zExpander`'s latency becomes smaller. For example, at the 99% percentile, the times for `H-zExpander` and `H-Cache` are 4.0μs and 4.6μs, respectively (see Figure 11b). Intensive lock contention has been notoriously known to cause long execution delay [15, 16]. `H-zExpander` accidentally ameliorates the contention at the N-zone by redirecting some requests to the Z-zone.

Figure 12 shows the miss rate, or number of misses per second in each of the experiments shown in Figure 10. The reductions of misses are significant, often by 30% to 40% and by up to 1.48 million requests per second. Though `H-zExpander` has a lower throughput (by 10% to 15%), its reduction on throughput does not overshadow its improvement on miss rate. Altogether H-zExpander's improvement on miss reduction is still impressive.

4.5 Effectiveness of Using Bloom Filter to Reduce Decompressions

To access a key in `zExpander` the binary trie for the Z-zone can only locate a block that possibly contains the key. To know if the key actually appears in the block, one has to perform expensive block decompression operation before the key can be searched. For reading non-existing keys, or `GET` misses, `zExpander` employs a Bloom filter (*Content Filter*) for each block to avoid unnecessary decompression operations. To quantitatively assess efficacy of the filter, we run a `GET`-only workload with different percentages of non-

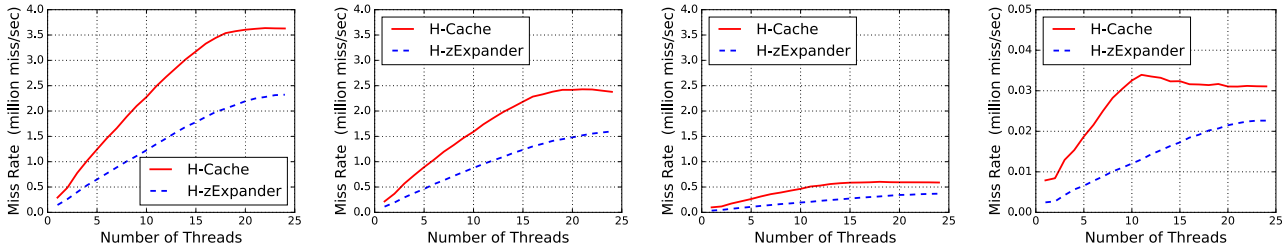


(a) Latency of `GET` (100% `GET`, 0% `SET`)

(b) Latency of `GET` (95% `GET`, 5% `SET`)

(c) Latency of `SET` (95% `GET`, 5% `SET`)

Figure 11: CDF curves of request processing time in the systems using the high-performance cache with the YCSB workload and different number of threads. The cache size is 60 GB. Different mixes of `GET` and `SET` requests are tested.



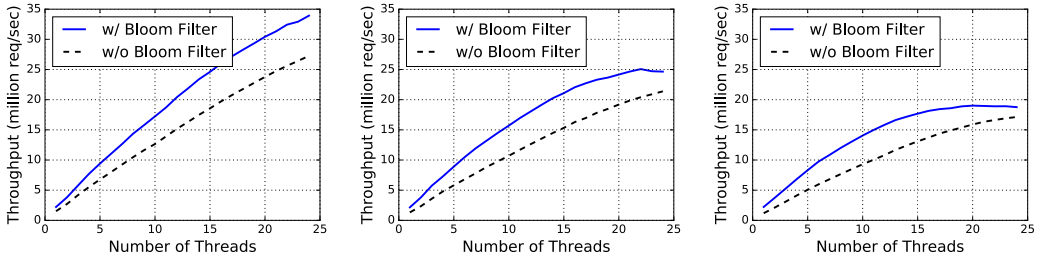
(a) 100% `GET`, 0% `SET`

(b) 95% `GET`, 5% `SET`

(c) 50% `GET`, 50% `SET`

(d) 5% `GET`, 95% `SET`

Figure 12: Miss rate, or misses produced per second by the systems using the high-performance cache with the YCSB workload and different number of threads. The cache size is 60 GB. Different mixes of `GET` and `SET` requests are tested.



(a) 50% Miss

(b) 75% Miss

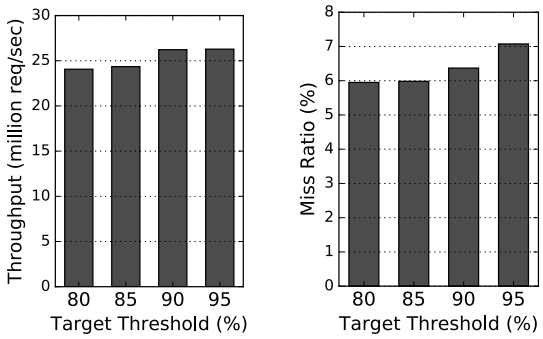
(c) 100% Miss

Figure 13: Throughput of `H-zExpander` that uses or does not use Bloom filters

existing keys, or different miss ratios. Figure 13 shows the throughput with the YCSB workloads at different thread count when the filters are used or not.

As shown in the figure, using the filters can substantially increase the cache's throughput. Our measurements show that the filters' false positive ratio remains at around 5%, or around 95% of misses do not come with block depressions. The throughput increase correlates with the miss ratio when the thread count is small. For example, with 50%, 75%, and 100% miss ratios, the increases are 39%, 53%, and 64% with five threads, respectively. However, when more threads are used, a higher miss ratio does not lead to a higher throughput increase. For example, with 20 threads the increases are 27%, 26%, and 20% at 50%, 75%, and 100% miss ratios, respectively. With a large number of threads and correspondingly high throughput, the impact of decompression on performance becomes less significant and other costs, such as lock contention, take a higher weight. In this case the benefit of using the filters does not increase with miss ratio.

Another observation on Figure 13 is that higher miss ratio leads to lower throughput, even when the Bloom filters are used. With a highly skewed access pattern, most request hits are served at the N-zone, thus are much more efficient than misses, which are always served at the Z-zone. Even though most decompressions can be avoided for misses, a higher miss ratio still degrades the throughput.



(a) Performance (b) Miss Ratio

Figure 14: Throughput and miss ratio of `zExpander` using the high-performance cache with the YCSB workload, 60 GB cache, and 24 threads. Different target percentage thresholds for accesses at N-zone are tested.

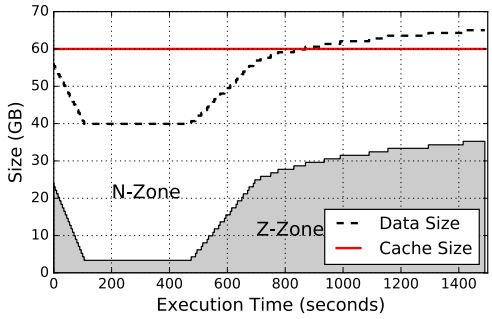


Figure 15: Allocation changes (from uniform to Zipfian patterns) between N-zone and Z-zone in response to change of access pattern. `H-zExpander` with 24 threads and a mix of 95%-`GET`/5%-`SET` is used. The cache size is 60 GB. The space allocation to the Z-zone is shown as the shaded area.

4.6 Impact of Space Allocation between N-zone and Z-zone

How to allocate cache space between N-zone and Z-zone is an important issue. With a too-large allocation to the N-zone, effort on miss reduction would be compromised. With a too-large Z-zone allocation, the cache's performance could be unduly affected. While the allocation has to change in respond to changes of access pattern, a parameter of `zExpander` about this is target percentage threshold, or the percentage of requests that should be processed at the N-zone. Figure 14 shows throughput and miss ratio of a 60 GB cache with the YCSB workload. As expected, the larger the threshold, the higher the throughput and the higher the miss ratio. As long as this threshold is sufficiently large (but not too close to 100%), its impact on throughput and miss rate is moderate. In the `H-zExpander` prototype, we choose 90% as the threshold, which provides high throughput and decent miss ratio reduction.

With a selected target threshold, `zExpander` automatically adapts to its space allocation to access pattern change.

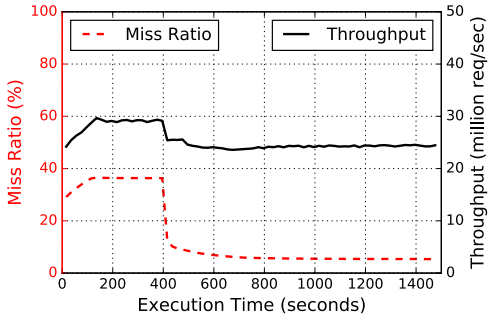


Figure 16: Miss ratio and throughput corresponding to the experiment results shown in Figure 15

To observe the adaptation and its performance implication, we write about 24 GB KV items to the N-zone and the rest to fill the Z-zone of a 60 GB cache. We then send requests with a mix of 95% `GET` and 5% `SET` to the cache, initially with a uniform access key distribution, then (at around 480th second) change access pattern to Zipfian, or the access distribution assumed in our YCSB workload. Figure 15 shows the amount of data cached in the N-zone and that in the Z-zone. Figure 16 shows the miss ratio and throughput corresponding to allocations in Figure 15. With the uniform access pattern, N-zone has its maximum allocation, and almost all items are uncompressed. The cache stores only about 40 GB items. In the meantime, it has high throughput (29 M RPS) and high miss ratio (about 37%). After changing to the Zipfian access pattern, it takes about 900 seconds for the space re-allocation to be completed. After the re-allocation, N-zone has about 25 GB and most of the space goes to the Z-zone. With compression at Z-zone, the size of cached items increases to about 65 GB. The throughput is only moderately reduced (from 29 M RPS to 24 M RPS), and miss ratio is reduced to only 5.2%.

5. Related Work

`zExpander` is a KV cache system that leverages highly-screwed access pattern to simultaneously achieve high performance and low miss ratio. There have been many efforts reported in the literature related to various components of this work, including those on memory compression, memory allocation, effective indexing of KV items, and high performance KV caches.

Compression in main memory. As memory is fast but expensive, Compression Cache [19], `ZSwap` [11] and `zram` [10] use some memory space as swapping area, in addition to that on the slow disk, and compress data in the area. Because memory pages in the space are compressed, the relatively small in-memory virtual area can be presented as a (much) larger swap space. With in-memory data compression, `zExpander` follows the same idea. However, it has to address a difficult challenge. As data for compression on swap area is in the page unit, which is usually 4 KB and

large enough to support an effective compression, KV items are of sizes distributed in a large range and often small (as small as a few bytes). Even though some KV in-memory stores, such as Redis [8], recommend users to individually compress KV items, the benefit is highly dependent on item sizes. `zExpander` is immune to this constraint by aggregating items into larger blocks before applying compression. Another issue with individual compression is that metadata cannot be 'compressed'. With a large number of small items in a cache, the metadata can be significant. By aggregating items, `zExpander` can also substantially reduce metadata. While small KV items are common in KV-cache workloads, `zExpander`'s contribution is substantial.

Indexing data structure. When metadata, or data for indexing KV items for their locations, can be a significant space overhead with small items, KV stores usually use compact data structure with sparse indices, such as LSM-tree [2, 29, 39]. As KV items of a KV store are mostly on the disks, the design goal is to minimize I/O operations, instead of processor cache misses, in searching for an item. In contrast, `zExpander` takes effort to reduce the metadata's memory size and the cache misses by using batched item storage, balanced binary trie, and address calculation.

Fragmentation in memory allocation. Another source of memory overhead is memory allocation. Dynamic memory allocators, such as `malloc/free` in `Glibc` and its alternatives, are convenient choices and widely used [3, 4, 21]. However, for frequent allocation and deallocation of a large number of small items, space overhead due to fragmentation can be very high. While small items are popular in KV caches, this issue has to be addressed. There are two possible approaches. One is that adopted in `memcached`, which obtains large fixed size memory chunks (2-MB slabs) from the system and performs memory allocation by itself. Slab allocation has been a very successful strategy for managing data items of fixed sizes, such as `inode` and `dentry` of file systems. However, KV items are of all different sizes, which makes substantial internal fragmentation in the slabs, as revealed in our experiments. Another approach is to use recently proposed log-structured memory allocation [37]. However, it requires constantly moving data objects, imposing high CPU overhead, especially when the memory space is fully occupied, which is almost always the case with a KV cache. By aggregating KV items into blocks and requesting memory in blocks, `zExpander` can use the `Glibc` allocator without concern of its space efficiency.

KV store performance. Recent research on KV cache is mostly on its performance, or on how to increase its peak throughput [28, 30, 32]. Their efforts include optimizing data structure to reduce processor cache misses, leveraging advanced hardware features, such as RDMA and Direct Cache Access, for fast networking, and efficient concurrency control. `zExpander` is complementary to the optimizations, as N-zone can be managed by any high-performance KV cache management to take advantage of the improvements.

Replacement strategy. KV cache systems usually employ light-weight replacement algorithms to identify and keep an active set of KV items in the cache, so that requests can be quickly processed. To this end, even the efficient LRU algorithm is replaced with a CLOCK algorithm in MemC3 [22] to remove two pointers for each KV items and associated operations on them in LRU. MICA [30] even uses a replacement policy similar to that for the set associative cache to minimize the cache footprint in each lookup. With this technical trend on replacement algorithms in KV caches, there is little room to accommodate more intelligent but more expensive replacement algorithms to reduce miss ratio. In contrast, `zExpander` takes a different approach to reduce miss ratio, which is to increase effective cache size. The replacement policy used in `zExpander`'s Z-zone is also of very low cost by only identifying victim items for replacement within individual blocks.

6. Conclusion

In this paper we propose `zExpander`, a KV cache with both high performance and substantially reduced misses at the same time. This is made possible by uniquely leveraging an observation common in KV-cache's workloads – accesses of the cache are highly skewed with a long tail. To enable an efficient system, we introduce a number of techniques, including batched compression, efficient indexing and data location on a complete binary trie, adaptive space allocation, and minimized data migration. More interestingly, `zExpander` can integrate any KV designs for high performance into its cache management with small code instrumentation. As an example, in one of the two prototypes, we add fewer than 100 lines of code into `memcached` to build the `M-zExpander` system. Porting more existing KV cache systems to `zExpander` is in our future work plan. We have extensively evaluated `zExpander` and demonstrated impressive results on both performance and miss reduction.

7. Acknowledgments

We are grateful to the paper's shepherd Dr. Donald Kossmann and anonymous reviewers who helped to improve the paper's quality. We thank Facebook Inc. and Eitan Frachtenberg for their donating servers and sharing Memcached traces, which allowed one of the authors (Song Jiang) to conduct extensive experiments for the evaluation. This work was supported by US National Science Foundation under CAREER CCF 0845711, CNS 1217948, and CNS 1527076.

References

[1] LZ4: Extremely Fast Compression algorithm. https://code.google.com/p/lz4/.

[2] LevelDB: A Fast and Lightweight Key/Value Database Library by Google. https://code.google.com/p/leveldb/.

[3] A Memory Allocator. http://g.oswego.edu/dl/html/malloc.html.

[4] TCMalloc : Thread-Caching Malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[5] complete binary tree. http://xlinux.nist.gov/dads/HTML/completeBinaryTree.html.

[6] Memcached: A distributed memory object caching system. http://memcached.org/.

[7] Protocol Buffers. https://en.wikipedia.org/wiki/Protocol_Buffers.

[8] How we cut down memory usage by 82 http://labs.octivi.com/how-we-cut-down-memory-usage-by-82/.

[9] How much text versus metadata is in a tweet? http://goo.gl/EBFIFs.

[10] zram: Compressed RAM based block devices. https://www.kernel.org/doc/Documentation/blockdev/zram.txt.

[11] The zswap compressed swap cache. https://lwn.net/Articles/537422/.

[12] A. Appleby. MurmurHash. https://sites.google.com/site/murmurhash/.

[13] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, New York, NY, USA, 2012. ACM.

[14] M. Blott, K. Karras, L. Liu, K. Vissers, J. Bär, and Z. István. Achieving 10Gbps Line-rate Key-value Stores with FPGAs. In *Presented as part of the 5th USENIX Workshop on Hot Topics in Cloud Computing*, Berkeley, CA, 2013. USENIX.

[15] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: An Operating System for Many Cores. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 43–57, Berkeley, CA, USA, 2008. USENIX Association.

[16] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[17] Z. Cheng, J. Caverlee, and K. Lee. You Are Where You Tweet: A Content-based Approach to Geo-locating Twitter Users. In *Proceedings of the 19th ACM International Conference on Information and Knowledge Management*, CIKM '10, pages 759–768, New York, NY, USA, 2010. ACM.

[18] 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, pages 143–154, New York, NY, USA, 2010. ACM.

[19] F. Douglis. The Compression Cache: Using On-line Compression to Extend Physical Memory. In *USENIX Winter*, pages 519–529, 1993.

[20] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 401–414, Berkeley, CA, USA, 2014. USENIX Association.

[21] J. Evans. jemalloc. http://www.canonware.com/jemalloc/.

[22] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 371–384, Berkeley, CA, USA, 2013. USENIX Association.

[23] T. H. Hetherington, M. O'Connor, and T. M. Aamodt. MemcachedGPU: Scaling-up Scale-out Key-value Stores. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 43–57, New York, NY, USA, 2015. ACM.

[24] X. Hu, X. Wang, Y. Li, L. Zhou, Y. Luo, C. Ding, S. Jiang, and Z. Wang. LAMA: Optimized Locality-aware Memory Allocation for Key-value Cache. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 57–69, Santa Clara, CA, July 2015. USENIX Association.

[25] R. Huggahalli, R. Iyer, and S. Tetrick. Direct Cache Access for High Bandwidth Network I/O. In *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ISCA '05, pages 50–59, Washington, DC, USA, 2005. IEEE Computer Society.

[26] S. Jiang and X. Zhang. LIRS: An Efficient Low Interreference Recency Set Replacement Policy to Improve Buffer Cache Performance. In *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '02, pages 31–42, New York, NY, USA, 2002. ACM.

[27] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA Efficiently for Key-value Services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 295–306, New York, NY, USA, 2014. ACM.

[28] S. Li, H. Lim, V. W. Lee, J. H. Ahn, A. Kalia, M. Kaminsky, D. G. Andersen, O. Seongil, S. Lee, and P. Dubey. Architecting to Achieve a Billion Requests Per Second Throughput on a Single Key-value Store Server Platform. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 476–488, New York, NY, USA, 2015. ACM.

[29] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A Memory-efficient, High-performance Key-value Store. In

Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, SOSP '11, pages 1–13, New York, NY, USA, 2011. ACM.

[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, pages 429–444, Berkeley, CA, USA, 2014. USENIX Association.

[31] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch. Thin Servers with Smart Pipes: Designing SoC Accelerators for Memcached. *SIGARCH Comput. Archit. News*, 41(3):36–47, June 2013.

[32] Y. Mao, E. Kohler, and R. T. Morris. Cache Craftiness for Fast Multicore Key-value Storage. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 183–196, New York, NY, USA, 2012. ACM.

[33] N. Megiddo and D. S. Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies*, FAST '03, pages 115–130, Berkeley, CA, USA, 2003. USENIX Association.

[34] C. Mitchell, Y. Geng, and J. Li. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 103–114, San Jose, CA, 2013. USENIX.

[35] 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 *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Lombard, IL, 2013. USENIX.

[36] J. Ou, M. Patton, M. D. Moore, X. Yuehai Xu, and S. Jiang. A Penalty Aware Memory Allocation Scheme for Key-value Cache. In *Proceedings of the 44th International Conference on Parallel Processing*, ICPP'15, 2015.

[37] S. M. Rumble, A. Kejriwal, and J. Ousterhout. Log-structured Memory for DRAM-based Storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST'14, pages 1–16, Berkeley, CA, USA, 2014. USENIX Association.

[38] Y. Wang, L. Zhang, J. Tan, M. Li, Y. Gao, X. Guerin, X. Meng, and S. Meng. HydraDB: A Resilient RDMA-driven Key-value Middleware for In-memory Cluster Computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 22:1–22:11, New York, NY, USA, 2015. ACM.

[39] X. Wu, Y. Xu, Z. Shao, and S. Jiang. LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data. In *2015 USENIX Annual Technical Conference*, NSDI'14. USENIX Association, 2015.

[40] Y. Xu, E. Frachtenberg, and S. Jiang. Building a high-performance key-value cache as an energy-efficient appliance. *Perform. Eval.*, 79:24–37, 2014.

[41] K. Zhang, K. Wang, Y. Yuan, L. Guo, R. Lee, and X. Zhang. Mega-KV: A Case for GPUs to Maximize the Throughput of In-memory Key-value Stores. *Proc. VLDB Endow.*, 8(11): 1226–1237, July 2015.

[42] Y. Zhou, Z. Chen, and K. Li. Second-Level Buffer Cache Management. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):505–519, June 2004.