Hotspot-Aware Hybrid Memory Management for In-Memory Key-Value Stores

Hai Jin, *Fellow, IEEE*, Zhiwei Li, Haikun Liu, *Member, IEEE*, Xiaofei Liao, *Member, IEEE*, and Yu Zhang, *Member, IEEE*

Abstract—Emerging Non-Volatile Memory (NVM) technologies promise much higher memory density and energy efficiency than DRAM, at the expense of higher read/write latency and limited write endurance. Hybrid memory systems composed of DRAM and NVM have the potential to provide very large capacity of main memory for in-memory key-value (K-V) stores. However, there remains challenges to directly deploy DRAM-based K-V stores in hybrid memory systems. The performance and energy efficiency of K-V stores on hybrid memory systems have not been fully explored yet. In this paper, we propose *HMCached*, an in-memory K-V store built on a hybrid DRAM/NVM system. HMCached utilizes an application-level data access counting mechanism to identify frequently-accessed (hotspot) objects (i.e., K-V pairs) in NVM, and migrates them to fast DRAM to reduce the costly NVM accesses. We also propose an NVM-friendly index structure to store the frequently-updated portion of object metadata in DRAM, and thus further mitigate the NVM accesses. Moreover, we propose a benefit-aware memory reassignment policy to address the *slab calcification* problem in slab-based K-V store systems, and significantly improve the benefit gain from the DRAM. We implement the proposed schemes with Memcached and evaluate it with Zipfian-like workloads. Experiment results show that HMCached significantly reduces NVM accesses by 70 percent compared to the vanilla Memcached running on a DRAM/NVM hybrid memory system without any optimizations, and improves application performance by up to 50 percent. Moreover, compared to a DRAM-only system, HMCached achieves 90 percent of performance and 46 percent reduction of energy consumption for realistic (read-intensive) workloads while significantly reducing the DRAM usage by 75 percent.

Index Terms—In-memory key-value store, non-volatile memory, hybrid memory system

1 Introduction

In-memory Key-Value (K-V) stores, such as Memcached [1], [2], have became an important component of modern data center infrastructure. They can significantly reduce response time of user requests by storing performance-critical data in main memory, and thus have fueled many popular data center applications such as Web service, social networking, and e-commerce. As a result, in-memory K-V stores are widely deployed in today's data centers, such as Facebook [2] and Amazon [3]. Most previous studies assume in-memory K-V stores are a volatile cache, with a back-end database to store massive data in persistent storage (e.g., HDD). The performance of those systems are highly dependent on the capacity of DRAM. If objects (i.e, K-V pairs) are not found in the K-V cache, the cost may be several orders of magnitudes higher than directly accessing them in the K-V cache.

With larger memory capacity, in-memory K-V stores can offer higher performance by caching more data in main memory. However, *Dynamic Random Access Memory* (DRAM) technologies are facing scalability problems in terms of density [4], [5] and power consumption [6]. Emerging *Non-Volatile Memory* (NVM) technologies, such as *Phase Change Memory* (PCM) [7], *Resistive-switching RAM* (ReRAM) [8] and *Intel/Micron 3D XPoint* [9] feature byte-addressability, persistence, high density, low cost per bit, near-zero standby power consumption [10]. They are expected to be a competitive replacement to DRAM. However, NVM also has some drawbacks. Both its read/write latencies and write energy consumption are higher than DRAM, and its write endurance is also limited. As a result, it is more practical to use NVM in conjunction with DRAM in a hybrid memory system. However, the challenging problem is how to best utilize NVM and DRAM to fully exploit their advantages and to overcome their drawbacks in K-V stores.

There have been many studies on improving performance and energy efficiency of DRAM/NVM hybrid memory systems. Because of the performance gap between DRAM and NVM, these studies mainly focus on using only a small amount of DRAM to serve a majority of memory references. This goal is achieved typically through *static data placement* [10], [11], [12], [13] and *dynamic memory migration* [14], [15], [16], [17]. The former approaches assume that applications show regular or immutable memory access patterns, and thus they can use offline profiling technologies to guide programmers or compilers for static memory allocations. However, there still exists many applications that shows irregular and unpredictable memory access

- *The authors are with National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China. E-mail: {hjin, leezw, hkliu, xfliao, zhyu}@hust.edu.cn.*

Manuscript received 20 Feb. 2019; revised 18 Aug. 2019; accepted 16 Sept. 2019. Date of publication 4 Oct. 2019; date of current version 10 Jan. 2020. (Corresponding author: Haikun Liu.)
Recommended for acceptance by J. Wang.
Digital Object Identifier no. 10.1109/TPDS.2019.2945315

patterns. *Static memory allocation* may be sub-optimal for those cases. As in-memory K-V stores are a typical middleware of many data center applications, the static data placement approaches are not effective for serving dynamic and bursting user requests.

Dynamic memory migration approaches usually rely on Operating Systems (OSes) or memory controller (hardware) to track memory access frequency (i.e., hotness) at runtime, and migrate hot data between DRAM and NVM periodically. In practice, current computer architectures do not yet provide hardware support for fine-grained memory access monitoring due to the hardware complexity and scalability issues. On the other hand, since modern OSes usually manage memory at the granularity of page (typically 4 KB), it is hard to monitor data accesses at more fine-grained sizes (e.g., objects). Moreover, memory references are not necessarily perceived by OSes because the translation lookaside buffer (TLB) handles a large portion of virtual-to-physical address translations. A few OS-level page migration policies [15], [17] track page accesses by invalidating TLBs, resulting in unacceptable performance overhead at the software layer. The cost of page access counting even exceeds the benefit of page migration in hybrid memory systems. As most in-memory K-V stores (e.g., Memcached) use slab-based memory allocation and requires more fine-grained memory management for K-V objects, OS-level dynamic page migration policies so far are not full-blown and applicable for K-V store systems.

In this paper, we explore how to implement an application-level object migration policy to best utilize DRAM and NVM resource for in-memory K-V stores, such as Memcached. Several challenging problems should be addressed in this system. First, how to monitor the access counts of objects and identify the hot objects in an effective and efficient way? Second, because the NVM write latency is several times higher than that of DRAM, how to redesign the K-V indexes to reduce write operations on NVM? Third, Memcached uses the recency-based LRU algorithm for object replacement in main memory. However, it is not applicable for frequency-based object replacement in hybrid memory systems. Finally, previous work uses object miss rate as an indication to address the slab calcification problem [18], [19] in Memcached. However, they are not efficient to utilize the scarce DRAM resource in hybrid memory systems.

We present *HMCached*, an in-memory K-V store to address the above challenges in DRAM/NVM hybrid memory systems. HMCached organizes the DRAM and NVM horizontally in a single address space while logically using the DRAM as an exclusive cache to the NVM. It only caches all objects' metadata and a portion of frequently-accessed (hot) objects in a small amount of DRAM, and thus can significantly reduce the DRAM usage while offering comparable performance to a DRAM-only system. We propose several novel designs to support hotspot-aware hybrid memory management for in-memory K-V stores, including an NVM-friendly index structure, hotness-aware object migration, access frequency aware object replacement, and benefit-aware DRAM reassignment. The contributions of this paper are as follows:

(1) We develop an application-level object access counting mechanism to identify hot objects in the NVM, and migrate them to the DRAM. These operations are performed by HMCached without any modification to hardware, OSes and user applications. As the object access counters are updated incidentally with the object metadata, the object access counting cause negligible runtime overhead.
(2) As each object request introduces an update of object metadata, we decouples the frequently-accessed portion of metadata from the objects and store it in the fast DRAM. This NVM-friendly index structure can further reduce data accesses to the NVM.
(3) We replace the LRU algorithm in the vanilla Memcachd with a Multi-Queue (MQ) algorithm to adapt to frequency-based object replacement in the DRAM. We also develop a clock algorithm for object replacement in the NVM, and thus significantly reduce the metadata size maintained by the LRU algorithm.
(4) We propose a benefit-aware DRAM reassignment policy to address the slab calcification problem in hybrid memory systems, and thus improve the performance gain from the scarce DRAM resource.

To the best of our knowledge, we are the first to explore object-level hotspot management for K-V stores in hybrid memory systems. We implement the proposed K-V store based on Memcached, and open the source codes of HMCached [20]. We evaluate HMCached with Zipfian-like workloads. Experiment results show that HMCached significantly reduces NVM accesses by 70 percent compared to the vanilla Memcached, and improves application performance by up to 50 percent. Moreover, compared to a DRAM-only system, HMCached approximates 90 percent of its performance by using only 25 percent of its DRAM capacity, and reduces energy consumption by 46 percent on average for realistic (read-intensive) workloads.

The remainder of this paper is organized as follows. We first describe the background and motivation in Section 2. We describe the detailed design and implementation of HMCached in Section 3. Section 4 provides a comprehensive evaluation of HMCached. We discuss the related work in Section 5 and conclude this paper in Section 6.

2 Background and Motivation

2.1 Memcached

Memcached is a high-performance, multi-threaded in-memory K-V store. It provides a number of simple application programming interfaces (APIs) for manipulating the K-V pairs (objects), for example, *SET*/*GET*/*DELETE*. A K-V pair and its metadata are stored tightly in a memory chunk and indexed by a hash table. Memcached solves the hash collision problem by *separate chaining*, i.e., chaining conflicting objects in a linked list. For each object request, Memcached first locates the key in the hash table, and then traverses the corresponding linked list to find the required object upon a hash collision.

The memory management policy of Memcached is simple yet effective. It uses a *slab-based memory management* policy to avoid memory fragmentation, which is often a major problem for buddy memory management systems using *malloc()* and *free()* APIs. For the slab allocation policy, memory is partitioned into multiple *slab classes*, and each slab

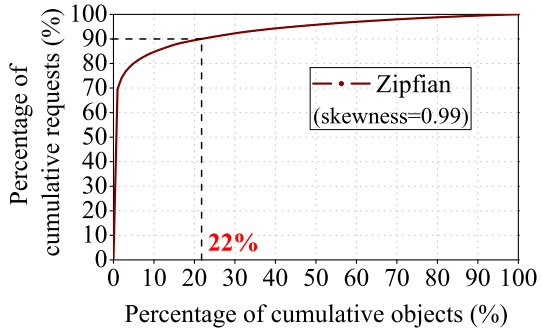


Fig. 1. The cumulative distribution function of object requests (In the *X*-axis, objects are sorted in a descending order according to the number of object accesses).

class has several fix-sized memory blocks, also named *chunks*. Memcached allocates available memory to each slab class at the granularity of *slab* (the default size is 1 MB), and then partition the slab into several fix-sized chunks. The chunk size of each slab class is determined by a geometric progression with a growth factor (e.g., 1.25 or 2). When Memcached allocates a chunk to a new object, it chooses a slab class whose chunk size is able to accommodate the object with a least waste of memory space.

The slab memory allocation often causes a problem known as *slab calcification* [18], [19]. Once a slab has been allocated to a slab class, it cannot be reassigned to another slab class. In other words, the allocated space for slabs can not be adjusted to adapt the dynamic change of access patterns. The default memory replacement policy in Memcached usually cannot best utilize the scarce DRAM resource. When a slab class needs more memory to store new objects, a slab in another slab class should be reassigned to accommodate the new object. However, this memory reassignment policy needs to evict all objects in the selected slab, and leads to non-trivial runtime overhead.

Memcached uses *least recently used* (LRU) algorithm to evict objects within a slab class when there is no available memory for assigning a new slab. For each slab class, Memcached ranks objects in a queue based on the access "recency" of them. When Memcached needs to allocate a chunk for an new object, the LRU based replacement algorithm places it in the head of the queue. The object at the tail of queue is deemed as the oldest objects, and would be evicted first. Note that, the LRU queue is implemented as a doubly-linked list. Thus, the system should maintain two pointers (16 bytes) for each object in the doubly-linked list. Since Memcached is a multi-threaded program, when the LRU chunk is evicted by a thread, the LRU queue should be protected by an exclusive lock to block other modifications.

2.2 Workload Characteristics

Workload characteristics have a significant impact on the performance of Memcached. There have been many analytical and experimental studies on workload characteristics of Memcached clusters, such as Facebook's production workloads [21], [22]. These studies have shown a number of interesting findings for directing the design of in-memory K-V stores in hybrid memory systems.

First, *the request distribution of objects are highly skewed*. A majority of workloads' requests are often distributed on a small portion of objects. The access pattern of requests generally follow a *Zipfian* distribution. Fig. 1 shows the cumulative distribution function curve of object requests that follow a Zipfian distribution with a skewness of 0.99, which is a common setting of realistic workloads in many studies [23], [24]. We can observe that almost 90 percent of the whole requests are distributed on 22 percent of top hot objects. This observation indicates that we have a potential to place the frequently-accessed (hot) objects in a small amount of fast DRAM, and store the large portion of cold objects in the large-size and cheap NVM. Thus, in-memory K-V stores are particularly suitable for running on a DRAM/NVM hybrid memory system, which can approximate the performance of DRAM-based Memcached while significantly reducing the cost of main memory.

Second, *the size of objects often vary over a wide range*. The size of an object may be as small as a few bytes, or as large as several megabytes. Buddy memory allocation for these objects can fragment main memory easily. Moreover, object migration in hybrid memory systems can aggravate the memory fragmentation problem. Thus, slab memory allocation is a better approach to memory management for in-memory K-V stores.

Third, *the access pattern of objects may vary over time*. The memory requirement of different slab classes may be dynamically changed at runtime. This would cause a slab calcification problem in slab-based memory management systems. Although there are some previous studies [18], [19] on addressing the slab calcification problem in a DRAM-based Memcached, they are not directly applicable in a hybrid DRAM/NVM system. There remains challenges to address this problem if a hot object migration scheme is applied to the vanilla Memcached.

Fourth, *the GET operations account for a majority of total object requests for most workloads*. The ratio of GET to SET can be as high as 30. Because each GET operation leads to an additional update of the object metadata in Memcached, this can cause significantly performance overhead if we directly deploy Memcached in DRAM/NVM hybrid memory systems. Thus, a special optimization on the K-V index structure is required to reduce NVM write operations particularly for the GET requests.

2.3 Motivation

The observations of workload characteristics suggest that in-memory K-V stores can benefit significantly from hybrid memory systems in terms of improved memory capacity and considerable cost reduction. However, there still remains several challenges to best utilize hybrid memories.

First, the object access counting mechanism, if not elaborately designed, can cause significant performance overhead because the access counters should be updated for each object request. We should design a lightweight object access counting mechanism to support object migration in hybrid memory systems. Also the decision making of object migration should be sophisticated to adapt to diversifying and dynamic workloads.

Second, the hash index structure in Memcachd is not effective for the NVM due to the following two reasons. (1) Upon a hash collision, multiple NVM accesses are required to retrieve an object due to traversing the linked list. Because

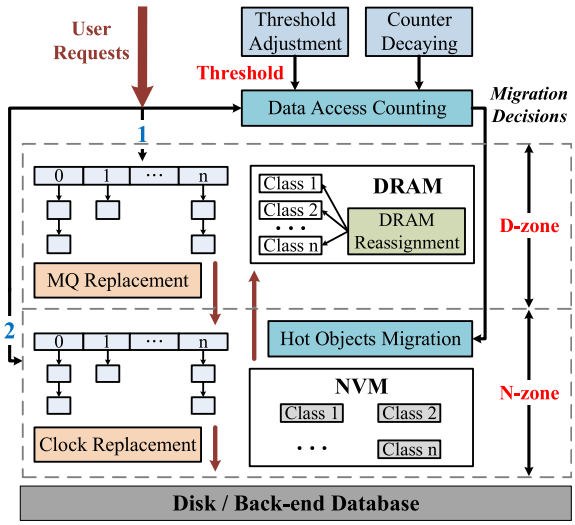


Fig. 2. Architecture of HMCached.

the access latency of NVM is relatively higher than that of DRAM, the costly traversal in the NVM increases the data access delay. (2) Each object request (including GET/SET operations) introduces an additional update to the metadata, such as the access timestamp, etc. In both cases, accessing objects' metadata in NVM suffers higher latency than that of DRAM. Thus, the index structure in Memcached should be redesigned to adapt to hybrid memory systems.

Third, the LRU-based object replacement policy in Memcached is not applicable for hybrid memory systems. The LRU algorithm exploits access "recency" for object replacement in each slab class. However, the "recency" can not reflect the hotness of objects in a long period of time. LRU had been superseded by adaptive replacement cache (ARC) [25] for page cache replacement. We argue that it is more favorable to replace objects in the DRAM ranked by both access "frequency" and access "recency".

Finally, previous work addresses the slab calcification problem in Memcached based on the miss rate of each slab class. However, the high miss rate may be attributed to a larger number of long-tail cold objects. They are not effective for access frequency based DRAM management. We need to design a new DRAM reassignment policy to maximize the benefit of scarce DRAM resource.

3 Design and Implementation

In this Section, we first present an overview of HMCached, and then introduce the detailed design of HMCached, including the K-V index structure, hotness-aware object migration, object replacement policies, and benefit-aware DRAM reassignment strategies.

3.1 Architecture Overview

Fig. 2 depicts the architecture of HMCached. The memory space is divided into a D-zone and a N-zone, in which HMCached stores objects in DRAM and NVM, respectively. We adopt different index structures to record the metadata of objects in the two memory zones (see Section 3.2).

The D-zone and the N-zone are logically organized as an exclusive cache architecture. In our design, GET/SET requests are first served by the D-zone. If the object is not found in the D-zone, it will be retrieved from the N-zone. For SET requests, if the object is not found in both the D-zone and the N-zone, it will be loaded to the N-zone first. However, the DELETE requests are served by both two memory zones.

Similar to Memcached, we use *slab memory allocation* for memory management. Because an object can be migrated only within the same slab class, our design allows that each slab class is composed of both DRAM and NVM concurrently. Each slab class can apply for different types of memories according to its demands till the available memory is used up. Unlike the conventional two-level cache structure, we do not fetch each object in the N-zone to the D-zone in a on-demand manner. Because of the byte-addressability of NVM, the N-zone can return required objects directly. We cautiously perform object migrations since object migrations do not necessarily improve the performance of K-V stores, and the cost of object migrations may even degrade the system performance. To make a tradeoff between the net benefit of object migrations and DRAM utilization, we develop an application-level data access counting mechanism, and propose a threshold-based mechanism that combines dynamic threshold adjustment and access count decaying to identify the hot objects. The object migration decisions are made by our *hotness-aware object migration* mechanism (see Section 3.3). Note that, after object migration, there is only one copy of data for each object that is stored in either the D-zone or the N-zone.

When there is no available memory for the D-zone to accommodate new objects, it needs to reclaim memory for the new objects by evicting cold objects to the N-zone. Similarly, the N-zone should reclaim memory for new objects by evicting some objects out of main memory if there is no free NVM resource available. We adopt multi-queue based and clock-based object replacement policies for the D-zone and the N-zone, respectively, according to different DRAM/NVM features (see Section 3.4).

As described in Section 2.1, the slab memory allocation for both DRAM and NVM can cause the *slab calcification* problem. Previous studies [18], [19] have already addressed this problem in homogeneous memory systems. In this paper, to address this problem in hybrid memory systems, we propose a DRAM reassignment scheme to best utilize the DRAM resource (see Section 3.5).

3.2 NVM-Friendly Index Structure

We maintain two different hash tables for indexing objects in the D-zone and the N-zone separately. Each object consists of three segments of data: metadata, key, and value. For the D-zone, we store the three segments of an object in a contiguous memory space, and resolve the hash collision problem by *separate chaining*, as shown in Fig. 3a. This index structure works well for the D-zone. However, it is not effective for the N-zone due to the much higher write latency of NVM, as described in Section 2.3. Because a portion of object metadata is frequently read and updated, storing it together with keys/values in the NVM would cause significant performance overhead.

In order to improve the access performance of object in the N-zone, we design a NVM-friendly index structure to reduce read/write accesses to NVM when serving GET requests in the N-zone. We decouple the frequently-updated fields from

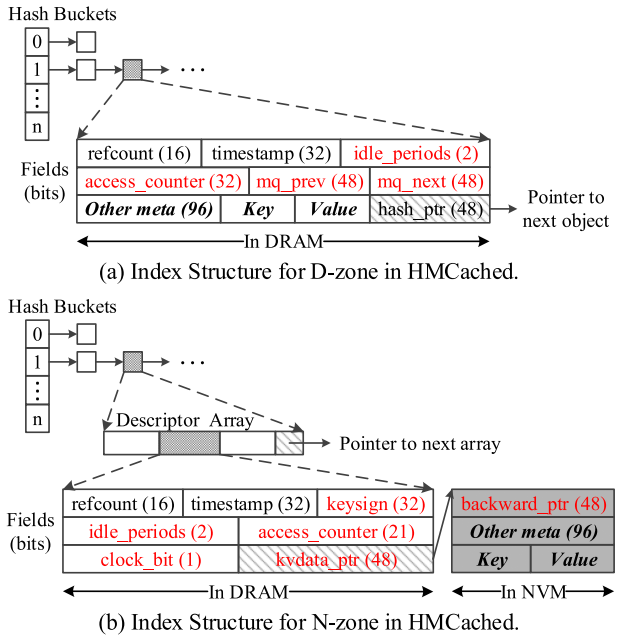


Fig. 3. Index Structures of HMCached. The red fields are newly added compared to the vanilla MemCached.

the K-V objects and use a *descriptor* structure to store them in DRAM separately. As shown in Fig. 3b, each descriptor consumes 19 bytes DRAM, and uses a pointer (*kvdata_ptr*) to point to the corresponding K-V pair and the other portion of metadata. We also maintain a backward pointer (*backward_ptr*) to reversely retrieve the portion of metadata in the DRAM. To align the data of descriptor, we package three descriptors into an array, with a 48-bit pointer pointing to the next array. Totally, an array contains 63 bytes data and can be aligned in 64 bytes (i.e., the cache line size). Thus, each array can be filled into the CPU cache by one memory access.

Each descriptor contains a very important field called "keysign", which is a hash-based signature of an object's "key". As we should reduce NVM reads when retrieving an object, a direct approach is to put the "key" in the descriptor. However, the size of keys may be different and too large. Thus, we use a fix-sized (32 bits) signature as a replacement for the keys. When an object is retrieved, we first calculate the hash signature based on the required object's "key" and compare it with a series of hash signatures in the list of descriptors. Once the corresponding "keysign" is found in a descriptor, we access the NVM to further match the "key".

The other portion of the descriptor are the frequently-updated metadata. The selection is based on whether a field of the metadata is updated when serving the GET requests. As we replace the LRU-based object replacement algorithm with a simple *clock* algorithm [26], [27], the memory space used for the two pointers (16 bytes) in the LRU list for each object have been saved. However, we also need to maintain one bit (*clock_bit*) for each object in the *clock* algorithm. As shown in Fig. 3, the *refcount* field is used to record the number of references to an object. The *timestamp* field is used to record the least access time of an object. The *idle_periods* is used to record the number of consecutive time periods in which the object is not accessed. The *access_counter* fields are used to record the access counts of objects.

In summary, to reduce the cost of NVM accesses, we have made two important designs for the K-V index structure in HMCached. First, we use the clock algorithm rather than the LRU algorithm for object replacement in the N-zone, and thus significantly reduce the size of metadata that should be frequently updated by the LRU algorithm, and also reduce the number of memory accesses due to updating the doubly-linked list in the LRU algorithm. Second, we decouple the frequently-updated fields of metadata from K-V objects and store them in the fast DRAM separately. This mechanism not only accelerate the accesses of K-V metadata, but also significantly reduce the NVM writes due to updating the metadata on each K-V request.

3.3 Hotness-Aware Object Migration

Object Access Counting. Because the memory capacity of the D-zone is usually limited and object migrations lead to non-trivial performance overhead, we should cautiously decide whether an object should be migrated to the D-zone. Because a frequently-requested object generally leads to a large amount of memory accesses, we estimate the hotness of an object by counting the total number of GET/SET requests. We record the number of access counts by using a counter (the *access_counter* field as shown in Fig. 3). The *access_counter* uses 4 bytes in the D-zone, while uses 21 bits in the N-zone in consideration of data alignment.

We note that a portion of hot objects with a strong temporal locality are resided in the on-chip cache, which can filter a larger number of data requests in main memory. However, as our object access monitoring mechanism is implemented at the application level, the value of access counter (*access_counter*) increases for each request to an object, no matter in the on-chip cache or in the main memory. Thus, for these hot objects, although they are often hit in the cache, we still have to migrate them to the fast D-zone for better performance. As the metadata of objects should be updated on each request, we update the access counter incidentally. This introduces trivial performance overhead of Memcached. Although the cache filtering has a impact on the actually memory accesses, our estimation is still reasonable and effective. Once the value of the access counter exceeds a given threshold, *migration_threshold*, we treat it as a hot object and migrate it to the D-zone immediately.

Access Counter Decaying. We hope that the access counters can reflect the hotness of objects, so that the object migration based on these access counts is effective. However, some objects may have a very long lifetime, and the accumulated object requests can finally let the counter value to be larger than the given hotness threshold. In fact, the migration of these objects to the D-zone is unnecessary and even degrades the efficiency of DRAM. To address this problem, we limit the growth of access counters by decaying their values periodically. In this way, we actually use the number of access counts in a given time period (i.e., access frequency) to reflect the hotness of an object.

We decay the value of access counters at different speeds. Since an object may be unlikely accessed if it has not been accessed for a long time, we record the number of consecutive periods that an object has not been accessed recently using the *idle_periods* field, as shown in Fig. 3. In the end of each time period, for each object, we update its counter

value by dividing it with $2^{(idle\_periods+1)}$. Furthermore, if an object has not been accessed for three consecutive periods, we reset its counter value as zero radically. The decaying operations are performed for all objects in the system concurrently. We use the total requests served by HMcached as a logical period for the decaying operations periodically (thirty million requests in our experiments).

Dynamic Threshold Adjustment. We notice that the setting of *migration_threshold* can have a significant impact on the efficiency of DRAM. Moreover, the access pattern of workload often changes over time, and thus a static threshold cannot adapt to dynamic workloads. To address this problem, we dynamically adjust the threshold to adapt the access pattern of current workload periodically. At the end of each period, we first calculate the benefit ($Benefit_{mig}$) brought by migrating hot objects to the D-zone. Intuitively, if a larger proportion of object requests are served by the D-zone, HMcached can deliver higher application performance. However, frequent migrating objects can also degrade the system performance because object migrations introduce data swapping between the D-zone and the N-zone. Thus, we count the number of accesses to objects in the N-zone caused by object migrations, and take the cost into account when calculating the benefit of object migration. We use Equation 1 to calculate $Benefit_{mig}$ as follows.

$$Benefit_{mig} = \frac{C_{dreq} - C_{mig}}{C_{dreq} + C_{nreq}}, \tag{1}$$

where C_{dreq} denotes the access requests served by the D-zone, C_{nreq} denotes the access requests served by the N-zone, C_{mig} denotes the number of object accesses caused by the object migration itself.

Algorithm 1. Dynamic Migration_threshold Adjustment

```
1: if C_{N-to-D} < DRAM_free / 2 then
2:   migration_threshold--
3: else
4:   if Benefit_mig in period_i > Benefit_mig in period_{i-1} then
5:     if migration_threshold increased in the previous period
       then
6:       migration_threshold++
7:     else
8:       migration_threshold--
9:     end if
10:  else
11:    if migration_threshold decreased in the previous period
       then
12:      migration_threshold++
13:    else
14:      migration_threshold--
15:    end if
16:  end if
17: end if
```

We use the $Benefit_{mig}$ in Algorithm 1 to guide the dynamic adjustment of *migration_threshold*. Assume $DRAM_{free}$ denote the number of free DRAM chunks, C_{N-to-D} denote the number of objects migrated from the N-zone to the D-zone in the current period. In the beginning, we first check whether C_{N-to-D} is smaller than half of the free DRAM chunks (line 1-2). If it is true, we decrease the threshold to migrate more objects to the D-zone, and thus improve the utilization of DRAM resource. This process often occurs when there is plenty of available DRAM in HMcached. Otherwise, we use a hill climbing algorithm to adjust the threshold (line 4-16). The hill climbing algorithm is simple for greedy searching, and it can search local optimal solution with very low computation overhead [16], [28]. As shown in the line 4 to 16, we compare the value of $Benefit_{mig}$ in the current period with the value in the previous period. If the value in current period is larger, meaning that the adjustment we did in the previous period is effective, we do the same adjustment as the previous period. Conversely, if the value of $Benefit_{mig}$ in the current period is lower than the value in the previous period, we conduct an opposite adjustment operation. Note that we use different thresholds for different slab classes, we use the number of requests served by each slab class as a logical time period for adjustment (10^5 requests in our experiments).

3.4 Object Replacement Policies

Multi-Queue Based Object Replacement for the D-zone. The object replacement policy of each slab class in the D-zone have a significant impact on system performance. Without an effective policy, object replacement may lead to frequent object swapping between the D-zone and the N-zone, similar to the *cache thrashing* problem. Because objects in the D-zone are frequently-accessed objects, the LRU algorithm is hard to rank them by using only the "recency" feature. We adopt *Multi-Queue* algorithm [14], [29] to replace objects in the D-zone. Our policy ranks objects by using the "recency" and "frequency" features simultaneously. It is composed of N queues (8 in HMcached), and each queue uses an LRU-based replacement policy. The queues at the higher levels handle objects with higher access frequency, and the MQ algorithm decides which queue an object should be prompted/demoted according to the rank of object hotness. By using the *access_counter* field which we have added to the metadata of each object, we can implement the MQ algorithm for the D-zone without additional space overhead.

On each request to an object, we place it in the *i*th queue if the *access_counter* is larger than 2^i but lower than 2^{i+1}. Besides, we place an object at a lower queue when we decay its *access_counter*. Note that, HMcached is a multi-threaded program, thus we use an exclusive lock to guarantee the exclusive access to each queue among different threads. However, the multiple queues can be accessed by different threads concurrently. Compared to the LRU algorithm, we improve the parallelism of replacement operations. When we need to evict an object in the D-zone, we traverse from the queue *0* to the queue *N-1*, and for each queue we traverse from the tail to the head till we find the first object for replacement.

Clock Based Object Replacement for the N-zone. If we use a LRU-based object replacement policy for the N-zone, the LRU needs to maintain two pointers (16 bytes) in the K-V index to construct the doubly-linked list. Because the objects in the N-zone are accessed infrequently, we adopt a simple *clock* algorithm [26], [27] as an alternative of the slab replacement policies for the N-zone. The clock algorithm is an approximate LRU algorithm, but needs much less memory space to record the access status of an object. It only needs to

maintain one bit to record the access recency for each object. Compared to the LRU algorithm, the clock-based replacement policy can also significantly reduce the number of NVM writes by avoiding updates of the doubly-linked list.

3.5 Benefit-Aware DRAM Reassignment

In this Section, we introduce a benefit-aware DRAM reassignment scheme to solve the *slab calcification* problem for the D-zone. Unlike the previous memory reassignment schemes that are based on the miss rate of different slab classes [18], [19], [30], we estimate the optimal DRAM reallocation for all slab classes periodically based on the access frequency of objects. Our target is to maximize the benefit of storing hot objects in the DRAM.

Assume the performance gap between requesting an object with the size S_0 from the D-zone and the N-zone is P_0. It is equal to the benefit of storing this object in the D-zone. The normalized benefit for a unit size of object becomes P_0/S_0. Assume there are total N slab classes and the number of slabs in the DRAM is M. As described in Section 3.3, each slab class may have a small portion of DRAM resource to store the top hot objects. Thus, the benefit of allocating m ($1 \leq m \leq M$) DRAM slabs to the nth slab class ($1 \leq n \leq N$) can be calculated using Equation 2.

$$Benefit[n][m] = \sum_{i=1}^{k} C_i * S_n * P_0/S_0 \tag{2}$$

where k denotes the number of hot objects in which m DRAM slabs can accommodate, and C_i ($1 \leq i \leq k$) denote the corresponding value of *access_counter* of object i, and $\sum_{i=1}^{k} C_i$ denotes the total number of requests to the k hot objects, and S_n denotes the chunk size allocated by the nth slab class.

In all, the value of $DRAM_{Benefit}$ can be calculated using Equation (3).

$$DRAM_{Benefit} = \sum_{n=1}^{N} Benefit[n][D_{opt}[n]], \tag{3}$$

where $D_{opt}[n]$ ($1 \leq n \leq N$) denotes the optimal size of DRAM allocated to the nth slab class.

Now, the optimal DRAM allocation problem is to select a set of optimal values of $D_{opt}[n]$ ($1 \leq n \leq N$), so that the total $DRAM_{Benefit}$ can be maximized under the constraint of Equation (4). The optimal DRAM allocation solution can be searched by using a simple three-dimensional dynamic programming algorithm [18].

$$\sum_{n=1}^{N} D_{opt}[n] = M. \tag{4}$$

A key problem is how to count $\sum_{i=1}^{k} C_i$ ($1 \leq i \leq k$) in an efficient way. An intuitive approach is to traverse the hash tables of both the D-zone and the N-zone to collect the access counts of objects for each slab class, and then sort them in a descending order. At last, we can select the top k hot objects and calculate the total access counts of those objects. However, traversing the whole hash tables is often too slow and causes high runtime overhead if there are millions of objects in the HMcached. To address this issue, we use an ordered data structure, such as red black tree (RBTree), to construct a *mapping set* (*MS*) in the form of $V \rightarrow O$, where V denotes the value of *access_counter*, and O denotes the number of objects. For example, a mapping "5 →10" indicates that 10 objects have the same access counts (5) in the current sampling period. We update the *MS* upon each object request. Because the MS has much less items than that of the whole hash tables, we can count $\sum_{i=1}^{k} C_i$ ($1 \leq i \leq k$) more efficiently by traversing the *MS*.

Note that we should use an exclusive lock to guarantee the safety of updating the MS among multiple threads. However, this approach have a potential to degrade the workload performance because of lock contentions. To solve this problem, we develop a lock-free array between the service threads and the *MS*. Since the size of an access counter is only 4 bytes in our system, the new value of the *access_counter* can be appended to its old value upon each update of the counter. Thus, the service threads can update the lock-free array using 8-byte atomic operation. This design allow multiple service threads can concurrently update the same item in the MS. Finally, a background thread fetches these updates from the lock-free array and merges them into the MS. In this way, we reduce the time spent in updating the MS for service threads.

4 Evaluation

In this Section, we first introduce the experiment setup, and then evaluate the efficiency of hot object migration scheme in terms of throughput, reduction of NVM write operations, and energy consumption in HMcached. We also evaluate the effectiveness of NVM-friendly index structure and benefit-aware DRAM reassignment schemes.

4.1 Experimental Methodology

NVM Emulation. Since the commercial NVM device (3D XPoint) [9] is still not available in our Lab, we use an NVM emulator called HME [31], [32] to emulate the performance characteristics of NVM devices. HME is based on an NUMA architecture and emulate the DRAM on a remote node as the NVM. It periodically counts the number of read/write accesses to the remote DRAM, and estimates the total software-generated delay that should be injected to the applications to emulate the NVM accesses. The number of memory accesses can be tracked by the Intel Performance Monitoring Unit (PMU) tools [33]. Moreover, HME can also estimate the total amount of data read/written from/to memory, and then we can further estimate the energy consumption of memory accesses. We deploy HME on a NUMA-based server, which is equipped with two-socket Intel Xeon CPU E5-2650 v3 @ 2.30 GHz processors and 128 GB DRAM. We use 64 GB DRAM on a NUMA node to emulate the NVM. The latency of DRAM is 76 ns in our server, and we set the NVM read/write latencies as 300 ns and 1000 ns [34], respectively.

Alternative Systems for Comparison. We compare HMCached with a set of representative policies as follows: (1) *HM-UNI*: a vanilla Memcached running on a DRAM/NVM hybrid memory system without specific memory management optimizations, and objects are stored in the DRAM

and NVM uniformly according to the ratio of the DRAM size to the dataset size. (2) *HM-TLC*: it organizes the DRAM and NVM in a two-level cache/memory hierarchy. All requests are served by the DRAM, and objects in the NVM are fetched into DRAM on demand. HM-TLC evicts cold objects in the DRAM to NVM using the LRU algorithm. HM-TLC works well for workloads with good data locality. (3) *HM-HC*: A variant of HMcached in which only the DRAM cache replacement policy is replaced by the policy of hyperbolic caching [35]. It considers both access frequency and recency for cache replacement. (4) *DRAM-only*: an vanilla Memcached running on the DRAM solely. We use this system as a reference of the performance upper bound.

Datasets and Workloads. In our experiments, we construct five datasets with different numbers of unique objects and distributions of object sizes. We use *Uni-X-Y* to denote a dataset which contains *X* million unique objects, and the object sizes follow an *uniform distribution* from 1 byte to *Y* bytes. The five datasets are represented by Uni-3-2K, Uni-3-4K, Uni-3-8K, Uni-2-8K and Uni-1-8K. The size of keys are 10 bytes for all datasets. In each experiment, we generate a series of mixed GET/SET requests with a Zipfian distribution of skewness 0.99, using the same configuration as YCSB [36]. We set the proportion of GET requests as 95 and 50 percent in each experiment to evaluate the system performance of read-intensive workloads and write-intensive workloads, respectively. Note that, the first three datasets show the same access intensity but have different distributions of object sizes. In contrast, the last three workloads show a reversed feature compared with the first three datasets. For example, although the top 10 percent objects of Uni-3-8K and Uni-1-8K receive the same number of requests, there are 0.3 and 0.1 million objects for the top 10 percent objects of Uni-3-8K and Uni-1-8K, respectively. Thus, the workload on the dataset Uni-1-8K shows higher access intensity.

Experimental Settings. In each experiment, we first load all objects into the K-V store system, and then use a *Request Generator* to send requests to the system. We use four service threads in each experiment. We set the chunk size of the first slab class as 96 bytes, and increases the chunk size of the subsequent slab classes by a factor of 1.25. If not otherwise specified, we configure the capacity of DRAM as only 1/4 of the total dataset size, and generate 150 million requests in each experiment.

4.2 Performance Studies

Fig. 4 shows the throughput (requests per second) of workloads running on different in-memory K-V stores. For the read-intensive workloads, HMCached achieves 36, 18, and 5 percent performance improvement on average compared to HM-UNI, HM-TLC, and HM-HC, respectively. The performance gap between HMCached and the DRAM-only system is as low as 12 percent. For the write-intensive workloads, HMCached achieves 68, 21, and 4 percent performance improvement on average compared to HM-UNI, HM-TLC, and HM-HC, respectively. When the size of objects become larger (Uni-3-2K versus Uni-3-8K), the throughput of workloads also declines. Because NVM writes are much more costly than NVM reads, and HMCached can place more write-intensive objects in the DRAM, HMCached can achieve higher performance

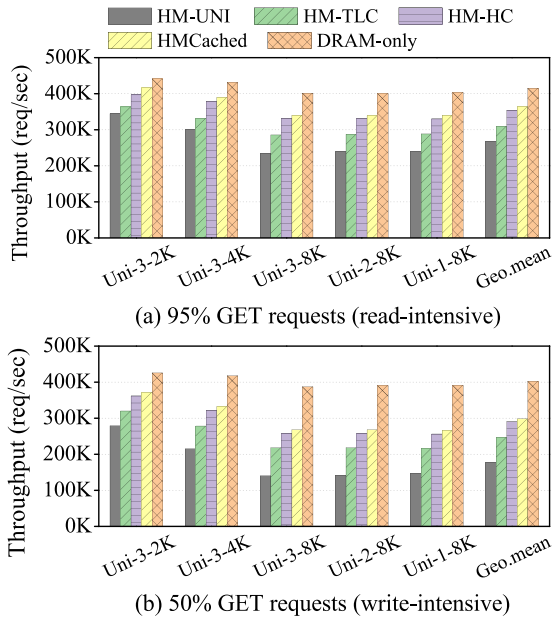


Fig. 4. The throughput of workloads in different systems.

improvement for the write-intensive workloads than the read-intensive workloads. However, the performance gap between HMCached and the DRAM-only system becomes larger (26 versus 12 percent) for the write-intensive workloads because of much higher NVM write latency and the increased portion of NVM writes on the cold objects. HM-HC achieves similar performance to HMCached because the two policies all exploit both access frequency and object residency time for DRAM cache replacement.

Fig. 5 shows the number of NVM accesses in those K-V store systems. Note that each memory access represent a read/write request to 64 bytes data according to our memory access monitoring approach. Note that the number of NVM accesses has included NVM accesses caused by background threads. For the read-intensive workloads, HMCached reduces NVM accesses by 70, 38, and 7 percent on average compared to HM-UNI, HM-TLC, and HM-HC, respectively. For the write-intensive workloads, HMCached reduces NVM accesses by 72, 38, and 6 percent on average compared to HM-UNI, HM-TLC, and HM-HC, respectively. We can find that the number of memory accesses reflects a roughly exponential growth when the largest size of objects increase from 2 KB to 8 KB. For HM-TLC, the reduction of NVM reads is close to HMCached, because it caches the recently-accessed objects in the DRAM, which can filter a large portion of memory accesses to the NVM. However, HM-TLC leads to much more NVM writes. The reason is that HM-TLC should first load all objects to the DRAM on-demand, and then returns the requested data, resulting in plenty of object swapping between the DRAM and the NVM when the DRAM resource is used up. In contrast, HMCached only migrates frequently-accessed objects to the DRAM, and thus improves the efficiency of DRAM usage, mitigating the object swapping operations between the DRAM and the NVM.

4.3 Sensitivity Studies

At first, we study how the system performance is sensitive to different DRAM/NVM configurations. As the emulated

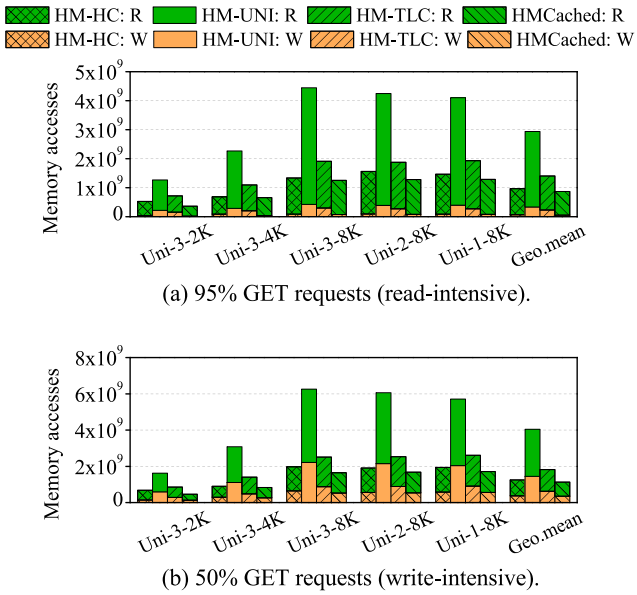


Fig. 5. The number of NVM accesses in different systems (R and W denote NVM reads and writes, respectively).

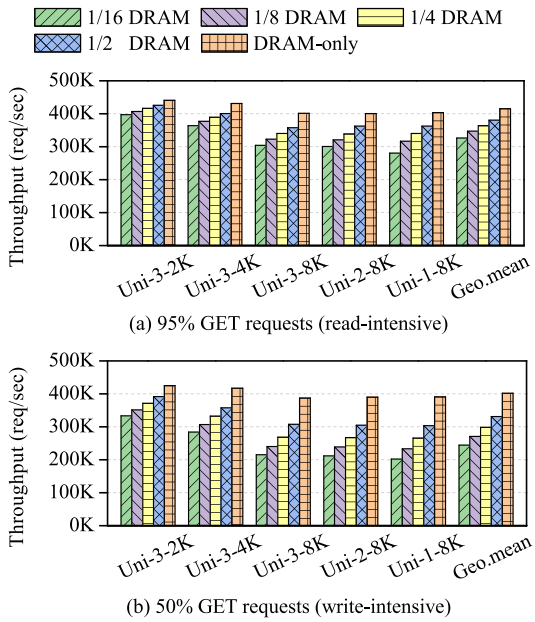


Fig. 6. The throughput of workloads in HMCached with different available DRAM size.

NVM is always sufficient for the evaluated workloads, we only change the size of DRAM in this experiment. Fig. 6 shows the throughput of workloads when the size of DRAM in HMCached is configured as 1/16, 1/8, 1/4, and 1/2 of the total dataset size, respectively. We also use the DRAM-only systems as references. For read-intensive workloads, when the DRAM size increases exponentially, HMCached achieves roughly linear performance improvement compared to the system using minimum DRAM. This implies the benefit of using more DRAM resource declines. Compared to the read-intensive workloads, HMCached can achieve higher application performance improvement for the write-intensive workloads because it can significantly reduce the costly NVM writes by caching the hot data in the DRAM. In summary, HMCached can improve the system throughput by using only a small size of DRAM, and mitigates its sensitivity to the size of DRAM by caching only very hot objects in the DRAM.

We also study the impact of different NVM read/write latencies on the performance of HMcached. We linearly increase the NVM read/write latencies to enlarge the performance gap between DRAM and NVM. Fig. 7 shows that the decrease of workload throughput is roughly linear to the increase of NVM read/write latencies. For the dataset Uni-3-2K, the workload only show slight performance slowdown with higher NVM read/write latencies. However, the workload throughput declines much faster when the objects become larger. This implies the NVM access latencies have a higher impact on the performance of HMcached when the object size become larger.

4.4 Reduction of Memory Energy Consumption

In this Section, we evaluate energy consumption of different K-V store systems. We use a statistical approach introduced by [31], [32] to estimate the energy consumption of hybrid memories. The DRAM energy consumption can be estimated by $E_d * (R_d + W_d) + P_s * T$, where R_d and W_d denote the total amount of data read/written from/to the DRAM, respectively, E_d denotes joule per byte consumed by DRAM reads/writes, P_s denotes the static power consumption of a unit of DRAM, and T denotes the program execution time. The energy consumption of NVM can be estimated by $E_{nr} * R_n + E_{nw} * W_n$, where R_n and W_n denote the total amount of data read/written from/to the NVM, E_{nr} and E_{nw} denote the NVM read and write energy consumption, respectively.

In our servers, we measure the DRAM power consumption based on Intel Processor Counter Monitor [37]. We find that the static power consumption is about 4W for 64 GB DRAM, and the dynamic power consumption is about 1 Joule for reading/writing 4 GB data from/to the DRAM. We refer the work [16] to configure the NVM energy consumption. The energy consumption of NVM reads is the same as that of DRAM reads/writes (i.e., $E_{nr} = E_d$), while the energy consumption of NVM writes is 10 times of the DRAM reads/writes (i.e., $E_{nw} = 10E_d$).

Fig. 8 shows the energy consumption of hybrid memories in different K-V store systems. HMCached introduces the minimum energy consumption on average compared to other systems. For the read-intensive workloads,

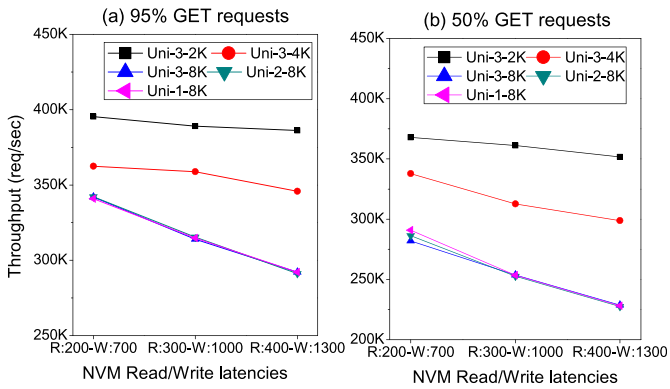


Fig. 7. The throughput of workloads in HMCached with different NVM read/write latencies.

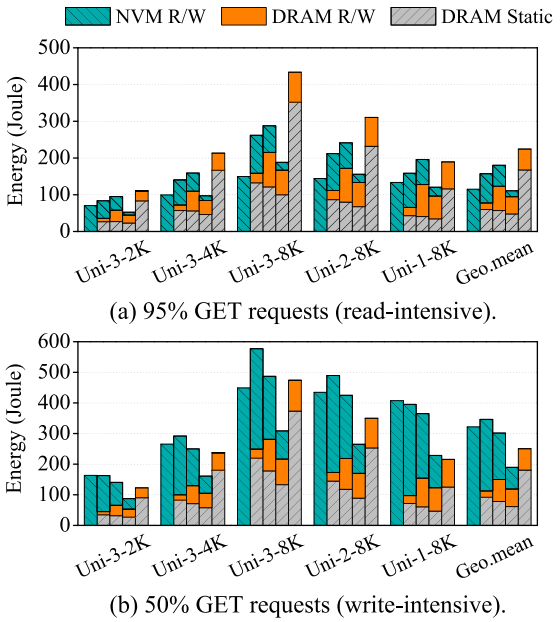


Fig. 8. The estimated energy consumption of hybrid memories in the different K-V store systems. For each dataset, the five bars from left to right represent HM-UNI, HM-TLC, HM-HC, HMCached, and DRAM-only.

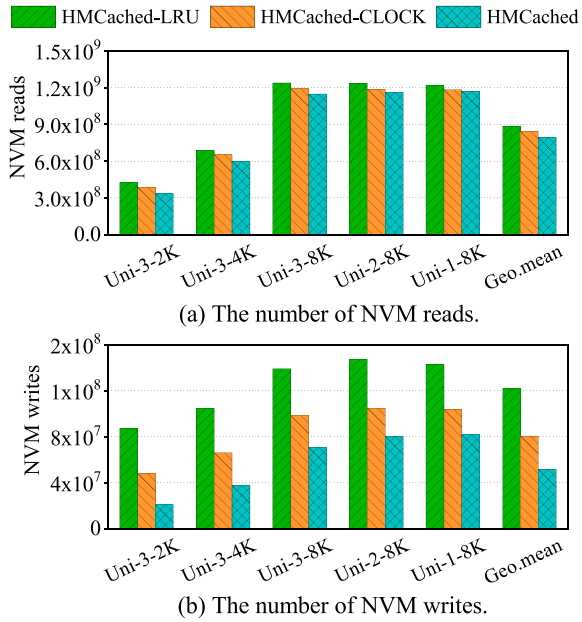


Fig. 9. The number of NVM reads/writes for different polices.

HMCached reduces 31, 37, 18, and 46 percent energy consumption on average compared to HM-UNI, HM-TLC, HM-HC, and DRAM-only, respectively. As the energy consumption of NVM reads is set the same as that of DRAM reads, using more DRAM resource would introduce more static energy consumption, as reflected by the worksets of Uni-3-2K, Uni-3-4K, and Uni-3-8K. Although the workload requests are the same in these experiments, we find that the portion of dynamic energy consumption for different systems are not equal. The reason is that the numbers of actual memory read/write accesses are different for systems due to retrieving and updating the metadata of K-V objects, and object migrations. HMCached avoids costly NVM writes by updating the metadata in the DRAM, and thus reduce the total energy consumption compared to other systems. For the write-intensive workloads, as the energy consumption of NVM writes is much higher than that of NVM reads, the energy consumption of NVM reads/writes account for the majority of energy consumption for HM-UNI, HM-TLC. However, HMCached still reduces energy consumption by 44, 29, 15, and 26 percent on average compared to HM-UNI, HM-TLC, HM-HC, and DRAM-only, respectively. Because HMCached can significantly reduce NVM accesses by caching the write-intensive hot data in the DRAM.

4.5 Efficiency of the NVM-Friendly Index Structure

As described in Section 3.2, the reduction of NVM accesses is attributed to two key designs for the K-V index structure: using the clock algorithm to reduce the size of metadata that should be frequently updated by the LRU algorithm, and storing the frequently-updated portion of the object metadata in the DRAM to avoid NVM writes. In this Section, we study how much NVM accesses can be reduced by this NVM-friendly index structure.

We implement two variants of HMCached for comparison: *HMCached-LRU* and *HMCached-CLOCK*. Both HMCached-LRU and HMCached-CLOCK adopt the same index structure with Memcached for the N-zone. HMCached-LRU and HMCached-CLOCK use the LRU-based and clock-based object replacement policies for the N-zone, respectively. Since the proposed K-V index structure mainly aims to reduce NVM reads/writes when HMCached serves GET requests in the N-zone, we only evaluate the read-intensive workloads (i.e., the proportion of GET requests is 95 percent).

Fig. 9 shows the number of NVM reads/writes for the three schemes. Due to hot object migrations, we find that only about 12 percent GET requests are served by the N-zone totally in our experiments. Compared to HMCached-LRU and HMCached-CLOCK, HMCached reduces the NVM reads by 10 and 6 percent, respectively, and reduces the NVM writes by 58 and 36 percent, respectively. Although our index structure designs can avoid memory accesses to the object metadata on the NVM, a majority of NVM reads are caused by the read-intensive workloads (i.e., the read accesses of *keys* and *values*), and thus the reduction of NVM reads is not as high as that of NVM writes. Compared to the LRU algorithm, the CLOCK algorithm can reduce plenty of NVM writes because it significantly reduces the size of metadata maintained by the algorithm. The LRU algorithm needs to updates many pointers of the doubly-linked list upon each request to an object. Moreover, because each GET request to an object also leads to an update (write) of the object metadata, storing the frequently-updated metadata in the DRAM also significantly reduce NVM writes.

4.6 Effectiveness of DRAM Reassignment

In this Section, we study whether our DRAM reassignment policy can mitigate slab calcification for dynamic workloads.

We construct dynamic workloads for simulating the case of slab calcification. The dynamic workloads access two datasets (*A* and *B*) with different distributions of object sizes. The two datasets all contain 1 million unique objects and total 4 GB data. We only use 1 GB DRAM for

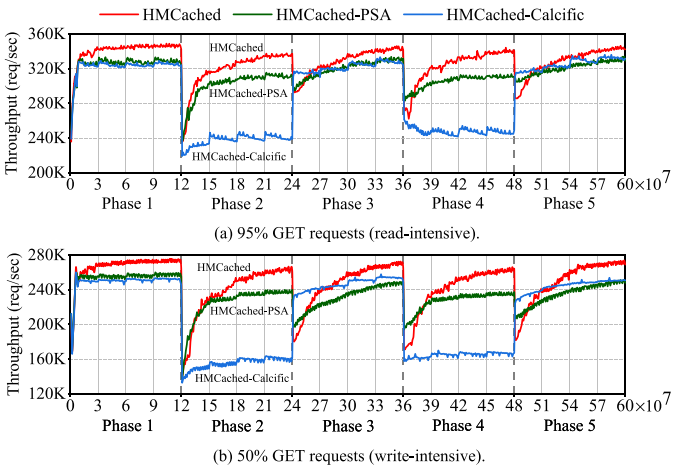


Fig. 10. The throughput of workloads in different systems.

HMCached in our experiments. The object sizes of dataset A follow normal distribution with a median of 4 KB and a standard deviation of 1 KB, and dataset B follows an uniform distribution from 1 byte to 8K bytes. We generate 600 million mixed GET/SET requests following a Zipfian distribution with a skewness of 0.99. We divide these requests into five phases, in which these requests access the two datasets alternately.

We also implement two variants of HMCached for comparison: *HMCached-PSA* and *HMCached-Calcific*. HMCached-PSA adopts *periodic slab allocation* (PSA) [30] as the DRAM reassignment policy. PSA exploits the number of object misses of slab classes as an indication for memory reassignment. HMCached-Calcific does not use any DRAM reassignment policy and would suffer the slab calcification problem. We use this system as a reference to show the influence of slab calcification.

Fig. 10 shows the throughput (requests per second) of different phases for the three schemes. The x-axis presents the number of requests ($\times$10 millions). We change the accessed datasets in 120 million requests periodically. In the first phase, the DRAM are allocated to dataset A, and the three schemes achieve similar performance. When the workloads begin to access dataset B at the beginning of phase 2, the throughput declines sharply because the requested objects miss in the DRAM. The slab classes also change because the dataset is changed to B. Both HMCached and HMCached-PSA begin to reclaim DRAM and re-assign it to dataset B, and thus their throughputs increase continuously. This implies that the DRAM reassignment policies are particularly effective when the access pattern of dataset is changed. However, HMCached-Calcific can not re-assign DRAM resource to the new slab classes, and suffers significant performance degradation due to the slab calcification problem. When the accessed dataset is changed to A again, the performance of HMCached-Calcific is close to HMCached and HMCached-PSA because most hot objects in dataset A is already cached in the DRAM. For both read-intensive and write-intensive workloads, HMCached achieves higher throughput than HMCached-PSA because it considers the overall benefit gained from DRAM for DRAM reassignment, while HMCached-PSA simply uses cache miss rates. These experimental results

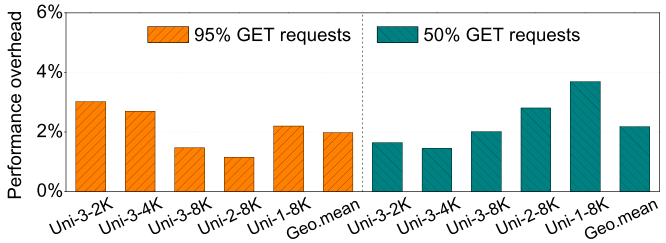


Fig. 11. Normalized performance overhead of HMCached.

demonstrates the effectiveness of our policy for solving the slab calcification problem.

4.7 Performance and Storage Overhead

We construct a simple experiment to measure the performance overhead caused by our strategies. We run the vanilla Memcached and HMCached on a DRAM-only server, and get the workload throughputs T_1 and T_2, respectively. The performance overhead can be calculated as $(T_1 - T_2)/T_1$.

Fig. 11 shows the performance overhead of hybrid memory management schemes in HMCached, all normalized to the vanilla Memcached. For all workloads, the runtime overheads are less than 5 percent. The overhead is attributed to object access counting, the computation of "keysign", hot object migration, and MQ-based object replacement. However, the overhead is offset by the benefit of HMCached. We note that the periodical access counter decaying and DRAM reassignment are performed by a background thread, and thus the CPU time is not on the critical path of K-V requests.

HMCached also introduces slight storage overhead due to some additional data structures. The first portion of overhead is the additional data structure in the metadata of K-V objects, including access counters and address pointers. To guarantee data alignment, we use 4-byte and 21-bit access counters for objects in the D-zone and N-zone, respectively. The second portion of overhead is caused by the mapping set (MS) which is used to select the top k hot objects. Because each item in the MS only requires 8 bytes, the space overhead of MS can be estimated by counting the number of unique objects in the HMcached. The MS only consumes 8 MB memory if there are one million unique objects in the system. The third portion of overhead is caused by the lock-free array. Each element of lock-free array requires 8 bytes. In our experiments, we maintain a lock-free array for each slab class, and each lock-free array contains about 0.1 million elements, resulting in about 0.8 MB memory for each slab class. Because there are total 31 slab classes in our experiments, the lock-free arrays only consume total about 25 MB memory.

5 Related Work

We introduce the related work in the following categories.

Data Placement in Hybrid DRAM/NVM Systems. There are generally two kinds of data placement polices in DRAM/NVM hybrid memory systems. The first one can be *static memory allocation* polices. Wu et al. observe that the MPI-based programs have similar access patterns in iterative execution phases, and thus analyze the access pattern of

objects in the first iteration to direct the data placement of subsequent iterations [12] in hybrid memory systems. Wu et al. also improve the data placement of task-parallel programs [13] by analyzing the similar programs. Static memory allocation generally requires that programs have regular memory access patterns. However, as in-memory K-V store systems are usually used as middleware for many data center applications, their memory access behaviors are too complex to predict. Thus, the *static memory allocation* polices are not effective for in-memory K-V store systems.

Dynamic memory migration polices are another approach to data placement in hybrid memory systems. They often rely on hardware or OSes to track page access counts and migrate the hot page to the fast DRAM at runtime. Ramos et al. [14] propose to monitor page accesses in the memory controller and migrate hot pages from NVM to DRAM. A few studies [15], [17] exploit OS-level page access counting mechanisms for page migrations. These approaches need to intentionally disable the hardware TLB, so that OSes can track the TLB misses to estimate the number of page accesses. The above *dynamic memory migration* polices cause significant hardware or software overhead, and are not applicable for hot object detection in K-V stores. HMCached develops a lightweight application-level object access counting mechanism, and design an NVM-friendly K-V index structure to mitigate the performance overhead of object access counting.

Memory Reassignment for In-Memory K-V Stores. There have been some proposals to solve the slab calcification problem in the DRAM-based Memcached. Memcached monitors the number of object evictions for each slab class in a interval of 10 seconds. It selects a slab class without object evictions in the recent three intervals, and reassign one slab to another slab class that shows the most object evictions. However, in practise, it is hard to find a slab class without object evictions within 30 seconds for real workloads. Carra et al. propose *periodic slab allocation* [30], which uses cache miss rate of slab classes as an indication for memory reassignment. Hu et al. [18] exploit cache miss ratio curves to determine the optimal memory size required by each slab class. Ou et al. [19] take the miss penalties of different slab classes into account for memory reassignment. All those schemes all use LRU-based object replacement algorithm for each slab class, and memory reassignment are based on the miss rate/penalty of different slab classes. In contrast, HMCached propose an access-frequency and benefit-aware DRAM reassignment policy to best utilized the scarce DRAM resource in hybrid memory system.

Persistent Memory Supported K-V Stores. Recently, there have been a few studies on data persistence in NVM-based K-V stores. HiKV [38] is a hybrid K-V index structure for hybrid memory systems. The hybrid indexes includes a hash index in the NVM to guarantee persistence of SET/GET/DELETE operations, and a B^+-Tree index in the DRAM to support fast range scan operations. Bullet [39] is an in-memory K-V store that leverages cross-referencing logs to mitigate the cost of persistent memory updates. *NVMcached* [40] provides consistency-friendly data structures for persistent memory supported K-V stores. NVHT [41] is a persistent memory enabled K-V store programming library with a log-based consistency mechanism supported. These studies are orthogonal and complementary to our work. HMCached focuses on application-level K-V object migration and NVM-friendly index designs to effectively exploit the different characteristics of NVM and DRAM, and leaves databases to handle data persistence problems.

6 Conclusions

In this paper, we present *HMCached*, an in-memory K-V store built on a hybrid DRAM/NVM system. HMCached utilizes an application-level data access counting mechanism to identify frequently-accessed objects in NVM, and migrates them to fast DRAM to reduce the costly NVM accesses. We propose an NVM-friendly index structure to store the frequently-updated portion of object metadata in the DRAM, and thus further mitigate the NVM accesses. In addition, we propose a benefit-aware memory reassignment policy to address the *slab calcification* problem for in-memory K-V stores, and significantly improve the benefit gain from the DRAM. We implement HMCached based on Memcached and evaluate it with Zipfian-like workloads. Experiment results show that HMCached significantly reduces NVM accesses by 70 percent and improves application performance by up to 50 percent compared to the vanilla Memcached. Furthermore, compared to a DRAM-only system, HMCached significantly reduces the DRAM usage by 75 percent, while approximating 90 percent of its performance and 54 percent of energy consumption for realistic workloads.

Acknowledgments

The authors would like to thank the anonymous reviewers for their insightful comments. This work is supported jointly by National Key Research and Development Program of China under grant No.2017YFB1001603, and National Natural Science Foundation of China (NSFC) under grants No.61672251, 61732010, 61825202.

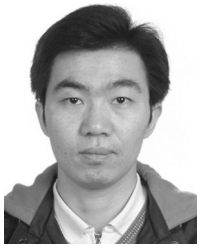
References

[1] "Memcached: A distributed memory object caching system," 2011. [Online]. Available: https://memcached.org/
[2] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling memcache at facebook," in *Proc. USENIX Symp. Networked Syst. Des. Implementation*, Apr. 2013, pp. 385–398.
[3] "Amazon ElastiCache," 2011. [Online]. Available: http://aws.amazon.com/elasticache/
[4] G. H. Loh, "3D-Stacked memory architectures for multi-core processors," in *Proc. Int. Symp. Comput. Architecture*, Jun 2008, pp. 453–464.
[5] S.-H. Lee, "Technology scaling challenges and opportunities of memory devices," in *Proc. IEEE Int. Electron Devices Meeting*, Dec. 2016, pp. 1.1.1–1.1.8.
[6] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T. W. Keller, "Energy management for commercial servers," *Comput.*, vol. 36, no. 12, pp. 39–48, Dec. 2003.
[7] J. Li and C. Lam, "Phase change memory," *Sci. China Inf. Sci.*, vol. 54, no. 5, pp. 1061–1072, May 2011.
[8] B. Hudec, C.-W. Hsu, I.-T. Wang, W.-L. Lai, C.-C. Chang, T. Wang, K. Fröhlich, C.-H. Ho, C.-H. Lin, and T.-H. Hou, "3D resistive RAM cell design for high-density storage class memory—A review," *Sci. China Inf. Sci.*, vol. 59, no. 6, pp. 061 403:1–061 403:21, Mar. 2016.

[9] "3DXPoint Technology," 2015. [Online]. Available: http://www.intel.com/content/www/us/en/architecture-andtechnology/3d-xpoint-technology-animation.html
[10] S. R. Dulloor, A. Roy, Z. Zhao, N. Sundaram, N. Satish, R. Sankaran, J. Jackson, and K. Schwan, "Data tiering in heterogeneous memory systems," in *Proc. Eur. Conf. Comput. Syst.*, Apr. 2016, pp. 15:1–15:16.
[11] W. Wei, D. Jiang, S. A. McKee, J. Xiong, and M. Chen, "Exploiting program semantics to place data in hybrid memory," in *Proc. Int. Conf. Parallel Architectures Compilation Techn.*, Oct. 2015, pp. 163–173.
[12] K. Wu, Y. Huang, and D. Li, "Unimem: Runtime data management on non-volatile memory-based heterogeneous main memory," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2017, pp. 58:1–58:14.
[13] K. Wu, J. Ren, and D. Li, "Runtime data management on non-volatile memory-based heterogeneous memory for task-parallel programs," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage, Anal.*, Nov. 2018, pp. 31:1–31:13.
[14] L. Ramos, E. Gorbatov, and R. Bianchini, "Page placement in hybrid memory systems," in *Proc. Int. Conf. Supercomputing*, May 2011, pp. 85–95.
[15] N. Agarwal and T. F. Wenisch, "Thermostat: Application-transparent page management for two-tiered main memory," in *Proc. Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Apr. 2017, pp. 631–644.
[16] H. Liu, Y. Chen, X. Liao, H. Jin, B. He, L. Zheng, and R. Guo, "Hardware/software cooperative caching for hybrid dram/nvm memory architectures," in *Proc. Int. Conf. Supercomputing*, Jun. 2017, pp. 26:1–26:10.
[17] S. Kannan, A. Gavrilovska, V. Gupta, and K. Schwan, "Heteroos: Os design for heterogeneous memory management in datacenter," in *Proc. Annu. Int. Symp. Comput. Architecture*, Jun. 2017, pp. 521–534.
[18] X. Hu, X. Wang, Y. Li, L. Zhou, Y. Luo, C. Ding, S. Jiang, and Z. Wang, "Lama: Optimized locality-aware memory allocation for key-value cache," in *Proc. USENIX Annu. Tech. Conf.*, 2015, pp. 57–69.
[19] J. Ou, M. Patton, M. D. Moore, Y. Xu, and S. Jiang, "A penalty aware memory allocation scheme for key-value cache," in *Proc. Int. Conf. Parallel Process.*, Sep. 2015, pp. 530–539.
[20] "HMCached," 2019. [Online]. Available: https://github.com/CGCL-codes/HMCached
[21] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *Proc. ACM SIGMETRICS/PERFORMANCE Joint Int. Conf. Meas. Model. Comput. Syst.*, 2012, pp. 53–64.
[22] W. Zhang, J. Hwang, T. Wood, K. K. Ramakrishnan, and H. H. Huang, "Load balancing of heterogeneous workloads in memcached clusters," in *Proc. Int. Workshop Feedback Comput.*, Jun. 2014.
[23] 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 *Proc. Eur. Conf. Comput. Syst.*, Apr. 2016, pp. 14:1–14:15.
[24] B. Chandramouli, G. Prasaad, D. Kossmann, J. Levandoski, J. Hunter, and M. Barnett, "Faster: A concurrent key-value store with in-place updates," in *Proc. Int. Conf. Manage. Data*, Jun. 2018, pp. 275–290.
[25] N. Megiddo and D. S. Modha, "Outperforming lru with an adaptive replacement cache algorithm," *Comput.*, vol. 37, no. 4, pp. 58–65, 2004.
[26] F. J. Corbato, "A paging experiment with the multics system," *Mit Project Mac Rep. MAC-M-384*, pp. 217–228, May 1968.
[27] B. Fan, D. G. Andersen, and M. Kaminsky, "Memc3: Compact and concurrent memcache with dumber caching and smarter hashing," in *Proc. USENIX Conf. Networked Syst. Des. Implementation*, Apr. 2013, pp. 371–384.
[28] H. Yoon, J. Meza, R. Ausavarungnirun, R. A. Harding, and O. Mutlu, "Row buffer locality aware caching policies for hybrid memories," in *Proc. IEEE Int. Conf. Comput. Des.*, Sep. 2012, pp. 337–344.
[29] Y. Zhou, J. Philbin, and K. Li, "The multi-queue replacement algorithm for second level buffer caches," in *Proc. USENIX Annu. Tech. Conf.*, Jun. 2001, pp. 91–104.
[30] D. Carra and P. Michiardi, "Memory partitioning in memcached: An experimental performance analysis," in *Proc. IEEE Int. Conf. Commun.*, Jun. 2014, pp. 1154–1159.
[31] "HME: A Lightweight Emulator for Hybrid Memory," 2018. [Online]. Available: https://github.com/CGCL-codes/HME
[32] Z. Duan, H. Liu, X. Liao, and H. Jin, "Hme: A lightweight emulator for hybrid memory," in *Proc. Des., Automat. Test Eur. Conf. Exhib.*, Mar. 2018, pp. 1375–1380.
[33] "Intel PMU profiling tools," 2015. [Online]. Available: https://github.com/andikleen/pmu-tools
[34] Y. Zhang, J. Yang, A. Memaripour, and S. Swanson, "Mojim: A reliable and highly-available non-volatile memory system," in *Proc. Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Mar. 2015, pp. 3–18.
[35] A. Blankstein, S. Sen, and M. J. Freedman, "Hyperbolic caching: Flexible caching for web applications," in *Proc. USENIX Annu. Tech. Conf.*, Jul. 2017, pp. 499–511.
[36] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. ACM Symp. Cloud Comput.*, Jun. 2010, pp. 143–154.
[37] "Processor Counter Monitor," 2017. [Online]. Available: https://github.com/opcm/pcm
[38] F. Xia, D. Jiang, J. Xiong, and N. Sun, "Hikv: A hybrid index key-value store for dram-nvm memory systems," in *Proc. USENIX Annu. Tech. Conf.*, Jul. 2017, pp. 349–362.
[39] Y. Huang, M. Pavlovic, V. Marathe, M. Seltzer, T. Harris, and S. Byan, "Closing the performance gap between volatile and persistent key-value stores using cross-referencing logs," in *Proc. USENIX Annu. Tech. Conf.*, Jul. 2018, pp. 967–979.
[40] X. Wu, F. Ni, L. Zhang, Y. Wang, Y. Ren, M. Hack, Z. Shao, and S. Jiang, "NVMcached: An NVM-based Key-Value Cache," in *Proc. ACM SIGOPS Asia-Pacific Workshop Syst.*, Aug. 2016, pp. 1–7.
[41] K. Huang, J. Zhou, L. Huang, and Y. Shen, "NVHT: An efficient key value storage library for non-volatile memory," *J. Parallel Distrib. Comput.*, vol. 120, pp. 339–354, 2018.

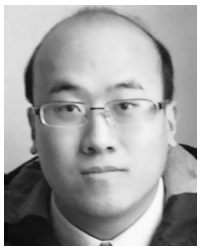
Hai Jin received the PhD degree in computer engineering from HUST, in 1994. He is a Cheung Kung Scholars chair professor of computer science and engineering with Huazhong University of Science and Technology (HUST), in China. In 1996, he was awarded a German Academic Exchange Service fellowship to visit the Technical University of Chemnitz in Germany. He worked with The University of Hong Kong between 1998 and 2000, and as a visiting scholar with the University of Southern California between 1999 and 2000. He was awarded Excellent Youth Award from the National Science Foundation of China in 2001. He is the chief scientist of ChinaGrid, the largest grid computing project in China, and the chief scientists of National 973 Basic Research Program Project of Virtualization Technology of Computing System, and Cloud Security. He has co-authored 22 books and published more than 800 research papers. His research interests include computer architecture, virtualization technology, cluster computing and cloud computing, peer-to-peer computing, network storage, and network security. He is a fellow of the CCF, IEEE and a member of the ACM.

Zhiwei Li received the BS degree from the School of Computer Science and Engineering, University of Electronic Science and Technology of China, China, in 2016. He is currently working toward the MS degree in the School of Computer Science and Technology, Huazhong University of Science and Technology, China. His research interests mainly include non-volatile memory and database.



Haikun Liu received the PhD degree from Huazhong University of Science and Technology, China. He is an associate professor with the School of Computer Science and Technology, HUST. His current research interests include in-memory computing, virtualization technologies, cloud computing. He was the recipient of outstanding doctoral dissertation award in Hubei province, China. He is senior member of the CCF, and a member of the IEEE.

Xiaofei Liao received the PhD degree in computer science and engineering from Huazhong University of Science and Technology (HUST), China, in 2005. He is now a professor with the School of Computer Science and Engineering, HUST. He has served as a reviewer for many conferences and journal papers. His research interests include in the areas of system software, P2P system, cluster computing and streaming services. He is a member of the IEEE and the IEEE Computer Society.



Yu Zhang received the PhD degree in computer science from Huazhong University of Science and Technology, in 2016. He is now an associate professor with the School of Computer Science, HUST. His research interests include computer architecture, system software, runtime optimization, programming model, and big data processing. His current topic mainly focuses on domain-specific architecture, runtime system, and programming model for graph processing. He is a member of the CCF, IEEE, ACM, and USENIX.