Put an Elephant into a Fridge: Optimizing Cache Efficiency for In-memory Key-value Stores

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

Jian Liu
Computer Science & Engineering
Louisiana State University
jliu@csc.lsu.edu

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

ABSTRACT

In today's data centers, memory-based key-value systems, such as Memcached and Redis, play an indispensable role in providing high-speed data services. The rapidly growing capacity and quickly falling price of DRAM memory in the past years have enabled us to create a large memory-based key-value store, which is able to serve hundreds of Gigabytes to even Terabytes of key-value data all in memory. Unfortunately, CPU cache in modern processors has not seen a similar growth in capacity, still remaining at the level of a few dozens of Megabytes. Such an extremely low cache-to-memory ratio (less than 0.1%) poses a significant new challenge—the limited CPU cache is becoming a severe performance bottleneck that hinders us from fully exploiting the great potential of high-speed memory-based key-value stores.

To address this critical challenge, we propose a highly cache-efficient scheme, called *Cavast*, to optimize the cache utilization of large-capacity in-memory key-value stores. Our goal is to maximize cache efficiency and system performance without any hardware changes. We first present two light-weight, software-only mechanisms to enable user to indirectly control the cache content at application level. Then we propose a set of optimization policies to address several critical design issues that impair cache's efficacy in the current key-value store systems. By carefully reorganizing the data layout in memory, redesigning the hash indexing structure, and offloading garbage collection, we can effectively improve the utilization of the limited cache space. We have developed a module in Linux as a kernel-level support, and implemented two prototypes based on Memcached and Redis with the proposed Cavast scheme. Our experimental studies show promising results. On a 6-core Intel Xeon processor with only 15-MB cache, we can raise the cache hit ratio up to 82.7% with a very small cache-to-memory ratio (0.023%), and significantly increase the key-value system throughput by a factor of up to 4.2.

PVLDB Reference Format:
Kefei Wang, Jian Liu, and Feng Chen. Put an Elephant into a Fridge: Optimizing Cache Efficiency for In-memory Key-value Stores. *PVLDB*, 13(9): 1540-1554, 2020.
DOI: https://doi.org/10.14778/3397230.3397247

Proceedings of the VLDB Endowment, Vol. 13, No. 9
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3397230.3397247

1. INTRODUCTION

In the past decade, the amount of digital data has been explosively growing at an astonishing rate. According to a recent report from International Data Corporation (IDC), the global datasphere, which was 33 Zettabytes in 2018, will grow to 175 Zettabytes by 2025 [60]. As a key component in today's data centers, key-value store plays a crucial role in providing high-speed data services.

In order to handle the huge traffic of key-value queries at a high speed, cloud service providers heavily rely on *In-memory Key-value Stores*, represented by Memcached [10] and Redis [13], to offer high-throughput and low-latency services. Facebook, for example, has deployed a fleet of over 800 Memcached servers for their daily operations [15]. Other service providers, such as Google, Twitter, YouTube, similarly have a large-scale deployment in various production environments [14, 18, 19, 43].

1.1 Technical Trend and Challenges

The widespread adoption of in-memory key-value stores is driven by the recent technical breakthroughs in memory technologies (e.g., the 10-nm lithography process, 3D integrated circuits, and multi-die packaging), which allow the industry to make large-capacity DRAM memory at a rapidly decreasing cost. In the past 20 years, the unit capacity of DRAM memory has increased by over 100 times [12, 16], while the price (U.S. $/GB) has decreased by more than 200 times [53]. Its impact is enormous—our long-held dream of having the entire data store completely in memory now becomes an economically viable solution in practice.

In contrast to DRAM memory, CPU cache in modern multi-core processors has not seen a similar growth in capacity, still remaining at the level of a few dozens of Megabytes. For example, Intel's current top-tier processor, Xeon Platinum 9282, supports a maximum cache of only 77 Megabytes [6]. Such a small capacity is far from sufficient for memory-based data stores. Consider an entry-level server with 128 Gigabyte memory, the cache-to-memory ratio is below 0.06%, which is multiple orders of magnitude lower than the normally expected ratio for effective caching [17].

Even worse, due to the limited die space and the stringent power budget, we are unlikely to see a dramatic increase of on-chip cache size in processors of the near future, meaning that the already significant capacity gap (1,000x-10,000x) between cache and DRAM memory will continue to widen, at an accelerating pace. In other words, the limited cache space is not only constraining the performance of in-memory key-value stores in our present deployment, but also severely limiting their *scalability* in the future.

On the other hand, processor's cache is playing an unprecedentedly important role. In traditional systems, data are stored in secondary storage devices, such as hard drives. The huge speed gap (nanoseconds vs. milliseconds) between cache and storage dimin-

ishes the relative importance of the small on-chip cache, since the storage speed dominates the entire system performance. By contrast, memory-based key-value stores directly maintain the entire dataset in memory. The much smaller speed gap (nanoseconds vs. tens of nanoseconds) between cache and memory makes an efficient use of cache resources crucial for performance.

Such a technical trend poses a grand challenge to system designers and practitioners—given the hardware constraint, we must carefully optimize the key-value system design to fully exploit the available cache resources in the existing architecture. To achieve this goal, we need to address several critical challenges, in both hardware and software.

• **Hardware challenges**. Modern processors are designed for general applications. In the current architecture, virtual memory abstraction separates hardware and software. All the complex CPU internals, including the cache management, are made opaque to applications running atop it. Hardware automatically handles the management of on-chip cache; application software simply allocates and manages data objects in memory. Such a clear separation simplifies the design but creates a significant barrier, which hinders applications from being able to explicitly manage the cache content, leaving applications largely unaware of and unoptimized for maximizing the efficient use of cache space.

• **Software challenges**. Besides hardware, key-value systems carry several unique properties, which make an efficient use of the precious cache resources particularly difficult.

First, data accesses in key-value workloads are highly skewed. A recent study from Facebook reports a strong locality in real-world workloads [24]. A small amount of key-value items are frequently accessed (millions of times a day), while most are accessed only a few times after being created. A random blend of hot and cold data in cache would cause intensive cache conflicts, weakening the cache's efficacy and causing unnecessary memory accesses.

Second, keys and values are inherently distinct. For example, the size of a key is typically much smaller than a value [24], meaning that caching a value could be at the potential cost of evicting multiple keys. Moreover, upon a query, the keys must be loaded for comparison, while the value data are often unneeded (a mismatch is common). Indistinguishingly mixing and loading keys and values together into cache would cause a significant waste of the cache space and also premature evictions of the needed keys.

Third, the linked-list based hash indexing structure is cache unfriendly. In a key-value store, the indexing structure plays an important role and facilitates a quick lookup to locate the target data in memory. In this process, a sequence of random point reads happens along the list, incurring a chain of small and individual memory reads, amplifying the amount of data access, and polluting the cache with irrelevant data.

In short, the existing design of large-capacity in-memory key-value stores is inadequate in exploiting the very limited cache resources on the current hardware architecture. We need to find a solution to *put the "elephant" into the "fridge"*.

1.2 Making Key-value Store Cache Aware

In this paper, we present a highly cache-efficient scheme, called *Cavast*, to address the above-said cache challenges. Our goal is to identify a *software-only* solution to optimize the cache utilization of in-memory key-value stores, improving their performance without incurring additional cost. To the best of our knowledge, this study is the first software-based work on optimizing processor cache's efficacy for in-memory key-value stores.

Our key idea is to leverage the existing memory management mechanisms in operating systems to *virtually partition* the cache and *reorganize* data layout of key-value stores in memory. Without need for any hardware change, we are able to indirectly control the cache content and effectively avoid undesirable cache conflicts by exploiting the semantic knowledge of key-value store about the stored data and its internal structures.

Our optimizations are based on three key considerations. First, exploiting the relative temperatures of the key-value items, we can determine the best placement of a key-value item according to its temporal locality, and regroup the hot and cold data in cache. It allows us to lower the possibility of prematurely evicting hot data from the cache. Second, recognizing the distinction between keys and values, we propose to separate and reorganize key and value data in memory to solve the cache pollution and read amplification problems. This ensures that when a large value is fetched into the cache, it would not be at the potential cost of evicting many small keys, which avoids creating a storm of cache misses at a later time. Finally, we also propose a redesign of the hash indexing structure, making it cache-friendly. With a comprehensive package of all these software techniques, the cache efficiency of memory-based key-value stores can be significantly improved, without requiring any change to hardware.

We have implemented two prototypes based on Memcached and Redis, two representative key-value stores widely used in the industry. We have developed a light-weight, application-level solution to virtually partition the cache into user-controllable, large memory chunks for intra-page mapping. As an alternative, an OS kernel module has also been developed in Linux to assist creating a pool of pages with distinct cache mappings at a finer granularity.

Our experiments on Memcached and Redis show that our solution can greatly improve the cache efficiency for in-memory key-value data stores. Even without any hardware assistance, only by making small changes to the existing key-value store design, we can significantly improve the system performance: On a 6-core Intel Xeon processor with only 15-MB cache, we are able to raise the cache hit ratio up to 82.7% with an extremely small cache-to-memory ratio (0.023%), which in turn increases the key-value system throughput by a factor of up to 4.2.

The rest of the paper is organized as follows. Section 2 presents the motivations. Section 3 and 4 introduce the mechanism and policy design. Section 5 introduces the experimental setup. Section 6 and 7 present our two prototypes on Memcached and Redis. Section 8 discusses the overhead and other related issues. Related work is presented in Section 9. Section 10 concludes this paper.

2. MOTIVATIONS AND CHALLENGES

2.1 The Role of CPU Cache

CPU cache plays an important role in computer systems. It is designed to bridge the speed gap between processor and memory. In a typical Intel processor, an L3 cache hit latency is about 30–75 CPU cycles; a memory access, by contrast, takes much longer, typically 50–100 nanoseconds [1, 5, 44]. Caching the frequently accessed data can effectively filter out most memory accesses and improve performance.

The capacity of on-chip cache, due to the die space and production cost, is very limited. In modern processors, the *Last-level Cache* (LLC) is often only of a few dozen Megabytes, shared among the cores. When filled up, the Least Recently Used (LRU) algorithm is used for cache replacement. In this paper, cache refers to the LLC, unless otherwise noted.

Over the years, such a relatively small cache in processor has been proven effective and also cost-efficient for general applications. In fact, the amount of LLC is roughly about 2–2.5 MB per

core, which is generally regarded sufficient for serving the purpose of accelerating computation. However, when it comes to in-memory key-value stores, the long-standing cache architecture falls short in its very limited capacity. It is essentially because memory plays a fundamentally different role in such applications—unlike general applications, which mainly use memory as an intermediate layer between processor and storage, memory-based key-value stores use memory as a high-speed main storage media to accommodate the huge key-value dataset in complete. As a result, the on-chip cache has to cache data for a *disproportionately* large amount of memory, which could be of hundreds of Gigabytes to even Terabytes. Such an extremely low cache-memory ratio (e.g., 77-MB cache for 128-GB memory) results in a variety of issues, such as cache contention, thrashing, inability to scale, etc.

In the following, we use an example to illustrate the impact of CPU cache on memory-based key-value store performance.

2.2 In-memory Key-value Store

Our example case runs on a 6-core Intel Xeon E5-2630 system with 15-MB L3 cache (LLC) and 64-GB memory. We use the popular YCSB benchmark [29] with the default configuration to generate the key-value datasets in different scales (64 MB to 60 GB). Despite the distinct dataset sizes, the generated key-value items follow the same Zipfian distribution. We also ensure that the memory capacity is large enough to contain the dataset completely. We have obtained several interesting findings in our experiments on Memcached.

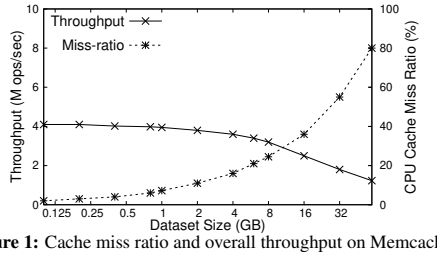


Figure 1: Cache miss ratio and overall throughput on Memcached.

As shown in Figure 1, as the dataset size increases, Memcached performance quickly drops, despite the fact that the datasets are all completely held in memory. In particular, with a 64-MB dataset, the throughput can reach 4.1 MOPS (million operations per second). As the dataset size increases to 60 GB, the throughput drops to 1.23 MOPS, which is a slow-down by a factor of 3.3.

To investigate the cause of this unusual and significant performance drop, we use Linux's `perf` tool [9] to collect hardware performance counters [4] and calculate the LLC miss ratio. In Figure 1, the dotted line plots the LLC miss ratios on the Memcached server. We can see a clear trend across the tests. As we increase the dataset size, the LLC miss ratio increases from 1.2% to nearly 80%, meaning that the cache is quickly approaching to being almost disabled as the dataset grows. The sharp increase of cache miss ratio explains the rapidly declining throughput—it is a result of ineffective caching.

This example demonstrates the strong impact of cache on the performance of in-memory key-value stores, urging us to carefully review the competence of the current key-value stores in performing and scaling on the existing hardware architecture.

2.3 Analysis and Discussions

Modern processors have multiple layers of on-chip caches. The LLC is typically a set-associative cache with the largest capacity. In a set-associative cache, the cache space is divided into multiple (e.g., 2,048) *Sets*. Each set is further divided into multiple *Cache Lines*. A cache line (typically 64 bytes) is the smallest unit for caching. A block of memory is mapped by the hardware to a specific set, but can be stored in any cache line of the set. If the set is filled up, a victim cache line is selected for eviction based on the LRU order. *Cache conflict* happens when multiple memory blocks are mapped to the same set and compete for available cache lines.

In-memory key-value store maintains its data and metadata structures all in memory. Both are subject to the caching effect. Several unique issues in the current design of key-value stores unfortunately undermine the efficacy of the CPU cache.

• **Issue #1: Disproportional key and value sizes**. Key-value stores adopt a simple data model: a key serves as a search index and uniquely identifies the data; the value simply holds the data content. Keys and values, by nature, are very different in many aspects, such as size. Typically, a key is of only several bytes. For example, some applications use an SHA-1 hash digest (20 bytes) or an MD5 hash digest (16 bytes) as a universally unique key. By contrast, a value is often larger and varies from bytes to Megabytes. Such a size difference has a strong implication for caching.

Consider a key of 16 bytes and a value of 512 bytes. When a value needs to be loaded into cache, we have two possible options: (1) evicting a 512-byte value, or (2) evicting 32 keys, each being of 16 bytes. Though both serve the same purpose of freeing up 512-byte cache space, the caching effects are different. The latter would result in a greater chance of cache misses in the future, since loading the evicted keys back would potentially incur 32 rather than only 1 memory access. Thus, mixing keys and values in cache would raise cache contention and severe cache conflicts.

• **Issue #2: Low cache utilization in hash indexing**. Hash table is a crucial data structure used for indexing data in key-value systems. A standard hash table manages an array of buckets. Each bucket maintains a pointer, either pointing to the next item in a linked list or indicating the list end (`NULL`). When searching for the target key, a bucket is read and the pointer is loaded to walk the list.

As the smallest unit for caching is a cache line (64 bytes), when a pointer is read, the entire 64-byte memory block, which contains the pointer and the pointers of adjacent buckets, has to be loaded as a whole into cache, despite the fact that only 8 bytes (the pointer) of the cache line is truly needed. Due to the random nature of hashing, the keys are evenly distributed over the buckets, so the adjacent buckets are mostly irrelevant, meaning that the rest 56 bytes of the cache line would be barely of any use, resulting in a waste of 87.5% cache space.

• **Issue #3: Read amplification with key-values**. When traversing a hash table list to find the target key-value item, we need to compare the item's key with the target key. If a match is found, the value is returned; otherwise we skip this item and follow the pointer to check the next one on the list until finding the key or reaching the end of the list.

When comparing the keys, the entire item, including both the key and the value, is loaded into cache. If the item's key does not match the target key, which is the most likely-to-happen case, the value part will be of no use, resulting in a *Read Amplification* problem. Read amplification is harmful, because it demands more memory bandwidth for transferring data over the bus and incurs extra latency. Loading irrelevant data also pollutes the cache, causing useful data to be evicted prematurely.

All the above-said issues severely damage the cache's efficacy. More importantly, when a key-value data store scales up, these problems would be even worse. We need a full reconsideration on the structural design of key-value stores.

3. MECHANISM

To address the above-said cache challenges, we propose *Cavast*, a **ca**che optimization scheme for memory-based key-**va**lue **st**ores. Following the system design principle, we first introduce a software-based cache partitioning *mechanism* that facilitates memory-based

key-value stores to indirectly control the cache content, and then a set of optimization *policies* that key-value stores can apply to leverage the mechanism for cache optimization.

Cache conflict happens when memory blocks are mapped to the same sets in cache. A cache-unfriendly, less valuable object (e.g., large, cold data) can pollute the cache and prematurely evict more valuable data. To address this issue, we divide the cache into multiple *virtual* partitions and allow applications to explicitly map conflicting data objects to distinct cache partitions, mitigating the collision in cache. We adopt a software-only approach by exploiting the existing mechanisms in Operating Systems (OS) to avoid disruptive hardware changes.

• **Page coloring**. In modern processors, the LLC is physically indexed. In order to maximize cache utilization, the hardware maps contiguous physical memory addresses to cache sets in a sequential manner. Operating system, which is responsible for virtual-to-physical mapping, adopts a mechanism, called *Page Coloring* (a.k.a. cache coloring). It works as follows.

Physical memory pages, and the corresponding cache sets, are assigned with different *Colors*, as shown in Figure 2. The OS tries to map an application's contiguous virtual memory pages to distinctly colored physical pages, which are correspondingly mapped to different sets in cache. The purpose is to spread an application's virtual pages uniformly over the cache.

Page coloring logically divides the cache. Horizontally, the cache sets with the same color form an independent partition of the cache. Cache lines in different sets have no effect on each other (i.e., no competition for space). If an application is aware of page colors, placing in-memory objects to distinctly colored pages would result in an effect that, the objects are mapped by the hardware to different sets in the cache, thus eliminating cache conflicts. The challenge is how to map a virtual page to a specific color.

• **Gaining control on cache**. As mentioned above, the cache colors are associated with *physical* memory addresses. Applications, due to the virtual memory abstraction, only see *virtual* addresses. We need a way to walk around this abstraction limit.

Figure 3a shows a 64-bit cache address and a 4-KB physical memory page address. A cache address consists of three parts, *Set Index*, *Line Offset*, and *Tag*. Set index is used to determine which set a particular cache line belongs to; Line offset is used to point to the specific byte in the cache line; Tag is used to compare with a target address. Since a cache line is fixed to 64 bytes, bit 0-5 are used as line offset. The number of bits used for set index is determined by the number of sets, typically 6 to 11 bits. On a typical Intel processor, bits 6-16 are used for indexing 2,048 sets in the LLC. More details can be found in prior works [39, 52, 67].

Because an application cannot decide on physical page allocation (the OS decides the virtual-to-physical mapping), it only has control on the lower 12 bits. This raises two issues. First, the application cannot guarantee to which color a virtual page would be mapped by the OS. Second, the application has very limited headroom to move the objects around within a 4-KB range. We find two possible solutions, as follows.

Option #1: Mapping with Hugepage. One solution is to enlarge the page size. Operating systems on modern processors support using a larger page size. For example, Linux on Intel processors supports two large page sizes, 2 MB and 1 GB, called *Hugepage* [8], which use the lower 21 or 30 bits as page offset, respectively. Hugepage enables us to have the page offset cover the entire set index, leaving all the bits for indexing into LLC (bits 6-16) visible to applications (see Figure 3b), which removes the need for page coloring. Since an application can see all cache colors in a large page, it can completely control the mapping of its memory objects to cache by carefully arranging the data layout inside a large virtual page.

In Cavast, we logically divide a 2-MB memory page into multiple *Columns*. Each column is of $64 \times N$ bytes, where N is the number of sets of the LLC. Horizontally, we divide it into $N \ / \ 64$ *Rows*. Each slice is called a *Tile*, whose size is 4,096 bytes. Taking a processor with 2,048 sets as an example, the column size would be 128 KB. A 2-MB page is sliced into 16 columns, and each column is divided into 32 rows. Figure 4 illustrates the structure.

Since a column spans all cache sets, two tiles of the same row in any columns share the same cache sets and may incur cache conflicts, while accessing two tiles in different rows would not incur any cache conflict. In our Memcached-based prototype, we use this approach for memory space management.

Option #2: Mapping with pre-allocated pages. Our second solution is to create a pool of pre-allocated pages with different page colors. We use the standard 4-KB page size. A simple module, `get_pgcolor`, is implemented in the OS kernel to support applications to query the color of a virtual memory page via a new system call. At the OS kernel level, the module translates a virtual page to a physical page, and returns the page color information; At the application level, we request the OS to allocate a number of random pages, which are clustered into multiple *Colored Page Pools*. Each pool contains pages of the same color. When an application needs to assign an object with a specific color, we can simply allocate the object in a page selected from the target color pool.

We note that the OS kernel does not guarantee to return a page of a specific color. However, requesting a number of contiguous memory pages would cover almost all colors due to the OS page coloring mechanism. If a colored page pool exhausts its pages, we can keep requesting the OS to allocate pages until we collect enough amount of pages of the specific color. The pages of the unwanted colors can be deallocated immediately.

This mechanism can simulate the column-row structure as described in the hugepage approach—each 4KB page of a color is a "tile"; the pages of the same color logically form a "row"; a group of pages, each having a distinct color, logically forms a "column". Similarly, accessing two pages from the same row (i.e., the same color) may incur cache conflicts, while accessing two pages from different rows incurs no cache conflicts.

This approach is simple and satisfies our needs. Although it is possible to modify the kernel code and directly request the OS to allocate a page of a specific color on demand [47, 48], it would pose intrusive and significant changes to the current memory management in the OS. Our solution only involves very minor changes and introduces minimum impact. In our Redis-based prototype, which allocates memory in small pieces individually, we use this approach to achieve fine-grained control on cache colors.

4. POLICY

Memory-based key-value store maintains a large amount of data and metadata in memory. Leveraging the Cavast mechanisms, we can manipulate the data layout in memory to achieve the desired caching effect. In this section, we discuss several general policies for cache optimizations in memory-based key-value stores.

4.1 Handling Hot and Cold Key-value Data

Key-value workloads are known for their highly skewed access patterns. Disregarding the locality difference and randomly placing data in cache would result in conflicts. Our first optimization policy is to avoid such conflicts and retain the hot data in cache.

In processor's cache, if a cache set is filled up, the LRU replacement evicts the *relatively* cold cache line. Data placement in cache can result in drastically different effects. If hot key-values are

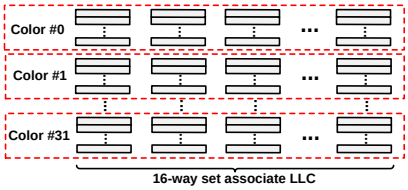
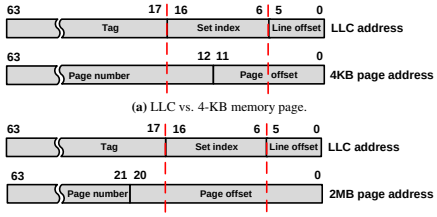


Figure 2: An illustration of Last Level Cache and page coloring. Each box represents a 64B cache line.



(a) LLC vs. 4-KB memory page.

(b) LLC vs. 2-MB memory page.

Figure 3: Address layout in LLC and memory page.

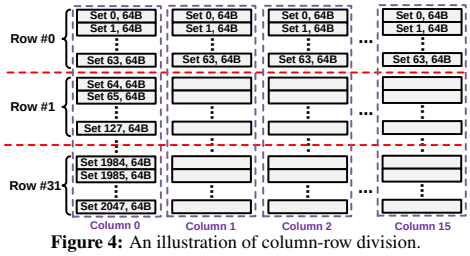


Figure 4: An illustration of column-row division.

cached together, a relatively less hotter item will be evicted, even though there might exist much colder items in other sets. Thus, to maximize the cache's efficacy, we desire to have a *mixed combination* of both hot and cold items in a "row" (see Figure 4). In other words, we desire to see that each row is filled up with both cold and hot data, which compete for the space within the row, and upon eviction, the victims would be the cold ones.

To accomplish such an effect, we first need to know the temperature (temporal locality) of the key-value items. Memcached maintains an LRU list per slab class to track each key-value item's relatively locality, while Redis maintains a pool of weak-locality (cold) key-values for eviction by sampling the dataset periodically. Leveraging these existing facilities, we can differentiate cold and hot data, and spread the data with similar locality across the rows. The process of relocating data is called *Re-partitioning*. In Section 6 and 7, we will discuss particular implementations in our Memcached and Redis prototypes.

4.2 Separating Key and Value Data

In current memory-based key-value stores, keys and values are placed together in memory (e.g., the `item` struct in Memcached). As so, keys and values are also loaded together into the cache, even if only the keys are needed (upon a mismatch). This is a significant waste of the limited cache space. Worse, since values are typically larger than keys, loading an unneeded value would be at the potential cost of prematurely evicting many small-size keys.

In Cavast, we restructure the in-memory layout of keys and values by placing them in *separate* rows. The effect we desire to achieve is that, in the cache, the keys are mapped to the same group of cache sets, while the values are mapped to a different group of cache sets. Such an arrangement brings two benefits. First, it protects relatively small keys from being polluted by large, unneeded values (so large values would evict each other). Second, since keys are grouped together, an access to a key would load the entire cache line into the cache, which in effect prefetches multiple keys in one access and further improves performance.

A side effect is that if the target key is found (a match), an extra memory access is needed to load the value data, which may increase the latency for individual requests. We have two methods to mitigate this effect. (1) *Parallel access*. We can maintain two pointers (one for the key and the other for the value) for each item, and use two threads to access them in parallel. If a match is found, the value is immediately returned; otherwise, the value is simply discarded. Since the two parallel memory accesses are overlapped, there is no extra delay for loading the value, but its limitation is the need for a second pointer to the value and the potential waste of bandwidth. (2) *Concurrent access*. We maintain two separate queues for key and value requests, each being served by dedicated threads. An incoming request is first put into the *key queue*, where the worker threads search the hash table for the target key. A pointer to the value is stored alongside the key, which avoids occupying the hash bucket space and making the hash table more compact. If a match is found, a request of retrieving the value is placed to the *value queue*, where the worker thread can follow the pointer to fetch and return the target value. Thus no value fetch happens until a match is found. Although it does not improve the latency for each individual request, using separate queues significantly improves throughput, which in turn decreases the average request latency. We have studied both approaches and find that the latter brings more benefits (see Section 6.2).

4.3 Cache-friendly Hash Indexing

Hash indexing structure is a crucial component in memory-based key-value stores. It is responsible for quickly locating the position of the key-value item in memory. Traditional hash table structure is very simple: An array of buckets divides the hash space into multiple segments. Each bucket maintains a linked list of key-value items. For a given key, it is first hashed into a bucket, and then traverses the linked list, following the pointers to locate the target key and the value. An example is illustrated in Figure 5a.

Such a classic hash table structure is simple and widely used but is inefficient for caching. First, traversing the linked list incurs a chain of small, random memory reads. Though each read only accesses a small amount of data, a complete cache line has to be loaded into the cache, causing read amplification. Second, since each pointer does not contain information about the key, an extra memory read is needed to load the key for confirmation. Third, when the hash table needs to be expanded, the entire structure has to be reconstructed, completely voiding the cache content.

We have developed three measures to optimize cache efficiency particularly for the hash indexing structure, described as follows.

• **Cacheline-based hash bucket**. Traditional hash table uses a set of small buckets, each containing only one 64-bit pointer as a list head pointing to the first key-value item in the list. A 64-byte cache line contains 8 buckets, corresponding to 8 linked lists.

In order to remove read amplification, we expand a hash bucket to contain a *Pointer Set*, which occupies a full cache line (64 bytes). Each bucket stores an array of up to eight 64-bit pointers, each pointing to a key-value item. A newly inserted pointer is stored in an empty slot of the 64-byte bucket. When a bucket is filled up, similar to the traditional approach, the hash table can be expanded by doubling the number of hash buckets. Later in this section, we will discuss a more efficient hash table expansion method.

The benefit is clear. As illustrated in Figure 5, the traditional hash table structure demands a sequence of memory accesses to traverse the list. Each access loads in a cache line but only uses a small amount of data. In our design, a cluster of 8 pointers can be loaded into the cache with only one single memory read, which minimizes the read amplification problem.

• **Tagging pointers**. In the traditional hash indexing structure, a pointer only specifies the memory location of the corresponding key-value item. An extra memory access is needed to load the key-value item for the key comparison. This incurs several issues. First, an extra memory read is needed, slowing down the search process. Second, since the key-value item and the pointer are stored separately, two cache lines are needed, polluting the cache. Third, finding a mismatch is a common case, meaning that most such additional costs are unnecessary.

To solve the above-said issues, we attach a *Tag* with each pointer to screen out the most unlikely keys before accessing the complete

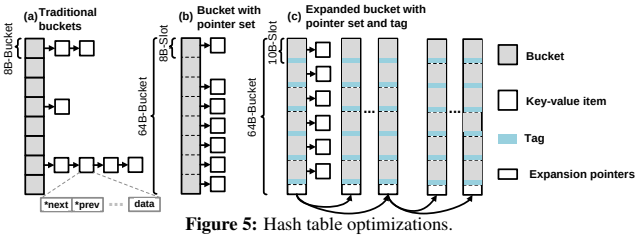


Figure 5: Hash table optimizations.

key-value item. We calculate a 16-bit hash digest as a tag summarizing the key (using Murmur3 hash [11]). The 2-byte tag is stored together with the 8-byte pointer as a slot of the pointer set in the bucket. Each bucket thus contains 6 slots in total, leaving 4 bytes for hash table expansion (see below). Upon a query, we first locate the hash bucket using Jenkins hash [7], then load the entire bucket in only one memory read and examine the pointers one by one (in cache). Only when we find the tag associated with a pointer matches the target key, we load the complete key-value item for a full comparison. This optimization reduces seven memory accesses to at most two memory accesses, and correspondingly, only needs two rather than seven cache lines. Prior works have used a similar hash-based method for fingerprinting data [20, 26, 65].

• **Localized hash table expansion**. The efficacy of a hash table decreases as the number of items held in a bucket increases (i.e., a bucket's list grows too long). To address this issue, the hash table needs to be expanded. Memcached, for example, expands its hash table by doubling the number of buckets, when the item-to-bucket ratio (a.k.a. load factor or fill ratio) exceeds 1.5.

Such an expansion process has two strong negative effects. First, the whole hash table needs to be reconstructed and the pointers need to be moved across buckets, which in effect invalidates the cached content completely, causing a disruptive performance drop. Second, the expansion happens over the entire hash table, including those underloaded buckets, which would create more holes (empty buckets) in memory, leaving hot buckets scattered in a sparse space and further reducing the efficiency of cache utilization.

To preserve the cache content and to keep the hash table structure stable, in Cavast, we adopt a lazy expansion approach, called *Localized Expansion*, to fully utilizing the spatial locality in the hash space. The key idea is to perform on-demand, small-scale expansion by only expanding the buckets that are heavily overloaded. When a 64-byte bucket is filled up, a set of sub-buckets (4 in our prototype) is created and linked to the original bucket (using the leftover 4 bytes for four pointers, each pointing to a sub-bucket). We support up to 255 sub-buckets in total before a global expansion is conducted. As the memory space for the 255 sub-buckets is pre-allocated, a 1-byte sub-bucket pointer is sufficient as an index into the array of sub-buckets. We reserve the sub-bucket pointer value "zero" as an expansion indicator. If all four pointers are zero, it means that the bucket is not expanded. Figure 5c illustrates this expansion structure.

When an item needs to be added into the expanded bucket, the least significant 2 bits of its tag are used as an index to determine the corresponding sub-bucket for insertion. Note that a sub-bucket could continue to expand in this manner, which eventually forms a *tree of sub-buckets*. Upon a query, we first search the bucket, if any non-zero sub-bucket pointer value is found, it means this bucket is expanded. We continue to look up the key in the linked sub-buckets. The tag associated with each pointer accelerates the searching. This process repeats until the list of buckets is traversed completely. An alternative is to only use one pointer, creating a *list of sub-buckets*. In Section 6, we will compare the two structures.

In essence, this bucket structure converts a fine-grained (item level) linked list structure into a tree structure with a grainier unit (64-byte pointer set bucket). This brings several important benefits. First, a bucket fits into a cache line perfectly, which minimizes the read amplification problem, since all the pointers of the bucket need to be examined anyway. Second, the linked list structure is condensed into a block of contiguous memory, which removes *point chasing* and the involved small, random memory accesses in different locations, eliminating the cache pollution problem. Third, when a bucket is overloaded, the changes can be confined in a subset of buckets, which protects the content in cache and stabilizes the hash table structure as well as performance. Forth, the tree structure divides the items, and together with the tags, accelerates the search. Finally, since the expansion only happens on overloaded buckets, we can avoid expanding those buckets that are largely empty or partially full, which avoids creating a sparse hash table with many holes in memory and preserves the spatial locality.

4.4 Cache-efficient Garbage Collection

In memory-based key-value stores, each key-value item is often associated with an *Expiration Time* to indicate its lifetime. A *Garbage Collection* (GC) process runs in the background and periodically scans the key-value items to recycle the memory space occupied by the expired items. For example, Memcached runs a service thread, called *LRU Crawler*, which constantly scans the LRU lists in the background. Redis applies a different policy. It attempts to keep the ratio of expired key-value items under 25%. In Redis, a service thread scans 200 items every second to remove the expired ones. This process does not cease running until less than 25% of the scanned items are found expired.

Although the GC process improves the memory space utilization, it foils the caching effort, since each scanned item has to be loaded into cache for validating its timestamp and then dropped. Such a one-time scan pattern is the worst case for caching, which pollutes the entire cache and evicts useful data out of cache [40]. We leverage the re-partitioning process (Section 4.1) to take a free ride for recycling the space. During re-partitioning, if any item is found expired, we simply skip it and reclaim its occupied space. This recycling is sufficient in normal conditions. The heavy-handed GC is only activated when the system is under severe memory pressure (e.g., lower than 5% in our prototype).

5. EXPERIMENTAL SETUP

We run our key-value store server on a Dell T620 server equipped with a 6-core Intel Xeon E5-2630 2.3 GHz processor with 15-MB L3 cache (LLC) and 64-GB 1600MHz DDR3 DRAM memory. We use two Lenovo TS440 ThinkServers as clients, each being equipped with a 4-core Intel Xeon E3-1245 3.4 GHz processor, 16-GB memory, and a 7,200 RPM 1-TB Seagate disk drive. Each client server runs 32 clients to generate requests to the key-value server. For sufficient network bandwidth, we configure a 20-Gbps network on the key-value server by bonding two 10-Gbps Ethernet ports together. Each client uses a 10-Gbps Ethernet connection to the server. We use Ubuntu 16.04 with Linux kernel 4.15 and Ext4 file system. For the hugepage setup, we configure the page size to 2 MB during the Linux boot time.

To test our design with faithful workloads, we synthesize three data sets, namely *APP*, *ETC*, and *SYS*, with three different key and value size distributions following a study of Facebook workloads [24]. Except that the value size in *ETC* follows a generalized Pareto distribution [3], all the other item sizes follow a generalized extreme value distribution [2]. Both distributions are found popular in Facebook workloads. Figure 6a shows the size distributions of keys and values in our datasets. We can see that each dataset has unique characteristics. For example, in *APP*, most of the keys are of about 31 bytes, and around 80% of the values are about 270 bytes.

SYS shows a similar trend but with more scattered sizes and a larger gap between keys and values. Most of the keys in *ETC* are from 20 to 50 bytes, this is in line with *APP* and *SYS*. However, unlike the other datasets, the value sizes in *ETC* are much more evenly distributed. We see more small values (the items with a value size of 11 byte or smaller account for about 40% of the entire dataset). The value sizes spread more evenly from 12 bytes to around 1 Kilobyte. Each dataset accounts for about 50 GB and is stored in memory in complete during our tests.

We use the Yahoo! Cloud Serving Benchmark (YCSB) [29] to generate workloads with two popular access patterns, *Zipfian* and *Hotspot*, to emulate realistic workloads [25, 30], and collect all the traces. Figure 6b shows the access distributions of the workloads. We find that *Zipfian* is relatively more skewed, a small portion of the keys serves the majority of requests. Whereas in *Hotspot*, the most popular 10% keys have similar hotness, leaving the rest 90% of keys to be similarly cold.

We use a homegrown tool, called *keystone*, to replay the workload traces against the key-value data stores. This tool allows us to precisely repeat a workload with any specified number of clients. We use the Linux's `perf` tool [9] to collect the hardware performance counters and calculate the LLC hit ratio.

6. CASE STUDY 1: MEMCACHED

Memcached is a widely deployed memory-based key-value store in industry. In Memcached, the basic memory management unit is called a *Slab*, which is a chunk of contiguous memory. Each slab is further divided into multiple *Slots* of a fixed size. Slabs with the same slot size are logically grouped into a *Slab Class*. Upon inserting a key-value item, a slab from the slab class with the smallest slot size that can accommodate the item is selected.

The current design of Memcached is sub-optimal for cache efficiency. Leveraging the Cavast mechanism, we enhance several key components in Memcached for cache optimizations.

6.1 Optimizations

• **Slab allocation**. Two mechanisms are provided in Cavast to enable application's indirect control on the cache, (1) statically using large-size hugepages, and (2) dynamically requesting pages with different colors. A key difference between the two methods is that, the former allows us to directly control a block of contiguous memory space (within a 2-MB page), which covers the range of all cache colors, while the latter gives us a fine-grained control on smaller 4-KB pages with distinct colors.

In Memcached, the slab system allocates memory in large chunks and then divides into fixed-size slots. Thus it suits the former method better. In our prototype, we allocate a 2-MB slab, which is a memory page. Since all cache colors are exposed to the application, we can manipulate the data layout within a slab to control their corresponding mapping locations in the cache.

• **Re-partitioning hot and cold data**. Memcached maintains an LRU list for each slab class. The hot (MRU) items are at the list head, and the cold (LRU) items are at the list tail. A limitation is that the LRU list could grow too long, causing several issues. First, list walking involves many memory accesses, increasing the chance of cache pollution. Second, the data relocation process would involve a large number of items, which are of different sizes. Third, the benefit of multi-threading is weakened due to the lock contention. Thus we split the LRU list into multiple smaller ones, each being attached to a slab (i.e., a 2-MB memory page).

For cache optimization, our goal is to lay out the hot and cold data *evenly* across all cache sets (so hot data would not evict each other). We use column as an allocation unit to contain a set of key-value items with similar locality. As the size of a column is 128 KB, a 2-MB Memcached slab can be divided into 16 physical *partitions*, each of which is a column. Accordingly, we divide the slab's LRU list into 16 logical *zones*, from hot to cold.

Our goal is to place the items of the same *logical* zone in the LRU list together in a *physical* partition in the slab. In this way, we can ensure that each cache set receives a roughly equal mix of hot and cold items. However, achieving this goal is non-trivial.

A critical challenge is that the position of key-value items on the LRU list dynamically changes. Upon an access, the item moves to the MRU position, pushing the other items one position down the list. It is unrealistic to update every item's physical location to accurately and immediately reflect its logical position in the LRU list, since it would raise excessive overhead.

We apply several rules to mitigate this problem. (1) *No intra-partition movement*. A key-value item does not change its physical location, unless it moves into a different zone on the LRU list, which indicates a significant change in locality. (2) *Lazy re-partitioning*. The cross-partition data movements are batched up to update the physical data layout periodically. (3) *Point-to-point movement*. We only move an item to a new position by swapping it with another item in the target partition to avoid the chaining effect (i.e., moving item A causes the movement of item B, which causes the movement of item C, and so on).

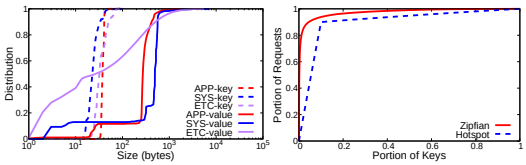
Algorithm 1 : Re-partitioning Process

$\boldsymbol{P_{logical}}$: Logical LRU partition;
$\boldsymbol{P_i(a)}$: Current physical partition of candidate a (Partition ID: i);
$\boldsymbol{P_t(a)}$: Target physical partition of candidate a (Partition ID: t);
for *every other* M *requests* **do**
  Check each candidate a in $P_{logical}$: bottom $\rightarrow$ top;
  if $P_i(a) \neq P_t(a)$ **then**
    Check each candidate b in $P_t(a)$;
    if $P_t(b) = P_i(a)$ **then**
      Select b as the *victim*;
      Swap a and b in physical partitions;
      goto done;
    end
    Find the LRU item c in $P_{i-1}(a)$ as the *victim*;
    Swap a and c in physical partitions;
    done;
  end
end

Algorithm 1 shows the re-partitioning process. It works as follows. Upon a change to the LRU list, for the affected items that move into a different LRU zone, we simply mark them as candidates for re-partitioning without further actions. Every M requests, we start from the bottom LRU zone. Assume an item, called an *initiator*, in partition P_i moves up to a (hotter) zone, which is corresponding to partition P_t, where $t < i$. We first try to find a *victim* in the target partition P_t that needs to move down to the initiator's current partition P_i. If found, we swap the victim and the initiator; Otherwise, we choose the LRU item in partition P_{i-1}, which is above the initiator's current partition, as the victim for swapping. This process repeats until all marked candidates are scanned. Figure 7 gives an illustration of before and after data re-partitioning.

Our per-slab LRU list design brings several benefits in this process. First, since the sizes of the slots in a slab are identical, swapping two slots in a slab does not need to consider the size mismatch problem. Second, all swapping operations are confined in a slab (i.e., a page), on which the application has full control. Third, the complexity of searching and identifying the initiator and victim is reduced and the search scope is limited in one slab.

• **Separating key and value data**. Keys and values have very distinct properties. Memcached stores both key and value together in



(a) Size distribution (b) Access distribution

Figure 6: Dataset and workload characterizations.

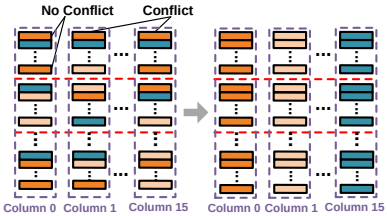


Figure 7: Data re-partitioning.

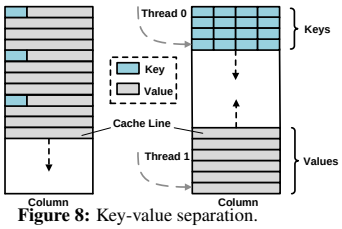


Figure 8: Key-value separation.

a slab slot as an `item` struct. As the key-value items are grouped in one slot after another sequentially, the keys and the values are mixed and interleavingly stored in a slab, raising the previously discussed cache pollution and read amplification problems.

To optimize the cache efficiency, we divide a slab into two separate regions, one for keys and the other for values. Figure 8 illustrates a 2-MB slab, each item of which has a 16-byte key and a 240-byte value. Each box represents a 64-byte cache line. The blue boxes represent the space occupied by keys, and the grey boxes represent the space taken by values. In the original design (on the left side), each item's key and value are stored together in memory and referenced with a single address. In Cavast (on the right side), all the keys are concentrated in a contiguous range of memory, being separated from the values. To connect a key with its corresponding value, we associate with each key a 3-byte pointer, pointing to the offset of the corresponding value within the slab. The hash index points to the key, from which we can locate the value.

A challenge is how to determine the sizes of the two regions in a slab. One simple solution is to assume certain static key-to-value size ratio based on workload analysis. However, workloads may change. This approach could incur substantial space waste, if a region cannot fully use the statically allocated space. We adopt an alternative solution, called *Head-to-head Allocation*. It works as follows. Rather than statically segmenting the slab space, we dynamically determine the space by allocating the key space from the top down (downwards), and allocating the value space from the bottom up (upwards). The two regions grow into each other until there is not enough space in between. If a head-to-head collision is to happen, a new slab should be allocated.

6.2 Performance Evaluations

We have implemented a prototype based on Memcached 1.5.12 by adding about 2,200 lines of C code. We configure Memcached and Cavast to use 6 worker threads on our 6-core test bed. The workloads are as described in Section 5.

• **Hot/cold data placement policies.** The hot and cold data placement in memory has a significant impact on the cache performance. To show an ideal case as a reference baseline, we first run a static re-partitioning test with three data placement policies, namely *random*, *row-partition*, and *column-partition*, which place the hot and cold items randomly, separately in different rows, and separately in different columns, respectively.

In this micro-benchmark, we first build the LRU lists offline using the traces. Based on the LRU lists, we can determine the temperatures of the key-value items and decide the data placement accordingly. Figure 9 shows the cache hit ratio and throughput with the Zipfian workloads. Cavast-Random, Cavast-Row, and Cavast-Column denote the three data placement policies, respectively.

We find the stock Memcached performs close to Cavast-Random. Both have a low hit ratio (around 20%) and a similar throughput (around 1 MOPS). It means that the stock Memcached's placement policy is no better than a random decision, completely disregarding the caching effect. Interestingly, separating hot and cold data in rows (Cavast-Row) performs even worse than random placement. This vividly illustrates the effect that hot key-value items are mapped to the same cache sets and evict each other, severely impairing the cache's efficacy. It also shows that the Cavast mechanisms enable us to control the cache, for good or bad effects. By contrast, Cavast-Column exploits the locality information and can avoid premature evictions of hot data, achieving a significantly higher hit ratio (64.3%) and throughput (2.865 MOPS).

• **Dynamic data re-partitioning.** In practical deployment, since the LRU list dynamically changes, key-value items need to move across partitions (re-partitioned). In our prototype, re-partitioning happens every 1,000 accesses (`SET` or `GET`) to a slab. In this experiment, we show the effect of dynamic data re-partitioning.

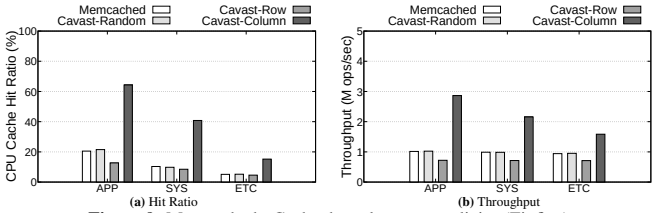
When the dataset is first loaded, all the items are randomly stored in memory. Therefore, Cavast undergoes two stages when serving the `GET` requests: (1) *Warm-up stage*. Initially, the key-value store needs to restructure the data layout in memory and relocate the key-value data according to the locality information. (2) *Stable stage*. After the initial warm up, the server continues to monitor changes in locality and make relatively small adjustments dynamically during run time. Naturally, we expect to see sub-optimal performance during the first stage. To obtain a complete picture, we show the performance in both stages. We identify the two stages by monitoring the average throughput of every 1,000 requests. If only minimal change (less than 5% difference for continuous 10,000 requests) is observed, we consider the system has entered the stable stage.

We first load the entire dataset into the Memcached server with random data placement policy, and then generate 500 million `GET` requests. Figure 10 shows the cache hit ratio and throughput for the stock Memcached and Cavast with Zipfian distributions. Cavast-WarmUp and Cavast-Stable denote the Cavast performance during the initial warm-up stage and the stable stage, respectively. We use the column-partition data placement policy for re-partitioning.

Our experimental results show that Cavast reorganizes the originally randomly placed data layout to a stable, column-based data placement status within 10 millions requests. It only takes 5-7 seconds to re-partition the entire 50-GB dataset, being warmed up for optimal cache efficiency. We also find that once reaching the stable status, our dynamic re-partitioning mechanism provides comparable performance to that in the previous static test, meaning that dynamic relocation involves minimal overhead. For example, Cavast-Stable has a hit ratio of 62.6%, 37.3%, and 15.2% for *APP*, *SYS*, and *ETC*, respectively, where the ideal-case results with static re-partitioning are 64.3%, 40.8%, and 15.5% (see Figure 9) .

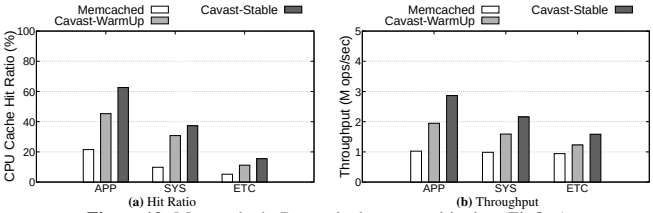
• **Key-value separation and multi-queue.** Separating keys and values can effectively mitigate the cache pollution and read amplification problems. In this set of experiments we only test on the key-value separation policy and keep other components unchanged.

In Section 4.2, we have discussed two approaches, parallel and concurrent access, to offset the negative effect caused by the extra memory access. We have evaluated both methods using *APP* with *Zipfian* distribution. We find that the 99th percentile latency for the stock Memcached is 2.1 ms. Our parallel access method achieves the same result. For concurrent access method, the tail latency is higher (2.4 ms), because for each request, the value is always read after the key. However, due to the throughput increase, when us-



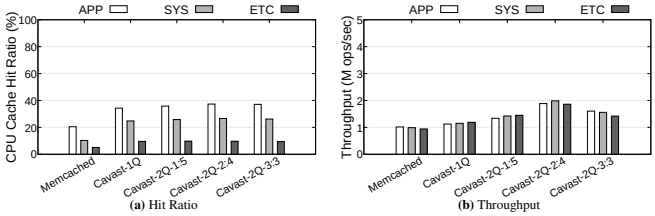
(a) Hit Ratio (b) Throughput

Figure 9: Memcached - Cache data placement policies (Zipfian).



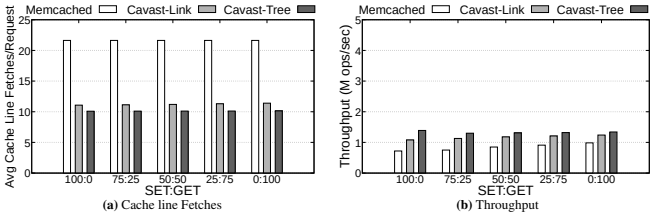
(a) Hit Ratio (b) Throughput

Figure 10: Memcached - Dynamic data re-partitioning (Zipfian).



(a) Hit Ratio (b) Throughput

Figure 11: Memcached - Key-value separation with queues (Zipfian).



(a) Cache line Fetches (b) Throughput

Figure 12: Memcached - Hash table restructuring (Zipfian).

ing concurrent access, the average latency actually decreases from 1.9 ms to 1.6 ms. As a tradeoff for space, we deem a slight (14.3%) increase in tail latency is acceptable. Thus we choose the concurrent access and further study its parallelization effect.

Figure 11 shows the results. In particular, Cavast-1Q shows separating key and value data without concurrency, meaning that the key and the value of an item are retrieved using one thread in a sequential manner, despite being stored separately. Cavast-2Q with suffix denotes the case using the multi-queue design with a different number of threads for the key and value queues. For instance, Cavast-2Q-1:5 means that 1 thread is used for the key queue and 5 threads for the value queue, making the total of 6 worker threads.

All the Cavast cases show a higher hit ratio than the stock Memcached. Since the keys and values are separated, it avoids evicting each other from the cache. Compared to the stock Memcached, Cavast increases the hit ratio by up to 16.8 percentage points (p.p.). Another interesting finding is that although Cavast with one queue or two queues have a similar cache hit ratio, they achieve very different throughputs. In our test, configuring 2 threads for the key queue and 4 threads for the value queue yields the most significant gain. Comparing Cavast 2Q-2:4 to Cavast-1Q, it results in a 56.4%–72.8% higher throughput. Our tests show that with key and value being separately stored, and with a reasonable resource allocation policy, Cavast is able to achieve much better performance than the stock Memcached.

• **Cache-friendly hash indexing.** The hash indexing structure impacts cache's efficacy. In this micro-benchmark test, we focus on studying the effectiveness of our new hash indexing structure, leaving other components unchanged. We design a set of tests using the *SYS* dataset (most keys are around 30 bytes and most values are around 500 bytes) with different `SET`/`GET` ratios. The read-only tests (ratio of 0:100) are performed after filling up the entire key-value store; the other tests are performed starting from an empty data store. For Memcached, we use the default 8-byte bucket size, which only contains the `h_next` pointer. For Cavast, we use the 64-byte, cache line based hash bucket (see Section 4.3) with localized expansion. We use the default load factor 1.5 as the threshold for expansion in both Memcached and Cavast. If the global load factor reaches 1.5, the hash table size is doubled.

In order to quantitatively measure how well Cavast mitigates read amplification in hash table, we calculate the average number of Cache Line Fetches (CLF) for each key-value request. In particular, each access to a hash table bucket accounts for one CLF; each key-value item access accounts for $\lceil \frac{item\_size}{cacheline\_size} \rceil$ CLFs. For example, a key-value item of 300 bytes needs 5 fetches of 64-byte cache lines. Figure 12 shows the average numbers of CLFs per request and the throughputs for 500 million requests.

For the stock Memcached, the average number of CLFs needed for one key-value item is about 21.6, which translates into 1,382 bytes, much larger (by a factor of 2.6) than the actual average key-value item size, 530 bytes. The result vividly illustrates our analysis of the read amplification problem caused by the inefficient design of the current hash table structure. By contrast, the degree of read amplification with Cavast is much lower. The average number of CLFs is 11.3 for Cavast-Link and 10.2 for Cavast-Tree, which is only half of that for the stock Memcached.

Comparing the two expansion structures, expanding a bucket using four sub-bucket pointers, which forms a tree of sub-buckets as described in Section 4.3, is more efficient than expanding using only one pointer, which forms a linked list of sub-buckets. It is because the tree structure splits the items into four sub-buckets. Only one sub-bucket needs to be searched, and less data need to be read from memory. Thus Cavast-Tree shows a 36%–93% throughput improvement over the stock Memcached, and a 8%–29% throughput increase over the linked list structure (Cavast-Link).

• **Put it all together.** In this test set, we enable and configure each component of Cavast with the optimal setting found in the previous tests. Namely, we use the column-partition data placement policy, concurrent accesses with two queues for serving the requests with 2 threads for keys and 4 threads for values, and a tree structure for the localized hash table expansion.

Figure 13-14 show the results. The performance difference is massive. Cavast improves the LLC hit ratio by up to 59.8 p.p., as compared to stock Memcached. The increased cache hit ratio in turn boosts the throughput by a factor of 3.9. In the best case scenario, Cavast shows a cache hit ratio of up to 81.3% for a 50-GB dataset with only 15-MB on-chip cache. It should be noted that such results are achieved without any hardware change and with an extremely small cache-to-memory ratio (only 0.023%). All the performance gains come solely from software optimizations.

We have also tested Cavast-GC, which is configured with all the above settings and the optimized GC (see Section 6.1). In the optimized GC, we turn off the background GC function in Memcached and recycle the expired items during data re-partitioning. Our optimization reduces the background GC's interference to cache, reaching a hit ratio of up to 82.7% and further improving the throughput by a factor of up to 4.2.

• **Worst-case study.** We have also conducted a worst case study, where the workload has a very weak locality, in which our optimizations could achieve little or no benefit. We create a set of workloads with *uniform* distribution. Figure 15 shows the cache hit ratio and throughput. Despite the weak locality, Cavast still tries to warm up the system by re-partitioning the key-value data in memory, which does not improve the cache hit ratio and incurs

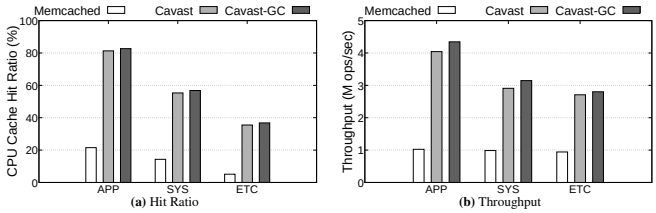


Figure 13: Memcached - Put it all together (Zipfian).

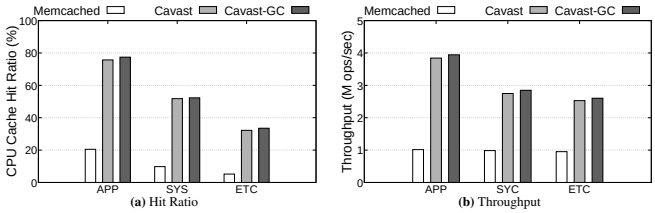


Figure 14: Memcached - Put it all together (Hotspot).

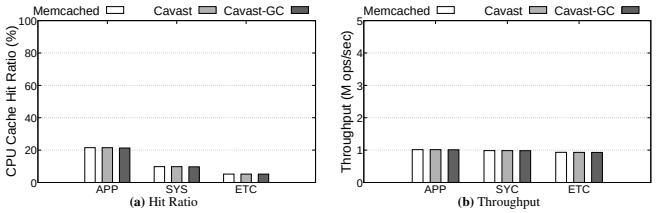


Figure 15: Memcached - Worst case study (Uniform).

overhead. However, due to the lazy re-partitioning, the system is only slightly impacted by the extra overhead. In fact, both Cavast and Cavast-GC show nearly identical performance as the baseline stock Memcached (performance difference within 2%).

7. CASE STUDY 2: REDIS

As another popular key-value store, Redis is different from Memcached in several aspects. First, unlike Memcached, which uses a multi-threaded design, Redis is single-threaded. Second, Redis is not slab based. It uses `zmalloc`/`zfree` for memory management. Third, Redis does not implement an LRU list for eviction. Instead, it maintains a pool of victim items via sampling.

Leveraging the Cavast mechanism, Redis can be optimized for cache efficiency. In this section, due to space constraint, we mainly focus on the aspects that are structurally different from Memcached. Other parts, such as hash indexing and GC optimizations, are similar to Memcached as we described in the prior section.

7.1 Optimizations

• **Memory space allocation**. Redis does not have a slab structure. It maintains a large chunk of pre-allocated memory space. The allocation requests for key-value items are served in their arrival order, disregarding other factors, such as size. As so, the key-value items are mixed and randomly placed at runtime.

To avoid intrusive modifications, we choose the approach of pre-allocating colored pages. We group the pre-allocated pages into 32 colors, each representing a "row" in the cache. A group of pages, each from an individual color, logically forms a "column". If needed, we can further group multiple colors together to a *Color Set* to divide the cache space at a coarser granularity. In this way, we can realize cache partitioning similar to that in Memcached.

• **Re-partitioning hot and cold data**. Redis also evicts key-value data based on their locality, but unlike Memcached, it does not implement an LRU list structure. It is partially because Redis does not partition memory space into slabs. Maintaining a complete LRU list would raise excessive overhead. To identify the victims for eviction, Redis associates with each item an *LRU clock*, which records the UNIX time when being accessed. In the background, a service thread periodically (10 times per second) samples a set of randomly picked items. The item with the smallest LRU clock is regarded as a cold item (victim). In default, the sample size is 5 items (i.e., five items are scanned each time). A pool of 16 victim items is maintained and ready for eviction.

We take advantage of this victim-identifying process to construct a "virtual" LRU list. Each time when the thread scans items, we collect the scanned items and sort them in the order of their LRU clocks. According to their access timestamps, we can form an LRU list of items, which account for 2 MB in total. Similar to our Memcached prototype, we divide the list into multiple logical zones, and then apply the re-partitioning algorithm to relocate the key-value items according to their temporal locality. In our prototype, we divide the items into 16 zones, each corresponding to a column partition.

• **Separating key and value data**. Redis stores keys and values in a different way than in Memcached. In Memcached, the slab system slices each 2-MB slab into fixed-size slots. Redis stores key-value items of different sizes in a mixed way. Thus, separating keys and values in the same page would be less suitable for Redis.

In Redis, we store keys and values in separate pages with different colors (i.e., rows). A challenge is how to divide the colors for storing key and value data. Without knowing a fixed key-to-value size ratio with Redis, we use an adaptive approach to dynamically determine the ratio of key pages to value pages, as follows.

Upon an insertion request, we split the item and separately place the key and the value parts to a key page and a value page, respectively. The hash index points to a block of two 8-byte pointers, which point to the key and the value parts, accordingly. If a page is consumed up, we take another free page with an unused color from the column. Note that once a color is used for keys, the color is reserved for keys thereafter. The same policy applies to the colors for values as well. Thus, after the first column is consumed, the ratio of key pages to value pages is determined and set thereafter. This approach automatically adapts to the workload during runtime. In our experiments, we find that the key-to-value ratio is rather stable and this adaptive solution works well.

7.2 Implementations and Evaluations

Our Redis-based Cavast implementation has two parts. An OS kernel module, `get_pgcolor`, is implemented in Linux kernel 3.18.12, with minimal changes, only about 200 lines of code. Another 2,000 lines of C code are added into Redis 4.10.14. Since Redis is a single-threaded key-value store, in order to fully exercise our prototype, we run 4 server instances simultaneously for performance evaluation.

• **Data re-partitioning with locality sampling.** Redis uses a sampling method to identify the cold key-value items. In this test, we study how well this approach emulates LRU and the effect of the sample size on cache hit ratio and system throughput.

We configure our Redis-based prototype with the sample size varying from 5 to 20 items each time. As a reference baseline, we build a real complete LRU list offline using the traces and partition the hot and cold data into LRU zones accordingly. We compare Cavast with different sample sizes to the stock Redis and the LRU baseline, which is considered as the ideal case. Figure 16 shows the cache hit ratio and throughput results. After reaching stable status, all Cavast versions show significantly higher cache hit ratios than the stock Redis. For example, in the Zipfian workloads with the *APP* dataset, we achieve a hit ratio of 46.7%–55.8%, and by contrast, the cache hit ratio of the stock Redis is only 22.5%.

We also find that a small sample size (5 items) consistently shows relatively worse result, since frequent sampling with a small sample size causes interference to the cache and disrupts the system. A reasonably large sample size (10 items each time) generally achieves a hit ratio close to the LRU baseline (57.3%), meaning that our sampling approach to emulating the LRU list works well in practice.

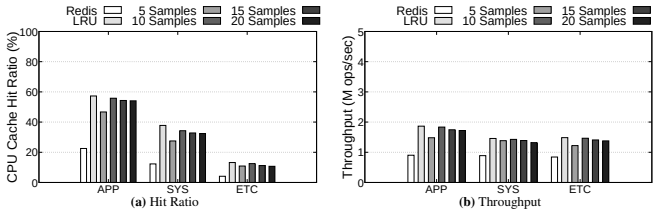


Figure 16: Redis - Data re-partitioning with sampling (Zipfian).

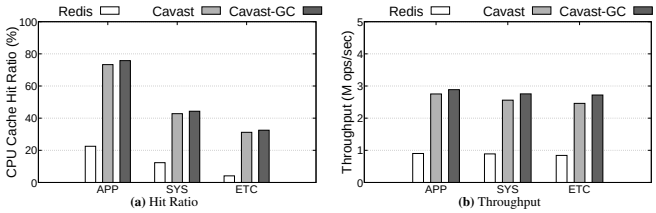


Figure 17: Redis - Put it all together (Zipfian).

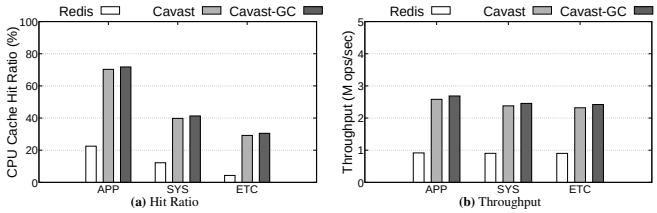


Figure 18: Redis - Put it all together (Hotspot).

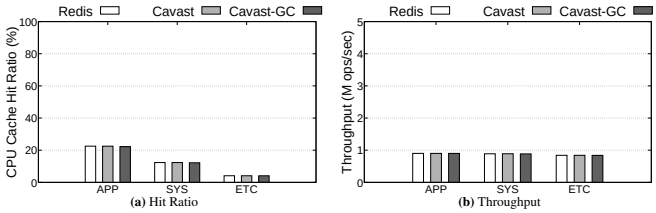


Figure 19: Redis - Worst case study (Uniform).

• **Put it all together.** Our Redis-based prototype implements the same functions as the Memcached-based prototype. Our experiments show that Redis can benefit from our optimizations as described previously. For example, separating keys and values improves cache hit ratio by up to 18 p.p., translating into a 56% increase in throughput. Similarly, the optimized hash indexing structure also brings a throughput improvement of 62%. It is noteworthy that Redis by default expands the hash table when the load factor reaches 1 (in contrast, 1.5 in Memcached). Even with a more aggressive expansion, Cavast still shows the benefits of using a carefully designed bucket and localized expansion.

Due to space constraint, we only show the performance of a fully configured prototype, which is optimized with column-based data partitioning, hot and cold data separation with a sample size of 10, key-value separation, and the restructured hash table for indexing. Figure 17 and 18 show the experimental results. We compare our prototype against the stock Redis. Another further improvement, denoted as Cavast-GC, enhances the GC operations. Similar to the test case in Memcached, Cavast-GC in Redis turns off the stock background GC thread, which samples at the rate of 10 times per second to look for expired key-value data, and only reclaims the expired items during data re-partitioning.

In Figure 17 and 18, we can observe significant performance gains with Cavast. In particular, for the Zipfian workload, Cavast achieves a cache hit ratio of 73.3%, 42.8%, and 31.2% for *APP*, *SYS*, and *ETC*, respectively. The stock Redis, in contrast, reports a cache hit ratio of only 22.5%, 12.3%, and 4.1%, respectively. The more efficient cache usage in turn brings a throughput improvement over the stock Redis by a factor of 3.1, 2.9, and 2.9, respectively. Comparing Cavast and Cavast-GC, we see a further improvement on cache hit ratio. Suspending the GC thread can reduce cache pollution and increase the hit ratio by up to 2.4 p.p., in which case the throughput is further increased by 4%.

• **Worst case study.** We have also tested Cavast using workloads with uniform distribution to study the worst case scenario. Results in Figure 19 show that Cavast performs no worse than the stock Redis even with a more complex design. The added kernel-level page coloring slightly lowers the performance by less than 1%.

8. DISCUSSIONS

8.1 Related Issues

Cavast provides an effective, software-only solution to optimize the cache usage in key-value stores. Here we discuss two related issues. (1) Effect of cache sets. Cache coloring enables us to control the mapping of memory objects to cache sets. As long as we have a reasonable number of cache sets for separating different data, having more cache sets in hardware is not expected to bring significantly more benefits. For example, optimizations, such as mixing hot and cold data, happens within a cache set; separating keys and values only divides the cache sets into two categories. Other optimizations, such as hash indexing restructuring, are also insensitive to the number of cache sets. (2) Side effect of hugepage. A larger page means more significant internal fragmentation. Thus it does not suit applications allocating memory in small pieces. In our current prototype, the hugepage is a system-wide setting, which affects all applications, meaning that any application could manipulate its data layout and disturb the shared cache, which might raise security and performance concerns (e.g., a malicious application can pollute the cache intentionally). A possible alternative solution (not implemented in current prototype) is to configure variable-sized pages for different applications, which is worth exploring in the future.

8.2 System Resource Usage

• **Memory usage.** In Cavast, the main memory overhead is for managing the separated key and value areas, which demands two pointers for each item. If using the hugepage approach (Cavast-Memcached), an extra 3-byte pointer is needed for indexing the value within the same 2-MB page of the key; if using the pre-allocated pages (Cavast-Redis), an extra 8-byte pointer is needed to locate the value stored in a different page.

Other minor memory overhead includes the following. (1) The zoned LRU list. In Cavast, each item needs additional 4 bits to record its logical zone (see Section 6.1) for re-partitioning. (2) The enhanced hash table. In Cavast, each item in the hash bucket is associated with a 2-byte tag. Each bucket also needs 4 bytes to link to its sub-buckets. In our current prototype, the memory overhead in Cavast-Memcached is 1.8%, 1.2%, and 0.6% for *APP*, *SYS*, and *ETC*, respectively, and 3.2%, 2.1%, and 1.1%, respectively in Cavast-Redis.

• **CPU usage.** The computational overhead of Cavast is mainly on handling the extra memory access for fetching the value data. The stock Memcached and Cavast-Memcached both use the same number of worker threads (6 in our prototype), but Cavast keeps a separate queue to serve the value requests rather than a single queue in Memcached. Unlike Memcached, the stock Redis only has one worker thread. Thus Cavast-Redis adds an extra thread to fetch the value data. Besides, Cavast also demands additional CPU resources for maintaining a background thread for data re-partitioning, and another background thread to expand hash table locally.

Table 1 shows a sample of the average CPU usage data collected in the "put it all together" experiments with *Zipfian* workloads. We find that the CPU usage overhead in Cavast is small. The worst case is observed with *ETC*, in which Cavast-GC uses an extra of 6.1% CPU resource over the stock Memcached.

Table 1: CPU Usage for Memcached, Redis, and Cavast with *Zipfian*.

	Memcached	Cavast	Cavast-GC	Redis	Cavast	Cavast-GC
APP	67.6%	70.2%	71.8%	58.2%	61.3%	62.2%
SYS	58.3%	61.1%	61.9%	55.9%	60.4%	60.8%
ETC	55.2%	59.7%	61.3%	56.3%	57.5%	57.8%

9. RELATED WORK

In recent years, key-value systems have gained popularity in both academia and industry [20, 28, 31–33, 38, 45, 46, 49, 51, 61–63, 66, 70]. Most of these prior works focus on how to efficiently utilize the limited DRAM space for performance. In order to reduce the high overhead of linked-list based LRU cache management for Memcached, Fan et al. [33] design an LRU-like caching algorithm based on the classic clock algorithm to lower the memory consumption (e.g., 1 bit per key). Rumble et al. [61] propose a new log-structured memory allocation mechanism to replace the conventional memory allocator (e.g., `malloc`) to enhance memory efficiency. Cidon et al. [28] use a dynamic partitioning policy instead of static partitioning for in-memory key-value cache to improve the memory utilization. Hu et al. [38] and Pan et al. [57] focus on using locality-aware memory allocation to replace the original naïve slab allocation in Memcached, hence improving memory efficiency. Wu et al. [64] dynamically partition the memory into two sub-zones. One zone is used for caching the hot data without compression to quickly serve the frequent accesses, and another larger one is used for holding the cold data with compression to improve the space efficiency. Zhang et al. [69] propose a hybrid data redundancy protection scheme to enhance the availability and efficiency of in-memory key-value store. Different from the above methods, which directly optimize memory usage for in-memory key-value systems, our work focuses on improving CPU cache efficiency thereby improving the performance.

There are also many researches focusing on designing cache-aware data structures and algorithms [21–23, 27, 34–37, 41, 54, 58, 59,68,72]. For example, Zuo et al. [72] find that the irrelevant small items of a hash table can pollute the cache line, causing low cache line efficiency. As such, they reorganize the hash table into an inverted binary tree, and the nodes in the same path of the tree used for resolving hash collision are stored in the continuous memory space to enhance the cache efficiency. Similarly, Zhang et al. [68] also point out the issue of low cache efficiency in regular hash tables. They modify the cache controller to enable L1 cache to fetch and serve an individual key-value pair rather than a cache line for improving the cache spacial locality. Hopscotch hashing [37, 41] also improves the cache line utilization by storing the items with hash collision in the consecutive buckets. Our work further carefully considers memory alignment with dedicated space and a tree-like structure for localized expansion. CSB+-Tree [35, 59] tries to store all the child nodes of a given node in continuous memory addresses for optimized cache line utilization. To enhance the cache space efficiency for in-memory string management, the prior works [21–23, 36] focus on designing a cache-aware trie by replacing the pointer-based data structure with an array. Our work focuses on solving cache conflicts and uses various techniques, such as tags and expansion tree, to maximize the cache utilization. Psaropoulos et al. [58] propose to mitigate the cache miss penalty and improve the performance of index joins by interleaving instruction stream at the language level using coroutines. Metreveli et al. [54] split a hash table into multiple partitions and distribute the partitions to different cores. An operation (e.g., insert, lookup) thus can be forwarded directly to the corresponding core for execution rather than fetching the entry and lock from that core and running locally. Benefiting from the cache affinity and less lock contention, a higher throughput can be achieved. Similarly, Farshin et al. [34] design a network I/O solution, called CacheDirector, which also considers the cache affinity to accelerate the network packet processing.

Most of the above-said prior works focus on increasing cache space efficiency by optimizing cache line utilization [21–23, 27, 36, 37, 41, 42, 59, 68, 72] for in-memory data structures such as hash table, tree, and trie. Some prior works try to alleviate cache miss penalty [58], or utilize the cache affinity to speed up data processing [34, 54]. Sharing a similar principle, our work focuses on addressing the cache efficiency issues specifically for in-memory key-value store by considering its unique properties and using a software-only solution and various techniques, such as relocating data, tagging keys, localizing hash table expansion, etc.

Cache partitioning has been extensively studied in multi-core applications. Noll et al. [56] point out that cache-insensitive operations, such as sequential scanning, can cause severe cache pollution for cache-sensitive operations, such as aggregation. They propose to allocate cache space separately for the two different operations, thus avoiding cache pollution and decreasing cache miss ratio. Lin et al. [47] propose to overcome the limitations of traditional simulation-based approaches by using page coloring to partition the cache in software, which enables a faithful evaluation of cache partitioning policies. Later, they further propose a light-weight hardware solution to reduce the overhead involved in the software-only cache partitioning [48] . To address the same problem, Zhang et al. [71] also propose a solution by enforcing coloring only on hot pages. Lu et al. [50] also present a software-based cache partitioning solution to optimize cache usage at the object level. Unlike the general-purpose solutions, we leverage cache coloring to enable the key-value stores to decide its data placement in cache, improving the cache utilization and performance.

A more recent work is called SDC [55]. Similar to our work, SDC also recognizes the cache under-utilization problem in key-value stores. They propose a new hardware support in processors by revising the cache management to allow application software to explicitly manage the cache as a look-aside buffer. As a hardware solution, this method is largely orthogonal to our work. We aim to provide a software-only solution, leveraging the existing available mechanisms in the OS to virtually partition the cache and optimize cache efficiency accordingly. In fact, an essential goal of our work is to avoid any hardware changes, which makes it practically and immediately applicable to real applications.

10. CONCLUSION

Memory-based key-value system is essential in data centers. Unfortunately, its performance potential has not been fully exploited due to the inefficient use of the very limited CPU cache space. As memory capacity continues to increase, the huge capacity gap between cache and memory poses a significant challenge in performance and scalability. In this paper, we present a highly cache-efficient scheme, called Cavast, to optimize the cache utilization in key-value systems. We have developed two prototypes based on Memcached and Redis. Our experimental results show that as a versatile design, Cavast can be seamlessly adopted into the existing systems, and substantially improve the cache efficacy and the system performance.

Acknowledgments

We thank the anonymous reviewers for their constructive feedback and insightful comments. We also thank Dr. John C. McCallum for collecting the memory price data for years. This work was partially supported by the U.S. National Science Foundation under Grants CCF-1453705, CCF-1629291, and CCF-1910958.

11. REFERENCES

[1] CAS latency. https://en.wikipedia.org/wiki/CAS_latency.

[2] Generalized extreme value distribution. https://en.wikipedia.org/wiki/Generalized_extreme_value_distribution.

[3] Generalized Pareto distribution. https://en.wikipedia.org/wiki/Generalized_Pareto_distribution.

[4] Hardware performance counter. https://en.wikipedia.org/wiki/Hardware_performance_counter.

[5] Intel Skylake. https://www.7-cpu.com/cpu/Skylake.html.

[6] Intel Xeon Platinum 9282. https://ark.intel.com/content/www/us/en/ark/products/194146/intel-xeon-platinum-9282-processor-77m-cache-2-60-ghz.html.

[7] Jenkins Hash. https://en.wikipedia.org/wiki/Jenkins_hash_function.

[8] Linux hugepage. https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt.

[9] Linux Perf. https://en.wikipedia.org/wiki/Perf_(Linux).

[10] Memcached. https://memcached.org.

[11] MurmurHash3. https://github.com/aappleby/smhasher/wiki/MurmurHash3.

[12] Random-access memory. https://en.wikipedia.org/wiki/Random-access_memory#Timeline.

[13] Redis. https://redis.io.

[14] Redis-based applications. https://techstacks.io/tech/redis.

[15] Scaling memcached at Facebook. https://www.facebook.com/notes/facebook-engineering/scaling-memcached-at-facebook/39391378919/.

[16] Synchronous dynamic random-access memory (SDRAM). https://en.wikipedia.org/wiki/Synchronous_dynamic_random-access_memory.

[17] The 10% rule for VSAN caching, calculate it on a VM basis not disk capacity! http://www.yellow-bricks.com/2016/02/16/10-rule-vsan-caching-calculate-vm-basis-not-disk-capacity/.

[18] Twemcache. https://github.com/twitter/twemcache.

[19] A. Adya, R. Grandl, D. Myers, and H. Qin. Fast key-value stores: An idea whose time has come and gone. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '19)*, pages 113–119, 2019.

[20] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*, pages 1–14, 2009.

[21] N. Askitis and R. Sinha. HAT-trie: A cache-conscious trie-based data structure for strings. In *Proceedings of the 30th Australasian Conference on Computer Science*, pages 97–105, 2007.

[22] N. Askitis and R. Sinha. Engineering scalable, cache and space efficient tries for strings. *The VLDB Journal*, 19(5):633–660, 2010.

[23] N. Askitis and J. Zobel. Redesigning the string hash table, burst trie, and BST to exploit cache. *Journal of Experimental Algorithmics (JEA)*, 15(1):1–61, 2011.

[24] 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 SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '12)*, volume 40, pages 53–64, 2012.

[25] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *Proceedings of IEEE Conference on Computer Communications (INFOCOM '99)*, volume 1, pages 126–134, 1999.

[26] F. Chen, T. Luo, and X. Zhang. CAFTL: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST '11)*, San Jose, CA, Feb 15-17 2011.

[27] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Making pointer-based data structures cache conscious. *Computer*, 33(12):67–74, 2000.

[28] A. Cidon, D. Rushton, S. M. Rumble, and R. Stutsman. Memshare: A dynamic multi-tenant key-value cache. In *Proceedings of 2017 USENIX Annual Technical Conference (USENIX ATC '17)*, pages 321–334, 2017.

[29] 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, 2010.

[30] C. R. Cunha, A. Bestavros, and M. E. Crovella. Characteristics of WWW client-based traces. Technical report, Boston University Computer Science Department, 1995.

[31] B. Debnath, S. Sengupta, and J. Li. FlashStore: High throughput persistent key-value store. *PVLDB*, 3(2):1414–1425, 2010.

[32] B. Debnath, S. Sengupta, and J. Li. SkimpyStash: RAM space skimpy key-value store on flash-based storage. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data (SIGMOD '11)*, pages 25–36, 2011.

[33] 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 Symposium on Networked Systems Design and Implementation (NSDI '13)*, pages 371–384, 2013.

[34] A. Farshin, A. Roozbeh, G. Q. Maguire Jr, and D. Kostić. Make the most out of last level cache in Intel processors. In *Proceedings of the Fourteenth EuroSys Conference (EuroSys '19)*, pages 1–17, 2019.

[35] R. A. Hankins and J. M. Patel. Effect of node size on the performance of cache-conscious B+-trees. In *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of computer systems (SIGMETRICS '03)*, pages 283–294, 2003.

[36] S. Heinz, J. Zobel, and H. E. Williams. Burst tries: A fast, efficient data structure for string keys. *ACM Transactions on Information Systems (TOIS)*, 20(2):192–223, 2002.

[37] M. Herlihy, N. Shavit, and M. Tzafrir. Hopscotch hashing. In *Proceedings of International Symposium on Distributed Computing (DISC '08)*, pages 350–364, 2008.

[38] X. Hu, X. Wang, Y. Li, L. Zhou, Y. Luo, C. Ding, and Z. Wang. LAMA: Optimized locality-aware memory allocation for key-value cache. In *Proceedings of 2015*

USENIX Annual Technical Conference (USENIX ATC '15), pages 57–69, 2015.

[39] R. Hund, C. Willems, and T. Holz. Practical timing side channel attacks against kernel space ASLR. In *Proceedings of 2013 IEEE Symposium on Security and Privacy*, pages 191–205, 2013.

[40] 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)*, pages 323–336, 2005.

[41] R. Kelly, B. A. Pearlmutter, and P. Maguire. Lock-free hopscotch hashing. In *arXiv preprint arXiv:1911.03028*, 2019.

[42] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, and et al. FAST: Fast architecture sensitive tree search on modern CPUs and GPUs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10)*, pages 339–350, 2010.

[43] M. C. Lee, F. Y. Leu, and Y. P. Chen. Pareto-based cache replacement for YouTube. In *World Wide Web*, pages 1523–1540, 2015.

[44] D. Levinthal. Performance analysis guide for Intel Core i7 processor and Intel Xeon 5500 processors. `https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf`.

[45] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A memory-efficient, high-performance key-value store. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 1–13, 2011.

[46] 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 Symposium on Networked Systems Design and Implementation (NSDI '14)*, pages 429–444, 2014.

[47] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *Proceedings of 14th IEEE International Symposium on High Performance Computer Architecture (HPCA '08)*, pages 367–378, 2008.

[48] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Enabling software management for multicore caches with a lightweight hardware support. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC '09)*, page 14, 2009.

[49] G. Lu, Y. J. Nam, and D. H. Du. BloomStore: Bloom-filter based memory-efficient key-value store for indexing of data deduplication on flash. In *Proceedings of 2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST '12)*, pages 1–11, 2012.

[50] Q. Lu, J. Lin, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Soft-OLP: Improving hardware cache performance through software-controlled object-level partitioning. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques (PACT '09)*, pages 246–257, 2009.

[51] L. Marmol, S. Sundararaman, N. Talagala, and R. Rangaswami. NVMKV: A scalable, lightweight, FTL-aware key-value store. In *Proceedings of 2015 USENIX Annual Technical Conference (USENIX ATC '15)*, pages 207–219, 2015.

[52] C. Maurice, N. Le Scouarnec, C. Neumann, O. Heen, and A. Francillon. Reverse engineering Intel last-level cache complex addressing using performance counters. In *International Symposium on Recent Advances in Intrusion Detection*, pages 48–65, 2015.

[53] J. C. McCallum. Memory prices 1957+. `https://jcmit.net/memoryprice.htm`.

[54] Z. Metreveli, N. Zeldovich, and M. F. Kaashoek. CPHash: A cache-partitioned hash table. *ACM SIGPLAN Notices*, 47(8):319–320, 2012.

[55] F. Ni, S. Jiang, H. Jiang, J. Huang, and X. Wu. SDC: A software defined cache for efficient data indexing. In *Proceedings of the ACM International Conference on Supercomputing (ICS '19)*, pages 82–93, 2019.

[56] S. Noll, J. Teubner, N. May, and A. Böhm. Accelerating concurrent workloads with CPU cache partitioning. In *Proceedings of 2018 IEEE 34th International Conference on Data Engineering (ICDE '18)*, pages 437–448, 2018.

[57] C. Pan, L. Zhou, Y. Luo, X. Wang, and Z. Wang. Lightweight and accurate memory allocation in key-value cache. *International Journal of Parallel Programming*, 47(3):451–466, 2019.

[58] G. Psaropoulos, T. Legler, N. May, and A. Ailamaki. Interleaving with coroutines: A practical approach for robust index joins. *PVLDB*, 11(2):230–242, 2017.

[59] J. Rao and K. A. Ross. Making B+-trees cache conscious in main memory. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD '00)*, pages 475–486, 2000.

[60] D. Reinsel, J. Gantz, and J. Rydning. Data age 2025: The digitization of the world from edge to core. *IDC White Paper*, 2018.

[61] 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, 2014.

[62] 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)*, pages 391–405, 2017.

[63] K. Wang and F. Chen. Cascade mapping: Optimizing memory efficiency for flash-based key-value caching. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '18)*, pages 464–476, 2018.

[64] X. Wu, L. Zhang, Y. Wang, Y. Ren, M. Hack, and S. Jiang. zExpander: A key-value cache with both high performance and fewer misses. In *Proceedings of the Eleventh European Conference on Computer Systems (Eurosys '16)*, pages 1–15, 2016.

[65] L. Xu, A. Pavlo, S. Sengupta, and G. R. Ganger. Online deduplication for databases. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*, page 1355–1368, 2017.

[66] S. Xu, S. Lee, S. W. Jun, M. Liu, and J. Hicks. BlueCache: A scalable distributed flash-based key-value store. *PVLDB*, 10(4):301–312, 2016.

[67] Y. Yarom, Q. Ge, F. Liu, R. B. Lee, and G. Heiser. Mapping the Intel last-level cache. *Cryptology ePrint Archive, Report 2015/905*, 2015.

[68] G. Zhang and D. Sanchez. Leveraging caches to accelerate

hash tables and memoization. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '19)*, pages 440–452, 2019.

[69] H. Zhang, M. Dong, and H. Chen. Efficient and available in-memory KV-store with hybrid erasure coding and replication. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16)*, pages 167–180, 2016.

[70] 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. *PVLDB*, 8(11):1226–1237, 2015.

[71] X. Zhang, S. Dwarkadas, and K. Shen. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys '09)*, pages 89–102, 2009.

[72] P. Zuo and Y. Hua. A write-friendly and cache-optimized hashing scheme for non-volatile memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 29(5):985–998, 2017.