Catalyst: Optimizing Cache Management for Large In-memory Key-value Systems

Kefei Wang*
Gannon University
wang039@gannon.edu

Feng Chen
Louisiana State University
fchen@csc.lsu.edu

ABSTRACT

In-memory key-value cache systems, such as Memcached and Redis, are essential in today's data centers. A key mission of such cache systems is to identify the most valuable data for caching. To achieve this, the current system design keeps track of each key-value item's access and attempts to make accurate estimation on its temporal locality. All it aims is to achieve the highest cache hit ratio. However, as cache capacity quickly increases, the overhead of managing metadata for a massive amount of small key-value items rises to an unbearable level. Put it simply, the current fine-grained, heavy-cost approach cannot continue to scale.

In this paper, we have performed an experimental study on the scalability challenge of the current key-value cache system design and quantitatively analyzed the inherent issues related to the metadata operations for cache management. We further propose a key-value cache management scheme, called *Catalyst*, based on a highly efficient metadata structure, which allows us to make effective caching decisions in a scalable way. By offloading non-essential metadata operations to GPU, we can further dedicate the limited CPU and memory resources to the main service operations for improved throughput and latency. We have developed a prototype based on Memcached. Our experimental results show that our scheme can significantly enhance the scalability and improve the cache system performance by a factor of up to 4.3.

PVLDB Reference Format:
Kefei Wang and Feng Chen. Catalyst: Optimizing Cache Management for Large In-memory Key-value Systems. PVLDB, 16(13): 4339 - 4352, 2023.
doi:10.14778/3625054.3625068

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/catalyst-kv/code.

1 INTRODUCTION

In recent years, we have witnessed an unprecedented data explosion. According to International Data Corporation (IDC), the global datasphere will grow from 84 Zettabytes in 2021 to 221 Zettabytes by 2026 [20]. Much of these data are unstructured data in forms of key values. In order to provide high-throughput and low-latency services, Internet service providers, such as Google, Meta, and Twitter, rely on *In-memory Key-value Cache*, represented by Memcached [4] and Redis [7], to handle a huge amount of key-value queries.

An important technical trend behind the wide-spread adoption of in-memory key-value cache systems is the fast-pace advancement of memory technologies. In the past twenty years, the industry has made tremendous progress in increasing DRAM memory capacity and reducing its cost. Benefiting from several important technology breakthroughs, such as the 10-nm lithography process, 3D stacking, and multi-die packaging, the capacity of DRAM memory has increased by more than 100 times, while the price (USD per MB) has significantly decreased at a similar magnitude [6, 9, 47].

On one hand, such a giant leap in memory hardware technology brings an enormous opportunity allowing us to build long-desired large in-memory key-value cache systems to accommodate more data in memory for high-speed data access. On the other hand, it also raises a critical challenge to us—*As the memory capacity grows by 100 times larger, how can we manage a massive amount of small key-value items still in a highly efficient way?*

1.1 Critical Issues

At the heart of an in-memory key-value cache system is cache space management. It is responsible for identifying and caching the most likely-to-be-reused (hot) data in memory while evicting the cold data that are unlikely to be accessed again. This caching decision has a direct impact on the system performance—a query hit in cache can be quickly served from fast memory, while a query missed in cache has to be diverted to the backend databases or data stores and the data has to be retrieved from block storage devices, which are multiple orders of magnitude slower than DRAM memory. Such delays could cause congestion on the backend servers and further propagate to other components in the whole data center system, resulting in a broad, system-wide performance impact. Thus, as the first line of defense, an effective and efficient key-value cache system plays a key role in modern data center systems.

Traditionally, achieving high *hit ratio* is the most important (if not the only) goal in the key-value cache system design. For this purpose, the cache system closely tracks each key-value item's access history to accurately estimate different items' importance for caching according to their access locality. However, realizing such a goal is not at no cost. Memcached, for example, adopts a simple Least Recently Used (LRU) based replacement policy. All the key-value items of a slab class are maintained in a doubly linked list. Upon access, the key-value item is moved to the list head; upon eviction, the item at the list tail is removed as a victim. It works well when dealing with a small cache, but unfortunately, as cache capacity quickly increases, even such a seemingly simple scheme is difficult to scale. As the number of key-value items increases, the overhead involved in caching-related metadata operations sharply

*The work was done while Kefei Wang was at Louisiana State University.

Proceedings of the VLDB Endowment, Vol. 16, No. 13 ISSN 2150-8097.
doi:10.14778/3625054.3625068

increases to an unbearable level. In experiments, we find that for a 100-GB Memcached system, disabling the LRU replacement can immediately observe a performance boost by a factor of 3.3, which strongly indicates that the overhead of such metadata operations with a large cache capacity is problematic.

In this work, our study reveals several critical issues inherent in the current key-value cache system design. First, the current cache system design mistakenly treats metadata operations for caching as "trivial-cost" operations and directly embeds them in the foreground service operations. The unfortunate consequence is that such non-essential maintenance work, which is not directly related to serving client queries, cumulatively incurs a significant amount of overhead, and even worse, it lies on the critical path, directly affecting the client-perceived performance. Second, in order to accurately estimate the key-value items' relative values for caching in terms of access locality, the current design relies on the lock-protected structure, which enforces frequent metadata operations to be performed in a serial manner, disabling the possibility of parallelizing metadata operations. Third, the current design adopts an unnecessarily fine-grained management by associating each small key-value item with an amount of metadata for tracking its access history. Though each being small individually, these metadata structures in aggregate incur non-trivial spatial overhead, especially considering that most key-value systems are dominated by small items (e.g., a few hundreds of bytes), not to mention that these memory space could be used for caching more data. Lastly, the constantly happening metadata operations are also directly competing with the latency-sensitive foreground service operations for very limited CPU resources in terms of both CPU cycles and the small on-chip cache space. Such an effect is particularly strong to in-memory key-value cache, which serves client queries directly from memory at nanosecond speed.

Due to the above-said issues, as the cache capacity quickly scales up, the traditional key-value cache system design becomes increasingly difficult to remain effective and efficient as desired. Put it simply, the traditional key-value cache design is *unscalable*.

1.2 A Scalable Cache Management Scheme

In this work, we present a highly efficient in-memory key-value cache system design, called *Catalyst*. To the best of our knowledge, this study is the first work focusing on addressing the scalability challenge of the metadata management for caching in large-capacity in-memory key-value cache systems.

Our design is based on three key considerations. First, we take the metadata operations for cache management out of the critical path, allowing queries to return immediately after finishing the essential service functions (e.g., indexing and data loading). The caching-related metadata operations are performed in an asynchronous manner. Second, we organize the metadata in a compact, parallelizable structure, which summarizes the data access history and can be updated and queried in parallel for high throughput. This ensures the metadata operations not become the bottleneck as the cache size grows. Third, we separate metadata operations (the "maintenance" work) from regular service operations (the "real" work) and move them off the CPU to avoid the competition for the limited CPU resources, minimizing the interference. Finally and most importantly, we relax the precision requirement for cache replacement. In a very large key-value cache, strictly following the rule of finding and evicting the "coldest" item is excessively costly and unnecessary. Instead, we aim to ensure the hot and warm data remain in cache safely. Such a relaxation on caching requirements enables us to achieve high efficiency with minimal loss in hit ratio. With more efficient usage of the system resources, we can achieve much better scalability compared to the current design.

We have developed a prototype based on Memcached, a widely used in-memory key-value cache in the industry. This prototype implements a compact metadata structure, called *Hitmap*, and an efficient caching scheme, called *Catalyst*, to manage the cache metadata. We also use Graphics Processing Unit (GPU) to offload the processing of metadata operations for hardware-assisted acceleration through parallel processing, which removes the interference to the main service operations on the host CPU. Our experimental results show that Catalyst can significantly improve the performance of in-memory key-value cache systems and increase the system throughput by a factor of up to 4.3 while still achieving comparable cache hit ratio to the traditional caching scheme.

The rest of the paper is organized as follows. Section 2 presents the background and motivations. Section 3 introduces the design. Section 4 presents the implementation and evaluation. Related work is discussed in Section 5. The last section concludes this paper.

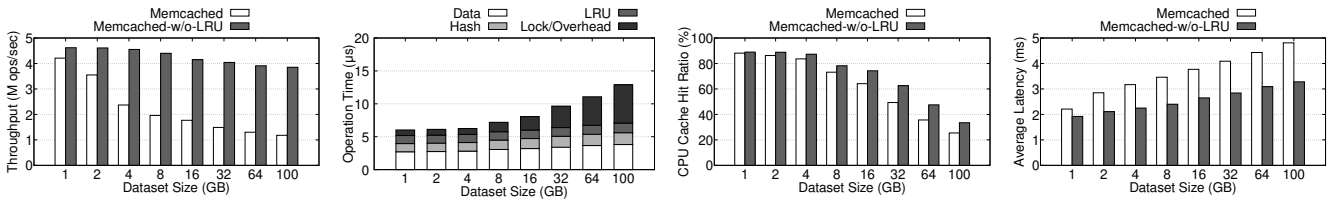
2 MOTIVATION

In a key-value cache system, a fundamental task is to identify the most likely-to-be-reused (hot) data and keep them in memory for fast access, while evicting the unlikely-to-be-reused (cold) data to make room for accommodating new data. With the rapid growth of key-value cache capacity, the traditional caching schemes are facing severe challenges in their scalability. In this section, we first use an experimental example to quantitatively illustrate the impact of the metadata operations and the associated overhead. Then we analyze and discuss the critical issues inherent in the current design.

2.1 An Example Case Study

Our example case runs on a W2600CR server equipped with two 8-core Intel Xeon E5-2690 2.90GHz processors and 128-GB DRAM memory. We use the popular YCSB benchmark [24] to generate multiple key-value datasets in different scales (1 GB to 100 GB). Despite the distinct dataset sizes, the workloads follow the same Zipfian distribution. We also ensure that all the datasets are contained completely in memory to avoid triggering replacement operations. For comparison, we turn off the LRU management in Memcached by bypassing all the pointer updates while still keeping the related metadata in memory. With a key-value size of 256 bytes in our experiments, each item requires 16 bytes for LRU pointers. Consequently, for a 100-GB dataset, approximately 6.25 GB of memory for metadata is allocated. We collect the throughput for each workload and compare the performance against the stock Memcached. We have some interesting observations.

As shown in Figure 1a, Memcached's performance is considerably boosted when it is not performing any LRU-related metadata operations. With a 100-GB dataset and LRU turned off, the throughput can reach 3.85 MOPS (Million Operations per Second) compared to 1.18 MOPS of the stock Memcached, which is an increase by a factor of 3.3. Although Memcached does not function as a key-value



(a) Throughput (b) Operation Time Breakdown (c) CPU Cache Hit Ratio (d) Average Latency

Figure 1: Overhead of Metadata Operations in Memcached for Caching

cache with LRU turned off, the performance gap between the two reveals an important insight to us—the current process of making caching decisions involves significant overhead.

To further study this unusual performance gap, we analyze the cost of each operation component in the query-handling process. In Figure 1b, we collect the average operation time for hash table operations, key-value data loading, LRU pointers updates, and the time that a thread spends on waiting for the lock when another thread is updating the same LRU list. We can see a small but steady increase in the operation time for loading the key-value data, which can be explained by the efficacy decay of the CPU cache [55]. Hashing operations also show slight increases due to the hash table expansions caused by the larger datasets. The cost associated with the LRU metadata operations, in contrast, shows a sharp increase with the dataset size. Although the time spent on manipulating the LRU pointers only slightly increases from 1.23 μs to 1.46 μs, the time on lock contention caused by the LRU structure manipulation increases by a factor of 6.4, from 0.9 μs to 5.8 μs. With the dataset size of 100 GB, the LRU-related metadata operations account for 56% of the total operation time, making it the dominant factor responsible for the observed performance drop. The cost breakdown reveals the root cause of the bottleneck and explains the observed throughput increase after disabling LRU in Memcached—the high overhead of metadata operations for caching.

In addition to the latency increase, the limited CPU resources are also under severe contention. In Figure 1c, we can see that the metadata operations negatively affect the CPU cache hit ratio, making the already scarce on-chip cache space even less accessible to service operations, which are responsible for handling client queries and retrieving the demanded data. After turning off the LRU, the key-value data retrieval operations experience less interference from the metadata operations. For example, with the 100-GB dataset size, the CPU cache hit ratio increases by 8 pp (Percentage Points). For in-memory key-value cache queries, which all happen in DRAM memory at nanosecond speed, such a substantial loss in CPU cache hit ratio has a significant performance impact.

2.2 Analysis and Discussions

The above-said experimental results demonstrate the strong negative impact of metadata operations in key-value cache management on both performance and scalability. Here we discuss several inherent critical issues in the current cache system design.

Issue #1: Metadata operations on the critical path. In order to differentiate hot and cold key-value items, conventional cache management needs to maintain certain metadata structures to track each item access. Such metadata operations are traditionally regarded as "trivial-cost" operations and thus conveniently embedded in the process of handling an incoming query. Memcached, for example, maintains a global LRU list for each slab class. Upon each access, it locks the entire list to ensure the structure's integrity. As the dataset size scales up, the number of items contained in each LRU list also increases, which naturally intensifies the lock contention due to frequent structure changes. As a result, the cost of metadata updates quickly increases, and worse, these non-essential, high-cost metadata operations are all synchronous, which puts the delays directly on the critical path of main service operations.

Issue #2: Serialized metadata operations. Traditional cache management relies on a linked list based structure. Memcached adopts an LRU-based algorithm, in which all items in the same slab class are organized in a doubly linked list. When an item is accessed, it becomes the Most Recently Used (MRU) item in the cache. To reflect this change, the newly accessed item needs to be unlinked from its original position and inserted to the head of the list. Due to the nature of the list operation, only one change can be made during each round of list updates. Thus, even with a number of such metadata operations pending in the system, the updates to the LRU list have to be performed one by one in a serial manner. Such a structure fundamentally prohibits performing frequent metadata operations in a parallelized manner.

Issue #3: Memory resource overhead. Traditional cache system design manages the key-value cache items in an overly fine-grained manner by attaching each individual data item with certain metadata. Such space overhead for metadata is non-trivial, especially considering that small items typically dominate key-value systems [13]. Assuming each 256-byte key-value item has 16 bytes of pointers for the LRU list, the metadata overhead would account for 6.25%. Although each item is managed individually, the metadata overhead in aggregate becomes significant for a large cache managing a massive number of small key-value items. For a 1-TB Memcached server with 256-byte items, the LRU metadata alone takes more than 64 GB of memory space, which could be used to accommodate more meaningful data in cache.

Issue #4: Computing resource overhead. In an in-memory key-value cache server, both processing client queries and updating the metadata of each accessed item consume CPU resources. Ironically, as the capacity increases, the system spends even more CPU cycles and processor cache space on maintaining metadata updates than handling client queries. In Memcached, each key-value item access involves at least five memory operations on the metadata for LRU. In contrast, loading the value data requires only one memory access. Even delete operations cannot be exempted from such a 5× operation overhead. These metadata operations not only compete for CPU cycles with the service operations, but also compete for the very limited on-chip cache space. In fact, the above-said five

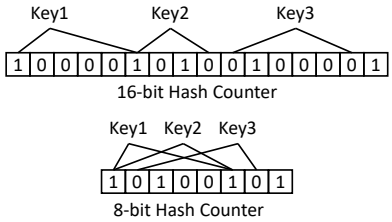


Figure 2: An Illustration of Hash Counter

memory accesses could evict more valuable key-value data from the CPU cache, causing reduced CPU cache hit ratio.

These issues severely impact the performance and scalability of in-memory key-value cache systems. We need to reconsider the cache management in the current system design. In this paper, we aim to design a new scheme to make asynchronous, parallelizable, space-efficient, and low-cost metadata operations.

3 DESIGN

To address the above-said challenges, we propose *Catalyst*, a **ca**che managemen**t** mech**a**nism for **l**arge in-memory ke**y**-value sy**st**ems. In this section, we first introduce a novel data structure for tracking a large amount of key-value items with low overhead, enabling the key-value caches to scale without performance degradation. We then present the procedure of updating the metadata structure and how to carry out a replacement algorithm to make eviction decisions. Finally, we discuss the key benefits of our design for solving issues in the current system design.

3.1 Organizing Key-value Metadata

A large key-value cache system often needs to manage billions of items with high performance. Attempting to accurately identify the "coldest" data for eviction is unrealistic and also unnecessary. We need a compact, parallelizable structure to closely approximate the LRU algorithm with minimal loss in hit ratio.

3.1.1 Hash Counters. We create an independent data structure, called *hash counter*, to estimate the relative temporal locality of key-value items in cache. Inspired by Bloom filter [18], hash counter is a hash-based bit array (see Figure 2) with N hash functions, using which a key is hashed and mapped to a set of N bits in the array. In our experiments, we find that using three hash bits (N=3) achieves good performance at a relatively low cost. Further increasing the number of hash bits for each key cannot bring substantial additional benefits. We use the widely adopted non-cryptographic hash functions, MurmurHash3 [5] and SpookyHash [8], to calculate the three hash values in a way similar to prior study [40]. The bit array is initialized to zero. Upon access, the set of N bits to which the key is mapped is set to one. Given a key, if its corresponding N bits are all set, we call this key is "marked", which indicates that the key has been accessed. Although the possibility of hash collision is very small, the hash counter may report false positive result (an unaccessed key is mistakenly reported as accessed), but it guarantees no false negative (an accessed key is reported as unaccessed).

Though simple, the hash counter structure brings several important advantages, making it particularly suitable for handling metadata operations. First, it gets rid of the lock-protected linked list structure, which fundamentally removes the lock contention problem and the strict requirement to perform metadata operations in serial. This not only immediately reduces the processing time of each individual metadata operation, and more importantly, it allows the key-value cache to parallelize the operations. Second, since we detach the metadata from the key-value data itself, this simple and independent data structure allows us to quickly access and update the metadata without reading the entire key-value data block, making it possible to asynchronize and batch metadata operations together, so that we can move these non-essential maintenance work off the critical path. Third, this structure also significantly reduces the memory overhead for metadata. Each item is mapped to only N bits, and each bit could be shared by multiple keys. Also, the size of the hash counter is adjustable. In contrast to the fixed metadata-to-data ratio with LRU, hash counter can flexibly adjust its size by mapping more or less keys to the hash bits at different precision. Finally, the independent bit array and the hash function based structure enable us to easily offload the metadata operations from the host CPU to a more suitable device, GPU. This not only removes the competition for the limited CPU resources, which now can be dedicated to service-handling operations, but also fully exploits the unique strength of the GPU hardware in handling massive parallel operations, such as hashing and bit array-based computing in our case.

The hash counter is a very efficient structure to manage the metadata of a large number of key-value items in cache. The size of the hash counter can affect the precision of verifying the presence of a key in the hash counter. Figure 2 illustrates a 8-bit and a 16-bit hash counter. Each block represents one bit. Each key is mapped to two unique bit entries of the hash counter. In this example, only Key1 has been accessed. Both hash counters correctly report the status of Key1 and Key3, while the smaller 8-bit hash counter gives a false positive result on Key2 due to hash collision. Note that hash counter does not report false negative results. Thus it guarantees that it is impossible to misclassify an accessed item as "unaccessed", meaning that hot and warm data would not be mistakenly evicted. We will study this effect in Section 4.3.

The hash counter can help us differentiate the relatively *hot* and *cold* items—a marked item in the hash counter is hot, whereas an unmarked item is cold. A rudimentary cache replacement mechanism can be implemented based on the hash counter. However, we can only make binary decisions, since the item is either hot or cold and it cannot further differentiate *hot* and *warm* items. We need a deeper access history information at a finer granularity to effectively approximate the LRU replacement.

3.1.2 The Hitmap Structure. We introduce a multi-level bit array structure, called *hitmap*, to differentiate the temperatures of the key-value items at a higher resolution. As illustrated in Figure 3, the hitmap integrates M levels of hash counters, each of which has a different size. Hitmap tracks the keys' access counts as follows.

Initially all the hash counters are reset to zeros. When a key is accessed, it is inserted (marked) in the hitmap from the lowest level to the highest level. We first check the level-1 hash counter. If any mapped bit is unset, meaning that the key has never been accessed, then all the mapped bits of the key are set to one to mark the access; if all the mapped bits are already set, meaning that the key has been accessed at least once, then we proceed to the next level and check

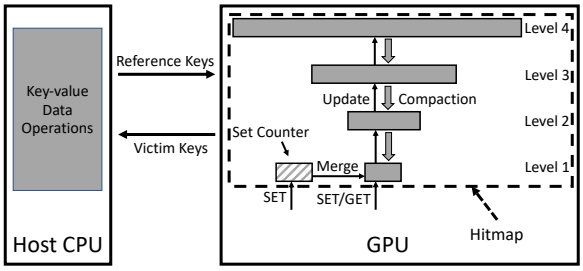


Figure 3: Hitmap and Overall Structure of Catalyst

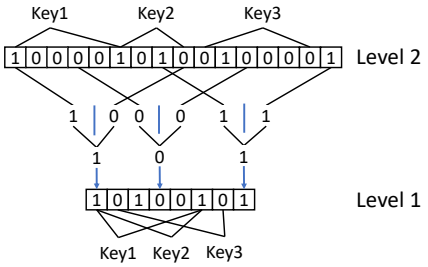


Figure 4: An illustration of Rolling Compaction

the mapped bits there in the same way. This process repeats until reaching the level where the key is not marked or the top level. If not marked, we set the bits and return.

Hitmap essentially records the access count (up to M) for the queried key. For example, in a four-level hitmap, if a key can be found in levels 1 and 2, its access count is two. If a key is tested positive in all four levels, we know that this key has been accessed at least four times. Only hot items could accumulate enough accesses to climb to the top level in the hitmap. The higher levels capture hotter items, and the colder items are at the lower levels. To some extent, this multi-level hitmap resembles a "heatmap" of accesses over the hash mapping space, allowing us to make more accurate eviction decisions to protect hot items while evicting the cold ones.

3.1.3 Hitmap Size. A naïve organization of hitmap is to make the hash counters at different levels the same size. In order to reduce the memory footprint of the hitmap structure, we organize the multi-level hitmap in an "inverted pyramid" shape, with the bottom-level hash counter being the smallest and the top level being the largest, as shown in Figure 3. In particular, the hash counter size at each level is twice of the level immediately below it. The 2:1 ratio is a deliberate choice, which will be explained in Section 3.1.4.

This compact metadata structure design uses much less memory than a hitmap with equal-sized hash counters, but it still offers comparable performance. This is for several reasons. First, the primary purpose of the lower-level hash counters is to filter out the cold items. Since the hash counter guarantees no false negatives, a small hash counter is adequate to find cold items, which are those with at least one "0" hash bit entry. Some cold items may have a slim chance of passing the test in a small hash counter because of the false positive effect, but they can still be filtered out when they move up to a larger, higher-resolution hash counter in the next level. Second, the level-1 hash counter in the hitmap receives the most accesses. Thus, a smaller hash counter can be loaded in GPU much faster due to its small size. The querying and updating are also faster due to fewer bit flips. Finally, the higher-level hash counters are mainly for differentiating the hot and warm items. A larger hash counter allows for a higher resolution, eventually translating into more accurate eviction decisions. With a stack of varying-sized hash counters, Catalyst can enjoy the best of both worlds, high accuracy and high efficiency. It allows us to achieve similar performance at a much lower cost compared to the traditional LRU. More detailed study can be found in Section 4.3.

3.1.4 Rolling Compaction. The *fill rate* (the percentage of "1" bits) can impact the efficacy of hitmap. Due to the nature of the hash-based bit array, multiple keys could be mapped to the same bit. Thus, once a bit entry is set, it can no longer be set back to zero. As the fill rate of a hash counter increases, the hot and cold items become indistinguishable, affecting our ability to make proper eviction decisions.

This aging effect could be mitigated by simply resetting the hash counters periodically. However, after a fresh reset, all the bits in the hash counter become zero, losing all previously recorded access history, which is unacceptable. We design a *rolling compaction* scheme to address this challenge. It works as follows.

As described previously, the multi-level hitmap structure records the access counts of key-value items. The presence of a key in the m-th level hash counter indicates that the key has been accessed for at least m times. In the rolling compaction process, we move each hash counter one level down and retire the level-1 hash counter. It essentially decrements every key's access count by one, making the keys originally being accessed only once eligible for eviction.

The challenge is how to compact a larger upper-level hash counter into a smaller lower-level hash counter without losing the access information. Since we cannot reverse-map a hash bit back to the original key, it is impossible to rerun the hash mapping process to set the bits one by one. Even if it was feasible, it would be too costly. We use a simple yet very effective method.

Recall that as described previously, we set the m-th level hash counter to be twice the size of the (m-1)-th level hash counter. This is a deliberate design to enable us to map the hash bit entries to the lower level without recalculating the hash values. Since we use the same set of hash functions at all levels, the generated hash values for mapping are identical. Thus, we can easily merge (`OR`) two bits into one and set the corresponding bit in the lower-level hash counter. As illustrated in Figure 4, we simply `OR` the first and second half of the original level-2 bit array into a new bit array of half size for level 1. Thus, the new level-1 hash counter is essentially a compact version of the original level-2 hash counter. It downsizes a higher-level hash counter to the lower level, which has a lower resolution but still retains most access information.

During each round of compaction, we first create a compacted version of the level-2 hash counter as described above to replace the current level-1 hash counter, and then we proceed to level 3 and repeat the same process until reaching the top level. After the compaction process propagates through all levels, the top-level hash counter is reset, while the lower levels still retain the access information rolled down from the upper levels. This process effectively preserves valuable access history and ensures non-disrupted operation, and it only takes a few simple bit array operations, which are highly efficient and easily parallelizable on GPU. This rolling compaction process is described in Algorithm 1.

Algorithm 1: Rolling Compaction

$L(m)$: Level-m hash counter;
$C_i(a)$: Hash counter value of Key(a) in level i;
T: The number of sampled keys in a batch;
S: Compaction trigger threshold;
M: The number of levels in the hitmap;
for *each batch of sampled keys* **do**
  $P \leftarrow 0$;
  for *each sampled key a* **do**
    if $C_1(a) = 0$ **then**
      $P++$;
    end
  end
  if $\frac{P}{T} < S$ **then**
    for *each level m from 2 to M* **do**
      $L(m-1) \leftarrow L(m)$;
    end
    $L(M) \leftarrow 0$;
  end
end

The appropriate timing of compaction is also critical. If compaction is triggered too early, we may lose too much access history and could mistakenly evict some hot items; If compaction is triggered too late, the fill rate of hitmap can become too high to differentiate hot and cold data. In Catalyst, we monitor the system status by sampling the memory content to determine the timing for compaction. The randomly selected key-value items can represent the overall system's status at anytime. We focus on the level-1 hash counter, since it has the highest fill rate of any level. By checking the sampled keys in level 1, we estimate the fill rate of level 1 based on the percentage of cold items found among the sampled keys. If most of the sampled keys are unmarked, it means that the fill rate is low and the hitmap can continue to function at high efficiency; otherwise, if the majority of the sampled keys are marked, it means that the current fill rate is too high and the level-1 hash counter is losing its efficacy, indicating that we should start rolling compaction to retire the aged hash counter. In this way, we create a connection between the system status and the fill rate, allowing us to fine-tune the hitmap based on system status. Once we acquire the status of the fill rate, we can set a threshold to trigger the rolling compaction. In our prototype, we start rolling compaction, if less than 5% of the sampled keys are unmarked. We find this setting works well in our experiments (see details in Section 4.3).

3.1.5 Protecting New Items. A side effect of the compaction process is that if compaction happens right after a key-value item is accommodated into the cache (`SET`), this item could be mistakenly identified as a cold item. Since rolling compaction retires the level-1 hash counter, the newly cached item would not have a chance to be reaccessed and could be evicted out of the cache prematurely. To prevent such a situation, we add a companion level-1 hash counter, called *set counter*, to the hitmap structure. The purpose is to record the recently cached key-value items since last compaction. The size of the set counter is equal to that of the level-1 hash counter. Upon a `SET` request, we update the bits in both the level-1 hash counter and the set counter. During compaction, we merge the set counter bits into the new level-1 hash counter and reset the set counter for the next round. This effectively sets the newly inserted keys twice, offering them a second chance and protecting them from being evicted without having a chance for reaccess.

3.1.6 Handling Zombie Data. A well-known limitation of LRU is on handling one-time-access data. Once being admitted into cache, these data becomes "zombie data" and stays in the cache for a long period of time until reaching the end of the LRU list. It negatively affects the caching performance, since these data occupies valuable cache space and causes unnecessary eviction of other more useful data. An added benefit of the set counter with rolling compaction is to clean such zombie data out. It essentially enforces a test period for each newly admitted data in cache—the zombie data would be flushed out of the cache after two compaction cycles. If a key is not found in any level of the hitmap or the set counter, it means that the key has not been accessed since being cached. These keys will be marked as eviction candidates. This allows us to proactively locate and evict the zombie data, minimizing its negative effect.

3.2 Metadata Operations in Catalyst

In Catalyst, the metadata operations, which handle the hitmap structure update and query, are asynchronized and separated from the main service operations, which handle client requests, such as `SET`, `GET`, and `DELETE`. We adopt GPU as the hardware accelerator to process metadata operations. It brings two benefits.

First, the hitmap structure is based on bit array and hash mapping, which makes metadata operations simple and parallelizable, and such operations are particularly suitable and highly efficient for running on GPU. Second, offloading metadata operations to GPU also physically separates the non-essential maintenance work from service operations, dedicating the limited CPU resources to the "real" work and removing the heavy interference.

In this section, we discuss three main operations in Catalyst, namely transferring data between devices, updating metadata in the hitmap, and making eviction decisions.

3.2.1 Transferring Access History. In Catalyst, a client request is handled by the main working threads. Each access should be passed to the GPU for updating the hitmap structure and preparing for the eviction decision later. A naïve approach is to send each accessed key to the GPU individually. This is clearly sub-optimal for three reasons. First, the system needs to pin a memory page (usually 4 KB) in memory for transferring data to the GPU via Direct Memory Access (DMA). However, the size of a single key is often too small compared to the page size. Second, the cache system needs to make a system call for each inter-device data transfer, adding more overhead to the CPU. Finally, GPU is a high-throughput computing device. Processing each key access individually is very inefficient and cannot exploit its strength in parallel processing.

To fully utilize the PCIe bandwidth and the GPU's computing power, Catalyst uses *reference pool* to collect a set of accessed keys to send to GPU in batch. To mitigate the overhead in memory transfer, Catalyst keeps the reference pool in a designated pinned memory block to ensure its availability. Each reference pool is a 4-KB page pinned in memory. So the memory block is always ready for transfer, which avoids the overhead of preprocessing.

Another optimization is to choose not to directly send the keys but the hash digests of the keys to the GPU. Since these hash digests

Algorithm 2: Hitmap Update

```
C_i(a): Hash counter value of Key(a) in level i;
C_set(a): Companion set counter value of Key(a);
M: The number of levels in the hitmap;
for each referenced key a do
    for each level i from 1 to M do
        if C_i(a) = 0 then
            C_i(a) ← 1;
            if i = 1 and SET then
                C_set(a) ← 1;
            end
            break;
        end
    end
end
```

are already used for hash mapping in the key-value cache, it would not incur extra overhead and it brings three benefits. First, the hash digest size (4 bytes) is typically smaller than the key size. Second, due to the smaller size, the reference pool can accommodate more keys, lowering the amount of data for inter-device transfer. Finally, the fixed size of hash digests allows us to avoid the alignment problem in the memory page, which simplifies the design.

3.2.2 Updating Hitmap Structure. Catalyst collects the access history of the keys on the CPU. For a `SET` request, the hash digest is logged into the reference pool after completing the insertion operations to the hash table and the cache space. For a `GET` request, similarly, the working thread first locates the target data and returns the value, then it adds the hash digest to the reference pool. If a key is accessed multiple times, we keep multiple hash digests in the reference pool to record its access history. For a `DELETE` request, the request returns after the removal of its index and data. As it is impossible to clear a hash bit once it is set, the metadata remains in the hitmap and will be flushed by the rolling compaction process.

The multi-level hitmap structure is maintained on GPU. Its size is determined by the size of the level-1 hash counter and the height of the hitmap. Assuming a four-level hitmap with a 256-MB level-1 hash counter, we need about 4-GB GPU memory in total. We will study the parameter setting in Section 4.3. It is also worth noting that as a general data structure, hitmap can be implemented on CPU, but it is more efficient running on highly parallel GPU.

In Catalyst, the keys are sent to the GPU in batches to update the hitmap. When the GPU receives a batch of referenced keys, it evenly distributes the jobs across all the Streaming Multiprocessors (SMs) to update the metadata in parallel. We first check if the key is marked in level 1. If it is unmarked, Catalyst inserts the key into the level-1 hash counter, and for a `SET` request, we also mark it in the companion set counter; if it is already marked, we move one level up to check the level-2 hash counter. This process repeats until finding a hash counter in which the key is unmarked or reaching the top level. If the key is unmarked, we mark it in the hash counter. This hitmap update process is shown in Algorithm 2.

3.2.3 Making Eviction Decisions. The hitmap structure captures the access history of key-value items and places them into different levels. The position of an object in the multi-level hitmap represents its recency and frequency combined—in the hitmap, an item moves one level up upon each access, and if not being accessed for a while, it moves down during compaction. The highest level where a key is tested positive represents the *temperature* of the key-value item. Put it simply, the higher the level is, the hotter the key is. If a key is tested negative at all levels in the hitmap, it means that this key-value item has not been accessed recently and has not been accessed often in the past, indicating that the key is among the coldest ones and thus is eligible for eviction.

We classify the key-value items into different *temperature zones*, according to their positions in the multiple levels of the hitmap. For example, with a four-level hitmap, the items are categorized into five temperature zones, from the *hottest* (Zone 4) to the *coldest* (Zone 0). If a key is found in the level-4 hash counter, it is placed in Zone 4; if a key is not found in any levels, it is placed in Zone 0. The different temperatures provide adequate locality information for us to approximate the LRU replacement at low cost. To achieve a similar effect of evicting the least recently used objects, we always evict the items whose keys are in Zone 0 first.

To make eviction decisions, Catalyst randomly samples a set of keys and sends them to GPU together with the batch of referenced keys. The number of sampled keys is set equal to that of the referenced keys, which is to ensure a sufficient number of candidate victims examined for eviction. Catalyst looks up each sampled key's position in the hitmap to determine its temperature zone. After all is done, the sampled keys and their temperature information are sent back to the host CPU, which then merges the sampled items that are colder than the existing ones in the *victim pool*. When space is needed, the coldest victim item is evicted from the pool. As the victim keys are sampled at the same rate as the key references, the more frequent the key references are, the more frequently the victim pool is refilled and updated. It ensures that the victim pool has sufficient items ready for eviction. Though rarely happens, in extreme cases when the victim pool is completely drained, inserting an item has to wait for the eviction process to complete, which is similar to other cache systems under such a situation.

3.3 Summary

Catalyst achieves its design goal with four important measures. (1) The design decouples the caching-related metadata operations from the main service operations, moving these non-essential maintenance work out of the critical path and making these metadata operations performed in an asynchronized way. (2) Catalyst abandons the lock-protected linked list based structure, removing the lock contention and the strict requirements for serial operations. The multi-level hitmap structure is based on bit array and hash mapping, which are parallelizable and very suitable for execution on GPU. (3) The hitmap structure provides a highly compact, summarized representation of the key-value items' access history, which allows us to approximate the principle of LRU replacement at a much lower cost. (4) Offloading the metadata operations to GPU not only removes the interference to main service operations and reduces the contention on the limited CPU resource, but it also exploits the unique strength of GPU in handling parallel tasks. With all these optimizations together, Catalyst fundamentally addresses the scalability challenge and allows a large-capacity in-memory key-value cache to handle a massive amount of small key-value items in a very efficient manner.

4 IMPLEMENTATION AND EVALUATION

4.1 Implementation

To evaluate the proposed scheme, we have developed a prototype of Catalyst based on Memcached [4], an in-memory key-value cache system widely deployed in industry. It adopts a multi-threaded design to handle concurrent requests, making it an appropriate platform to demonstrate the performance of our design.

We have modified the metadata management and operations related to cache replacement in Memcached, while keeping the workflow of regular query operations unchanged. We added about 2,600 lines of CUDA C code to realize the functions presented in Section 3. As a versatile programming model, CUDA supports different memory allocation and data transfer modes. We use *explicit memory allocation* (e.g., with `cudaMalloc`) and actively manage the data transfer between devices (e.g., with `cudaMemcpy`) in our prototype, which allows us to explicitly store all metadata structures in GPU memory for best performance. The GPU also supports *unified memory* (e.g., with `cudaMallocManaged`), which allows the GPU to share the host memory space with some performance penalty. Our prototype can also be adapted to run with unified memory for obtaining more memory space to host the metadata.

We have also implemented a lightweight intermediate library for efficiently operating the GPU device. It has three main functions: (1) Sending jobs. Initiate data transfer to send the access information collected in reference pool to the GPU device memory. (2) Launching kernels. After receiving the jobs from the key-value cache, launch GPU kernels to process and update the metadata of the keys in the current batch. (3) Receiving results. After each batch finishes updates, transfer the victim list back to the host CPU.

4.2 Experimental Setup

Our key-value cache system runs on a W2600CR server equipped with two 8-core Intel Xeon E5-2690 2.90GHz processors, 128-GB DRAM memory, and an NVIDIA GTX 1050 Ti GPU with 4-GB GDDR5 memory via PCIe 3.0 x16. This entry-level GPU costs about $80-100 on the second-hand market, incurring minimal extra cost. A 120-GB Intel 540 flash SSD is used for the experiments with Extstore [1]. We use two Lenovo TS440 ThinkServers as clients. Each server has a 4-core Intel Xeon E3-1245 3.4 GHz processor, 16-GB DRAM memory, and a 7,200 RPM 1-TB Seagate disk drive. Each server simulates 64 concurrent clients to send queries. Our backend database is a MongoDB 4.2 database running on a 4-TB 7,200 RPM Seagate hard drive, hosted on a Dell T620 server with a 6-core Intel Xeon E5-2630 2.3 GHz processor and 32-GB memory. For sufficient network bandwidth, we aggregate four 10-Gbps Ethernet ports together on the key-value server and two 10-Gbps Ethernet ports on each client machine and the database server. We use Ubuntu 20.04 with Linux kernel 5.4 and Ext4 file system. Our CUDA C code is compiled using *nvcc* with CUDA 9.2.

We use the Yahoo! Cloud Serving Benchmark (YCSB) [24] to generate workloads with four access patterns, *Zipfian*, *Hotspot*, *Latest*, and *Uniform*, to emulate different workloads [19, 26] and collect the traces. The key-value data content is not of interest in this study and is thus filled with random data. Our synthesized workloads follow the size distribution of dataset *APP*, *SYS*, and *ETC* reported in a study of Facebook workloads [13]. The majority of the items in *APP* are around 300 bytes, following a generalized extreme value distribution [14]. *SYS* follows the same distribution but comprises more larger items. Most are around 600 bytes. *ETC* follows a generalized Pareto distribution [11], and its value sizes are more evenly distributed from 14 bytes to around 1 KB.

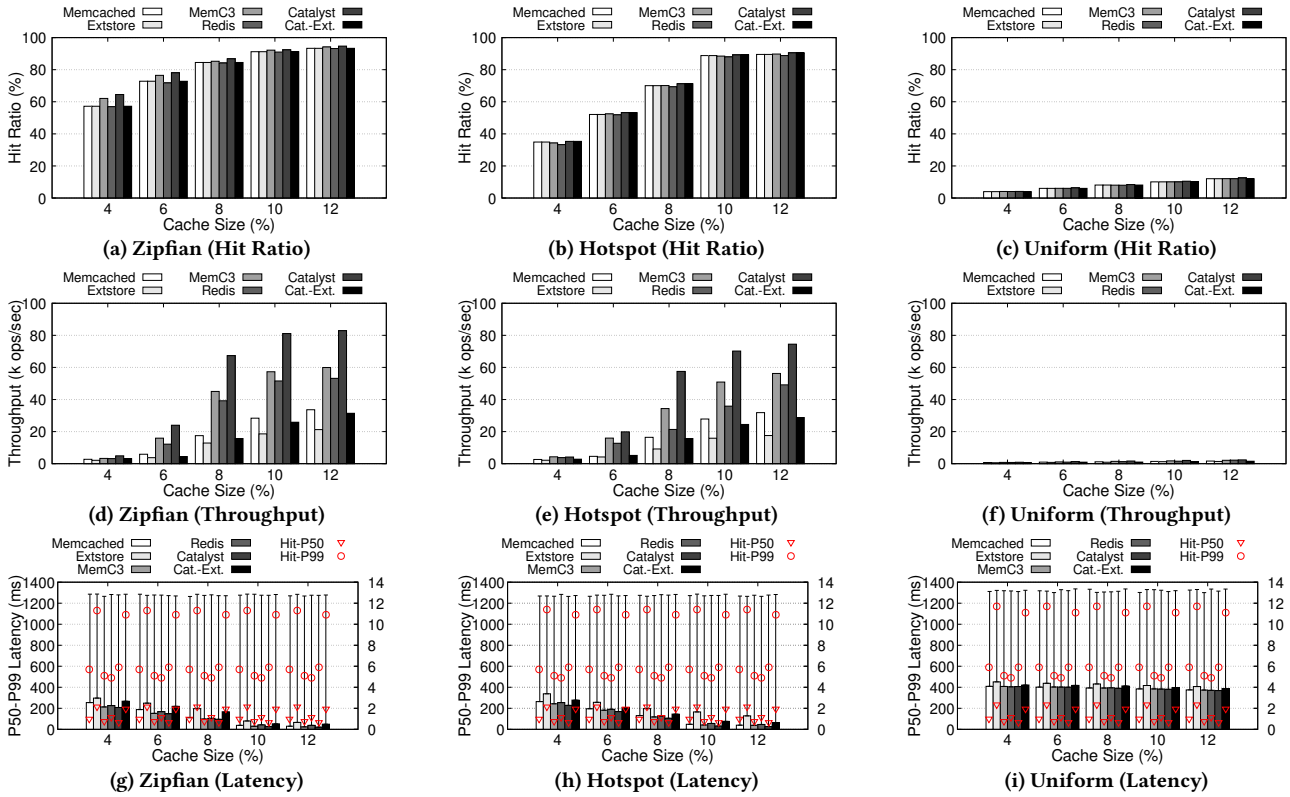
We use a homegrown tool, called *keystone*, to replay the workload traces against the key-value cache system for repeatable tests. This tool allows us to precisely repeat a workload with a specified number of clients and test the system performance with each of the above-mentioned data sets. We collect the system timestamp in nanoseconds with the `CLOCK_MONOTONIC_RAW` clock [2] and calculate the operation time on the key-value cache server. We use the tool `perf` [3] in Linux to analyze the CPU cache hit ratio.

4.3 Evaluation

4.3.1 Overall Performance. In this set of experiments, we evaluate the overall performance of the in-memory key-value cache by pairing it with a backend database server with 1 TB of key-value data, allowing the system to test the effectiveness of its cache space management scheme. We configure Catalyst with the best overall settings. The hitmap is configured with four levels of hash counters, whose sizes are 256 MB, 512 MB, 1 GB, and 2 GB, from level 1 to level 4, respectively. A 256-MB set counter is employed to keep track of newly inserted keys through `SET` requests.

We use the *APP* dataset with *Zipfian* and *Hotspot* workloads to simulate typical use cases of key-value caches. We also run the *Uniform* workload to represent the worst-case scenario in which the workload access pattern exhibits minimal locality. In our evaluation, we compare Catalyst against three baseline cases, namely Memcached, MemC3 [29], and Redis [7]. In addition to the three in-memory cache systems, we also set up a flash-based cache, Extstore [1], which extends Memcached with an external flash SSD, to demonstrate the versatility of the proposed scheme and its compatibility with different storage configurations.

Our experiments show that Catalyst is very efficient and closely approximates LRU with low overhead. For given memory space, Catalyst is able to cache more key-value items than Memcached for better performance. As shown in Figure 5a-5c, Catalyst shows up to 7.3 pp higher hit ratio than Memcached with Zipfian workload. Even in cases with similar hit ratios, Catalyst is able to process requests at a much higher throughput (Figure 5d-5f). Thanks to the highly efficient metadata management, Catalyst sees a 23%–333% throughput increase. Figure 5g-5i show the latency results. Compared to Memcached, Catalyst reduces the 50th percentile overall latency (represented by the thick bars) by up to 20.6%. The 99th percentile overall latency (represented by the line bars) does not show notable differences, as it is primarily determined by the slow backend database rather than the caching mechanism. We also show the 50th and 99th percentile latencies of the "hit-in-cache" queries, illustrated as triangles and circles in Figure 5g-5i, respectively. The corresponding values can be read from the right axis. In the figures, we can see that the two Extstore-based schemes exhibit higher latencies than the others, due to the use of flash SSD as cache extension. Compared to Memcached, Catalyst shows up to 35% lower 50th percentile latency and slightly higher (up to 3.5%) 99th percentile latency, which shows the benefit of removing interference of metadata operations and the small overhead in handling

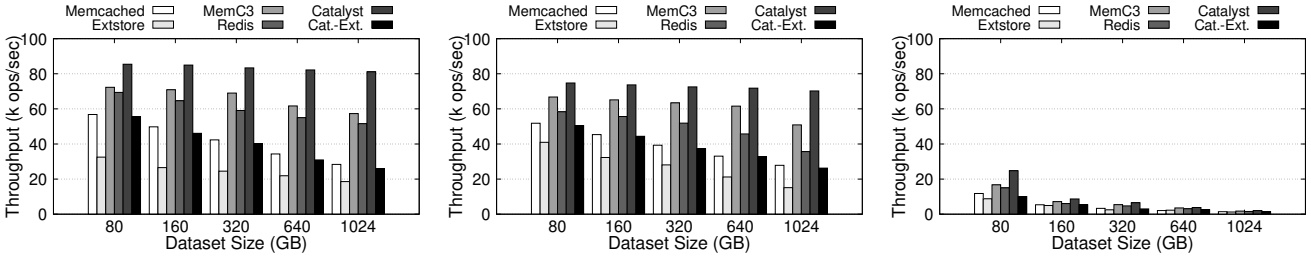


(a) Zipfian (Hit Ratio) (b) Hotspot (Hit Ratio) (c) Uniform (Hit Ratio)

(d) Zipfian (Throughput) (e) Hotspot (Throughput) (f) Uniform (Throughput)

(g) Zipfian (Latency) (h) Hotspot (Latency) (i) Uniform (Latency)

Figure 5: Overall Hit Ratio, Throughput, and Latency Performance of Catalyst



(a) Zipfian (b) Hotspot (c) Uniform

Figure 6: Scalability of Catalyst with Various Dataset Sizes (10% Cache Ratio)

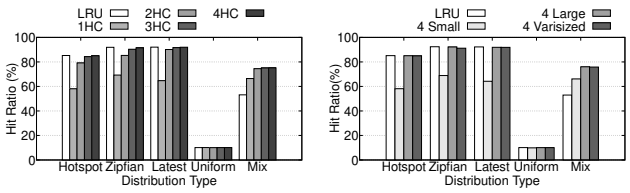
cache hits. With the *Uniform* workload, Catalyst's performance aligns with Memcached, showing that it does not bring negative impact on performance, even in such a worst-case scenario.

MemC3 uses the CLOCK replacement algorithm to reduce its metadata footprint, which makes it, like Catalyst, able to keep more data in memory. However, MemC3 exhibits slightly lower cache hit ratios compared to Catalyst across all tests. For a similar purpose, Redis also attempts to reduce the memory usage of metadata by associating a 3-byte LRU clock with each key-value item and relying on random sampling to identify cold items for eviction. However, due to its less effective caching mechanism, Redis cannot achieve the same hit ratio as Catalyst. Besides the lower hit ratios, both MemC3 and Redis involve metadata operations on the critical path. For instance, MemC3 needs to update the circular list and clock hand upon each access, while Redis requires to acquire timestamps and update each item's LRU clock. In contrast, Catalyst processes the metadata asynchronously, eliminating the overhead from the critical path. In our tests, Catalyst outperforms MemC3 and Redis by up to 67.4% and 169.4%, respectively, showcasing its significantly better performance.

As a versatile scheme, Catalyst can be adapted into various key-value systems. We have implemented an enhanced version of Extstore by augmenting it with Catalyst, denoted as *Cat.-Ext.* In this configuration, we allocate 50% of the key-value items to be stored on flash SSD as an alternative to the all-in-memory cache systems. As shown in Figure 5, Cat.-Ext. consistently outperforms Extstore and exhibits throughput improvement of up to 71.4%. It is worth noting that the performance improvement brought by Catalyst on Extstore is less significant compared to that on Memcached. This can be attributed to the substantial speed disparity between DRAM

Table 1: Throughput (KOPS) with Small Datasets (Zipfian)

Dataset (GB)	1	2	4	8	16	32
Memcached	139	137	132	124	112	109
Catalyst	231	230	229	227	221	216



(a) Number of Hash Counters **(b) Hash Counter Size**

Figure 7: Hitmap Configuration

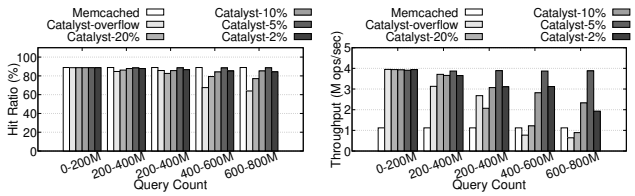
memory and flash SSD. While Cat.-Ext. offers superior efficiency in metadata management, the overall system performance is still constrained by the speed of the underlying flash SSD.

4.3.2 Scalability Study. Figure 6 presents a comparison of the scalability of the six cache systems. The dataset sizes range from 80 GB to 1 TB, with a key-value cache configured at a fixed 10% cache ratio, varying from 8 GB to 100 GB. Although the hit ratio largely remains constant as the system scales up due to the fixed cache ratio, the overall throughput shows significant changes.

In our tests, Memcached experiences a 50% drop in throughput, while Catalyst maintains more stable performance even with a scale-up factor of 12.5×. Notably, Catalyst's throughput only decreases by 5.1% for the *Zipfian* workload and 6.1% for the *Hotspot* workload, showcasing its better scalability, in contrast to MemC3's 23.8% and Redis's 38.9% drop in throughput. It clearly demonstrates that Catalyst scales well with growing data volumes. Table 1 compares Memcached and Catalyst with small datasets. It shows that under less scalability pressure, Memcached performs better with smaller datasets, however, it still significantly lags behind Catalyst in throughput. In the next sections, we will study each component individually. To fully exercise the cache server, we run our experiments with the cache server only and use a cache size of 100 GB and the *Zipfian* workload, unless otherwise specified.

4.3.3 Hitmap Configuration. The configuration of hitmap can impact its efficacy. This set of tests studies the effect of the number and size of hash counters in hitmap. Figure 7a shows the hit ratios with different hitmap heights. With only one or two levels of hash counters, the hit ratio is noticeably lower than LRU. When using three levels of hash counters, Catalyst's cache hit ratio quickly improves, with only 0.4-1.6 pp lower than LRU across all workloads. With four levels of hash counters, Catalyst can nearly match the hit ratio of LRU under the *Hotspot* and *Latest* workloads and is only 0.4 pp lower than that of LRU with the *Zipfian* workload.

We have also tested the key-value cache server with a mixed workload to better represent real-world scenarios. We mix the *Zipfian* workload with one-time write requests at a 1:1 ratio, denoted as *Mix*. Our evaluation results reveal that with a dedicated set counter and rolling compaction, Catalyst can distinguish the one-time requests and actively evict the zombie data, which protects the hot items. In contrast, LRU inserts the one-time accessed items to the head of the LRU list upon access and waits for them to be flushed



(a) Hit Ratio **(b) Throughput**

Figure 8: Compaction Configuration

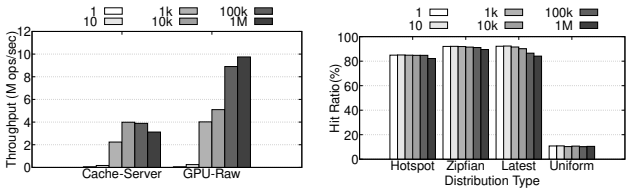
after staying through the long list. The poor efficiency of LRU when dealing with such a workload is reflected in the hit ratio, which is up to 22 pp lower than Catalyst.

Figure 7b shows the hit ratio of a four-level hitmap with different sizes. When using four equal-sized 256-MB hash counters (a narrow rectangle shape), we cannot match the hit ratio of LRU. This is due to the high fill rate of the small hash counters, which fail to distinguish hot items from the warm ones. Having four such small hash counters does not improve either. In contrast, when using four equal-sized 2-GB counters (a wide rectangle shape), we can achieve a high resolution (false positive rate lower than 0.0001%) on all four levels. This improves the hit ratio, making it similar to LRU. However, the performance comes at the cost of GPU memory. Catalyst uses hash counters that vary in size at different levels. The first level is set to 256 MB, and each upper layer doubles the size of the one below it (an inverted pyramid shape). This allows us to enjoy the best of both approaches. The smaller hash counters at the bottom can filter out cold items even with a higher false positive rate, while the larger hash counters at the top can distinguish hot and warm items with a higher resolution. We achieve a hit ratio nearly identical to that of using four large hash counters with only 46.9% of the memory usage (3.75 GB vs. 8 GB).

4.3.4 Hitmap Compaction. Due to the nature of the hash counter design, we expect to see the efficacy decreases as the fill rate increases. To maintain the effectiveness of the hitmap, we need to periodically perform hitmap compaction to retire the aged hash counters. In Figure 8, we show how different compaction configurations affect the cache performance. We vary the compaction threshold setting from 2% to 20% (the percentage of the sampled keys that are unmarked in the level-1 counter). We have also tested the case with compaction disabled, denoted as *Catalyst-overflow*. We can see that when allowed to overflow, the hit ratio decreases as the query count increases, which clearly proves that compaction helps maintain the efficacy of hitmap over time.

Our experimental results show that compaction at 5% yields the best hit ratio results. A more aggressive compaction trigger (20%) may evict warm data prematurely, while a delayed compaction (2%) may lead to lack of available cache space. In Figure 8a, with the configuration at 5%, Catalyst is able to maintain a stable hit ratio of 88.5%-88.7%, nearly identical to the 88.9% of LRU. Other settings show more fluctuations in hit ratio. The 5% compaction trigger also gives the best throughput. In Figure 8b, the throughput with compaction at 5% eventually is 67%–336% higher than other compaction settings, up to 506% higher than the overflow case, and 246% higher than the stock Memcached.

4.3.5 Batch Size. GPU is known to have high computation bandwidth and supports a high degree of parallelism. We test the cache



(a) Throughput **(b) Hit Ratio**

Figure 9: Data Transfer Batch Size

server performance with different reference pool transfer batch sizes, ranging from one to one million keys per batch.

As shown in Figure 9a, the overall throughput is very low when using an excessively small batch size. For example, when transferring a single key at a time, the key-value cache server can only process requests at a throughput of 0.05 MOPS, and 0.17 MOPS with transfer batch size of 10 keys. This is because the cache server needs to be constantly waiting for the GPU to process metadata in many small batches to reclaim enough space. The whole system is in effect stalled due to the low throughput on the GPU.

In addition, using a very small batch size also leads to underutilized PCIe bandwidth without reducing the batch transfer time much. In fact, transferring a batch of 1-10k keys demands almost the same amount of time, which is mostly for making system calls, initiating transfer, and launching kernels, etc. Our tests show that on average, it takes the system 170 μs to transfer a batch of 10k keys to the GPU and about 140 μs to process the metadata in hitmap.

As the batch size increases, the overall throughput improves. After reaching 10k batch size, the raw GPU throughput keeps increasing, but the cache server experiences a slowdown with further increased batch size. In particular, when the batch size reaches 1 million, the GPU can update 9.75 million keys per second, but overall the cache system sees a throughput of only 3.12 MOPS.

To study the root cause of this unusual phenomenon, we collect the cache server hit ratio in Figure 9b. We see a clear trend in our evaluations—a larger batch size results in a lower hit ratio. For example, the hit ratio of 1-million batch size is up to 7.5 pp lower than the 1k batch size. This is due to the lag between the time when a query is served and when its metadata is updated. Ideally, we desire to have such lag as small as possible, so that the hitmap can timely reflect the changes. Although a larger batch size can better exploit the computing power of GPU, it makes the cache less responsive. As a result, we see the disparity between the throughput observed on the GPU and on the cache server.

4.3.6 System Overhead. Catalyst also aims to achieve low overhead on system resources. Benefiting from its compact and flexible metadata structure, Catalyst is much more efficient in memory usage than Memcached. As shown previously (Figure 7a), Catalyst can achieve a comparable hit ratio to LRU with a three- or four-level hitmap. In Figure 10a, we compare their memory usages for metadata. When managing a 100-GB cache with three-level hitmap, Catalyst's metadata footprint is only 37.4%, 36.5%, and 33.2% of Memcached's usage with dataset APP, SYS, and ETC, respectively; With a four-level hitmap, Catalyst matches the LRU hit ratio while still saving at least 25% of memory for metadata.

Catalyst is also highly efficient on operations. Figure 10b shows the overhead breakdown in details. In Catalyst, the metadata operations have a fixed overhead per request, around 0.4 μs, regardless of the dataset size differences. The stable operation cost shows high scalability of our design. As a result, when serving a 100-GB dataset in the cache server, the system spends about 6.8% of the total time on Catalyst-related operations, which is substantially lower than the 56% overhead of LRU in Memcached (see Figure 1b).

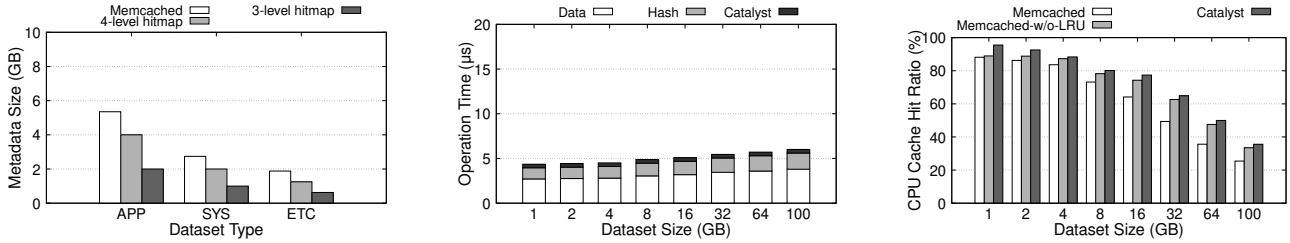
4.3.7 Effect on CPU Cache. CPU cache is an important resource. LRU in Memcached impairs the CPU cache hit ratio in two aspects. First, when a key-value item is loaded, the LRU pointers are also loaded and take extra space. Second, when the LRU list is updated, the related pointers also need to be loaded into the CPU cache to complete the list change. Both can cause interference to other operations that are in need of the limited CPU cache space.

In this test, we study how metadata management impacts the CPU cache hit ratio and its effect on the cache server's throughput. Figure 10c shows the CPU cache hit ratio of the stock Memcached, Memcached without LRU (same as in Section 2), and Catalyst. In the figure, we can see that Catalyst shows up to 15.5 pp higher in CPU cache hit ratio compared to the stock Memcached. The modified Memcached without LRU operations sits between the two, as it still has the extra spatial overhead. It is interesting to find that Catalyst performs even better than Memcached without LRU. As CPU cache is much faster than DRAM memory, the requests can be served at a higher rate with more efficient usage of the on-chip cache space. In particular, with a 64-GB dataset, the throughput of Catalyst is 5.12% higher than the modified Memcached.

4.3.8 GPU Hardware Resources. Catalyst demands GPU hardware resources for processing metadata to make eviction decisions. In this section, we study the impact of memory and computing power of GPU on the performance of Catalyst.

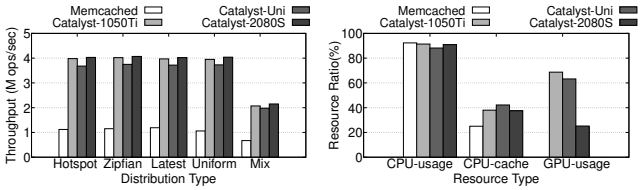
Unified memory. CUDA provides support for *unified memory*. By managing host memory and device memory in a single space, we can exceed the limit of physical memory capacity on GPU by using available system memory on the host, which could be beneficial when the GPU memory alone is insufficient for accommodating the entire hitmap structure. In this experiment, we evaluate Catalyst's performance with unified memory as an alternative to explicit memory allocation in the scenarios in which the system relies partially on the host memory to store the metadata. Figure 11a illustrates Catalyst's unified memory configuration, denoted as *Catalyst-Uni*. The results show that when using unified memory, we can observe a decrease of up to 7.5% in throughput compared to fully storing all metadata in GPU memory, but Catalyst still significantly outperforms Memcached.

Computing power. Catalyst demonstrates its ability to effectively manage the key-value cache metadata with a low-cost, entry-level GTX 1050Ti GPU, an "obsolete" device by today's standard. To evaluate the effect of using a more powerful GPU, we replace the GTX 1050Ti with an RTX 2080 Super GPU. Interestingly, although the number of CUDA cores increases from 768 to 3072, it does not bring notable performance difference in Figure 11b. The throughput difference between the two devices is less than 2.2%. Although the average GPU usage of the RTX 2080 Super is reduced to 25.2% due to more available cores, the computing resources on the GTX 1050 Ti are not fully utilized either (only 68.7% with explicit memory allocation, and 63.2% with unified memory). These results indicate that Catalyst does not require an expensive, high-end GPU. A low-cost



(a) Memory Overhead (b) Operation Overhead (c) CPU Cache Hit Ratio

Figure 10: System Overhead Comparisons



(a) Throughput (b) Resource Usage

Figure 11: GPU Hardware Resources

GPU, such as GTX 1050 Ti (only $80-100 USD on the second-hand market), works sufficiently well for processing metadata, and it adds minimally to the overall system cost, which makes it a cost-efficient solution in practice.

5 RELATED WORK

In recent years, key-value systems have been extensively studied in both academia and industry [4, 7, 10, 13, 22, 23, 25, 27, 28, 43, 44, 48, 51, 52, 60, 63]. In this section, we discuss the most related works.

The scalability concerns of large-capacity key-value stores have motivated many research studies [12, 16, 30, 31, 36, 50, 54, 55, 58, 59]. To reduce memory overhead, Wu et al. apply data compression to Memcached to virtually expand the memory capacity for storing more key-value data [59]. Wang et al. make use of page coloring to improve CPU cache efficiency and enhance the scalability of Memcached and Redis by reorganizing key-value items into dedicated memory partitions [55]. Ghigoff et al. apply pre-stack processing to intercept key-value requests from the NIC to overcome the limitation of Linux's network stack [30]. Unlike the above works, we focus on an important bottleneck that has not received its deserved attention, the scalability problem caused by metadata operations in in-memory key-value cache systems.

There are also many research works focusing on optimizing the caching algorithms for key-value stores [15, 17, 21, 29, 37, 41, 42, 49, 56, 57, 61]. For example, Fan et al. replace the caching algorithm in Memcached with a CLOCK-based algorithm to achieve a smaller memory footprint [29]. They also free certain operations (e.g., `SET` and `GET`) from lock contention by maintaining version counters. Wu et al. propose an efficiency-centric variant of LRU, the E-LRU, which divides the memory space into three components to store different data [57]. Hyperbolic Caching introduces the notion of cost classes for managing the cached objects [17]. The algorithm requires extra metadata attached to each object to record cost and expiration time. Beckmann et al. introduce the least hit density (LHD) algorithm for key-value caches, which makes eviction decisions based on an object's age and lifetime [15]. Our rationale in this work is entirely different from these prior studies. Instead of only focusing on the algorithm's hit ratio, we study the severe overhead involved in the process of making caching decisions and find solutions to address the incurred scalability problem. Furthermore, rather than tailoring the existing data structures, our scheme proposes a novel metadata structure to overcome the critical issues inherent in the current design for cache management.

Some prior works seek to increase the performance of key-value stores by adding additional hardware to the system and accelerating certain operations [32–35, 38, 39, 45, 46, 53, 62–64]. For example, Mega-KV maintains the entire hash indexing structure in GPU [63]. With its high computing bandwidth, hashing operations on GPU can be completed much faster than on CPU. Due to the nature of the key-value store's workflow, indexing needs to be calculated before the data can be located. The tradeoff between hashing throughput and latency must be carefully balanced when offloading the indexing request to GPU. In this work, we offload the metadata-related maintenance work to GPU. Since such operations can be performed asynchronously, our approach is insensitive to the latency issue caused by using GPU, which allows us to exploit the computational advantages without the need to sacrifice performance. Mao et al. implement a Memcached accelerator using FPGA, in which a portion of the key-value data is stored on a dedicated SRAM cache [46]. Such an approach requires a redesign of the existing application and it is also deeply integrated with specific hardware. Our approach optimizes caching-related metadata management and operations, which can run on a low-cost, general-purpose GPU, making it a cost-efficient solution in practice.

6 CONCLUSION

In-memory key-value cache systems are essential in today's Internet services. However, the cache management in the current system designs is unscalable due to the sharp increase of overhead associated with intensive metadata operations. In this paper, we present a scheme, called Catalyst, to efficiently manage the metadata of key-value items with minimal overhead. Our experimental results show that Catalyst can significantly improve the performance of in-memory key-value cache systems at a low cost.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd for their constructive feedback and insightful comments. This work was partially supported by the National Science Foundation under Grants CCF-1910958 and CCF-2210755.

REFERENCES

[1] 2023. Extstore. https://github.com/memcached/memcached/wiki/Extstore
[2] 2023. Linux Hardware Clock. https://man7.org/linux/man-pages/man3/clock_gettime.3.html
[3] 2023. Linux Perf. https://en.wikipedia.org/wiki/Perf_(Linux)
[4] 2023. Memcached. https://memcached.org
[5] 2023. MurmurHash3. https://en.wikipedia.org/wiki/MurmurHash
[6] 2023. Random-access Memory. https://en.wikipedia.org/wiki/Random-access_memory.
[7] 2023. Redis. https://redis.io
[8] 2023. Spooky Hash. http://www.burtleburtle.net/bob/hash/spooky.html
[9] 2023. Synchronous Dynamic Random-access Memory (SDRAM). https://en.wikipedia.org/wiki/Synchronous_dynamic_random-access_memory.
[10] David G Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. 2009. FAWN: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*. 1–14.
[11] Barry C Arnold. 2008. Pareto and generalized Pareto distributions. *Modeling income distributions and Lorenz curves* (2008), 119–145.
[12] Nikolas Askitis and Ranjan Sinha. 2010. Engineering scalable, cache and space efficient tries for strings. *The VLDB Journal* 19, 5 (2010), 633–660.
[13] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '12)*. 53–64.
[14] Turan G Bali. 2003. The generalized extreme value distribution. *Economics letters* 79, 3 (2003), 423–427.
[15] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. 2018. LHD: Improving cache hit rate by maximizing hit density. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18)*. 389–403.
[16] Lawrence Benson, Hendrik Makait, and Tilmann Rabl. 2021. Viper: An efficient hybrid PMem-DRAM key-value store. *Proceedings of the VLDB Endowment* 14, 9 (2021), 1544–1556.
[17] Aaron Blankstein, Siddhartha Sen, and Michael J. Freedman. 2017. Hyperbolic caching: Flexible caching for web applications. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC '17)*. 499–511.
[18] Burton H Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.
[19] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. 1999. Web caching and Zipf-like distributions: Evidence and implications. In *Proceedings of IEEE Conference on Computer Communications*. 126–134.
[20] Eric Burgener and John Rydning. 2022. High data growth and modern applications drive new storage requirements in digitally transformed enterprises. *IDC White Paper* (2022).
[21] Daniel Byrne, Nilufer Onder, and Zhenlin Wang. 2018. mPart: Miss-ratio curve guided partitioning in key-value stores. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on Memory Management (ISMM '18)*. 84–95.
[22] Jiqiang Chen, Liang Chen, Sheng Wang, Guoyun Zhu, Yuanyuan Sun, Huan Liu, and Feifei Li. 2020. HotRing: A hotspot-aware in-memory key-value store. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST '20)*. 239–252.
[23] Asaf Cidon, Daniel Rushton, Stephen M. Rumble, and Ryan Stutsman. 2017. Memshare: A dynamic multi-tenant key-value cache. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC '17)*. 321–334.
[24] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*. 143–154.
[25] Lixiao Cui, Kewen He, Yusen Li, Peng Li, Jiachen Zhang, Gang Wang, and Xiaoguang Liu. 2021. SwapKV: A hotness aware in-memory key-value store for hybrid memory systems. *IEEE Transactions on Knowledge and Data Engineering* 35, 1 (2021), 917–930.
[26] Carlos R Cunha, Azer Bestavros, and Mark E Crovella. 1995. *Characteristics of WWW client-based traces*. Technical Report. Boston University Computer Science Department.
[27] Biplob Debnath, Sudipta Sengupta, and Jin Li. 2010. FlashStore: High throughput persistent key-value store. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 1414–1425.
[28] Biplob Debnath, Sudipta Sengupta, and Jin Li. 2011. 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)*. 25–36.
[29] Bin Fan, David G Andersen, and Michael Kaminsky. 2013. 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)*. 371–384.
[30] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. 2021. BMC: Accelerating Memcached using safe in-kernel caching and pre-stack processing. In *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI '21)*. 487–501.
[31] Steffen Heinz, Justin Zobel, and Hugh E Williams. 2002. Burst tries: A fast, efficient data structure for string keys. *ACM Transactions on Information Systems (TOIS)* 20, 2 (2002), 192–223.
[32] Tayler H Hetherington, Mike O'Connor, and Tor M Aamodt. 2015. MemcachedGPU: Scaling-up scale-out key-value stores. In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC '15)*. 43–57.
[33] Tayler H Hetherington, Timothy G Rogers, Lisa Hsu, Mike O'Connor, and Tor M Aamodt. 2012. Characterizing and evaluating a key-value store application on heterogeneous CPU-GPU systems. In *Proceedings of the 2012 IEEE International Symposium on Performance Analysis of Systems Software*. IEEE, 88–98.
[34] Konstantinos Iliakis, Konstantina Koliogeorgi, Antonios Litke, Theodora Varvarigou, and Dimitrios Soudris. 2022. GPU accelerated blockchain over key-value database transactions. *IET Blockchain* 2, 1 (2022), 1–12.
[35] Zsolt István, David Sidler, and Gustavo Alonso. 2017. Caribou: Intelligent distributed storage. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1202–1213.
[36] Yichen Jia, Zili Shao, and Feng Chen. 2018. SlimCache: Exploiting data compression opportunities in flash-based key-value caching. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '18)*. 209–222.
[37] Hai Jin, Zhiwei Li, Haikun Liu, Xiaofei Liao, and Yu Zhang. 2019. Hotspot-aware hybrid memory management for in-memory key-value stores. *IEEE Transactions on Parallel and Distributed Systems* 31, 4 (2019), 779–792.
[38] Min-Gyo Jung, Chang-Gyu Lee, Donggyu Park, Sungyong Park, Jungki Noh, Woosuk Chung, Kyoung Park, and Youngjae Kim. 2021. GPUKV: An integrated framework with KVSSD and GPU through P2P communication support. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing*. 1156–1164.
[39] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D Nguyen, Tim Kaldewey, Victor W Lee, Scott A Brandt, and Pradeep Dubey. 2010. 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)*. 339–350.
[40] Adam Kirsch and Michael Mitzenmacher. 2006. Less hashing, same performance: Building a better bloom filter. In *Algorithms–ESA 2006: 14th Annual European Symposium*. Springer, 456–467.
[41] Conglong Li and Alan L Cox. 2015. GD-Wheel: A cost-aware replacement policy for key-value stores. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15)*. 1–15.
[42] Liwen Li, Chunyang Ye, and Hui Zhou. 2022. Cache replacement algorithm based on dynamic constraints in microservice platform. In *Proceedings of the 2022 International Conference on Service Science (ICSS '22)*. IEEE, 167–174.
[43] Hyeontaek Lim, Bin Fan, David G Andersen, and Michael Kaminsky. 2011. SILT: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. 1–13.
[44] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: A holistic approach to fast in-memory key-value storage. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14)*. 429–444.
[45] Kevin Lim, David Meisner, Ali G Saidi, Parthasarathy Ranganathan, and Thomas F Wenisch. 2013. Thin servers with smart pipes: Designing SoC accelerators for Memcached. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. 36–47.
[46] Howard Mao, Sagar Karandikar, Albert Ou, and Soumya Basu. 2014. *Hardware acceleration of key-value stores*. Technical Report. University of California, Berkeley.
[47] John C. McCallum. 2023. Memory Prices 1957+. https://jcmit.net/memoryprice.htm.
[48] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)*. 385–398.
[49] Cheng Pan, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. 2019. pRedis: Penalty and locality aware memory allocation in Redis. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '19)*. 193–205.
[50] Kai Ren, Qing Zheng, Joy Arulraj, and Garth Gibson. 2017. SlimDB: A space-efficient key-value storage engine for semi-sorted data. *Proceedings of the VLDB Endowment* 10, 13 (2017), 2037–2048.
[51] Stephen M Rumble, Ankita Kejriwal, and John Ousterhout. 2014. Log-structured memory for DRAM-based storage. In *12th USENIX Conference on File and Storage Technologies (FAST '14)*. 1–16.
[52] Zhaoyan Shen, Feng Chen, Yichen Jia, and Zili Shao. 2017. DIDACache: A deep integration of device and application for flash-based key-value caching. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17)*. 391–405.
[53] Tobias Vincon, Arthur Bernhardt, Ilia Petrov, and Andreas Koch. 2020. nKV in action: Accelerating KV-stores on native computational storage with near-data

processing. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2981–2984.

[54] Kefei Wang and Feng Chen. 2018. Cascade Mapping: Optimizing memory efficiency for flash-based key-value caching. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '18)*. 464–476.

[55] Kefei Wang, Jian Liu, and Feng Chen. 2020. Put an elephant into a fridge: Optimizing cache efficiency for in-memory key-value stores. *Proceedings of the VLDB Endowment* 13, 9 (2020), 1540–1554.

[56] Yandong Wang, Xiaoqiao Meng, Li Zhang, and Jian Tan. 2014. C-hint: An effective and reliable cache management for rdma-accelerated key-value stores. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '14)*. 1–13.

[57] Fenggang Wu, Ming-Hong Yang, Baoquan Zhang, and David HC Du. 2020. ACkey: Adaptive caching for LSM-based key-value stores. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference (ATC '20)*. 603–615.

[58] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. 2015. LSM-trie: An LSM-tree-based ultra-large key-value store for small data. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (ATC '15)*. 71–82.

[59] Xingbo Wu, Li Zhang, Yandong Wang, Yufei Ren, Michel Hack, and Song Jiang. 2016. zExpander: A key-value cache with both high performance and fewer misses. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. 1–15.

[60] Shuotao Xu, Sungjin Lee, Sang-Woo Jun, Ming Liu, and Jamey Hicksb Arvind. 2016. BlueCache: A scalable distributed flash-based key-value store. *Proceedings of the VLDB Endowment* 10, 4 (2016).

[61] Juncheng Yang, Yao Yue, and Rashmi Vinayak. 2021. Segcache: A memory-efficient and scalable in-memory key-value cache for small objects. In *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI '21)*. 503–518.

[62] Kai Zhang, Jiayu Hu, Bingsheng He, and Bei Hua. 2017. DIDO: Dynamic pipelines for in-memory key-value stores on coupled CPU-GPU architectures. In *Proceedings of the 2017 IEEE 33rd International Conference on Data Engineering (ICDE '17)*. IEEE, 671–682.

[63] Kai Zhang, Kaibo Wang, Yuan Yuan, Lei Guo, Rubao Lee, and Xiaodong Zhang. 2015. Mega-KV: A case for GPUs to maximize the throughput of in-memory key-value stores. *Proceedings of the VLDB Endowment* 8, 11 (2015), 1226–1237.

[64] Teng Zhang, Jianying Wang, Xuntao Cheng, Hao Xu, Nanlong Yu, Gui Huang, Tieying Zhang, Dengcheng He, Feifei Li, Wei Cao, et al. 2020. FPGA-accelerated compactions for LSM-based key-value store. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST '20)*. 225–237.