Abstract In-memory key-value stores are attractive to contemporary applications due to their exceptional performance. However, their scalability and applicability are limited by the high cost, limited capacity, and volatility of DRAM. To address these challenges, emerging byte-addressable storage technologies, such as NVM, provide an affordable and high-capacity alternative with performance comparable to DRAM. This paper delves into the utilization of heterogeneous memory for effective and efficient KV stores. We propose HeterMM, a general and efficient framework designed for applying in-DRAM indexes to heterogeneous memory-based key-value stores. It supports plugged-in indexes in order to leverage kinds of mature and well-designed in-DRAM indexes. It employs a unified data storage framework across different storage devices, harnessing the larger capacity of NVM and superior performance of DRAM simultaneously. Our experiments, conducted on the YCSB benchmark, demonstrate the superiority of HeterMM. With the same plugged-in index, HeterMM-based systems achieve up to 7.37x better performance compared to systems based on the state-of-the-art framework. This highlights the effectiveness and efficiency of HeterMM in harnessing the potential of byte-addressable storage in key-value stores.

Keywords key-value stores, byte-addressable storage, NVM, volatile indexes

1 Introduction

As representative NoSQL systems, Key-Value (KV) stores have become crucial to contemporary application systems [1,2]. In-memory storage solutions, such as Redis [3], are particularly appealing due to their exceptional performance. However, their scalability is hindered by the high cost, limited capacity, and volatility of DRAM, which further influences their scope of application. Particularly, the growing disparity between rapidly expanding data sizes and the more gradual growth of DRAM capacity has lessened the appeal of in-memory systems for general applications [4,5]. Emerging byte-addressable storage (BAS) devices [6], such as the Non-volatile Memory (NVM) technology [7], are designed to overcome the scaling constraints of the DRAM and offer affordability, enhanced capacity, and performance comparable to DRAM. This paper delves into utilizing NVM for creating effective and efficient KV stores. In order to maximize the system performance, it is important to consider the specific characteristics of NVM devices, especially compared to DRAM. In this paper, we adopted the typical performance assumption in previous works [8], which are also exhibited in existing NVM devices [9, 10].

- Read-write asymmetry in terms of latency and bandwidth, potentially leading to write operations becoming a performance bottleneck.
- Poor random access performance compared to sequential accesses, highlighting the importance of data locality.

Therefore, replacing DRAM with NVM directly is not an effective-enough solution [11]. Instead, the heterogeneous memory architecture combining both emerges as a promising alternative [12, 13].

Numerous efforts have been dedicated to redesigning conventional structures on NVM [14,15]. However, these initiatives either did not consider a heterogeneous memory architecture or were challenged by the substantial engineering cost and increased complexity of integrating into existing systems [16]. As a result, there is an increased interest in a general framework to apply existing indexes to KV

stores on NVM, due to its simplicity and the potential to utilize extensive DRAM index research for KV store implementation.

Several studies have proposed general frameworks for converting volatile indexes persist using NVM [16, 17]. However, none of them have fully utilized the potential of a heterogeneous memory architecture while persisting both the index and the data in NVM. In particular, TIPS [16] stands out by employing an in-DRAM hash table as a cache, improving the performance greatly. However, our experimental results reveal a significant performance gap between a purely in-DRAM KV store and the one incorporating TIPS. This is mainly because it does not consider the difference in performance characteristics between NVM and DRAM and directly applying in-DRAM indexes to NVM would result in inefficient utilization of NVM.

In this paper, we propose a general framework named HeterMM, designed to apply in-DRAM indexes to heterogeneous memory-based KV stores. It is designed to fully leverage the superior performance of DRAM, and make the performance of the system as close to the in-DRAM one as possible.

To summarize, HeterMM aims to achieve three primary objectives: (1) providing ease of application for different types of indexes, (2) optimizing the utilization of the superior performance of DRAM and specific characteristics of NVM to deliver high performance, and (3) fully leveraging the persistence of NVM to address the volatility of in-DRAM KV stores with minimal overhead. The specific contributions of this work are as follows:

1. Designing HeterMM, a versatile framework that applies mature and well-designed volatile indexes to heterogeneous memory-based KV stores while reserving their strengths. Utilizing an operation log mechanism to ensure persistence and facilitate failure recovery.
2. Dedicated to achieving performance as close to that of in-DRAM systems as possible by maintaining the index and hot data in DRAM. Particularly, it employs a hotness-aware storage mechanism for heterogeneous memory, including an adaptable read cache in DRAM, an append-only storage mechanism on NVM, and a thread-oriented allocation mechanism.
3. Combing HeterMM with different types of indexes (including Hash and B+ tree) and evaluating their performance using the YCSB benchmark [18]. The evaluation demonstrates that HeterMM outperforms leading conversion frameworks and NVM-optimized designs across various workloads.

The rest of this paper is organized as follows: In Section 2, we discuss the related work. Section 3 presents the system architecture and Section 4 provides an overall introduction to the heterogenous memory management. Then, details of optimizing and implementation are presented in Section 5. The results of the experimental study are reported in Section 6. Finally, there is a discussion about the future of byte-addressable devices in Section 7 and a conclusion in Section 8.

2 Related Work

2.1 Specified KV Stores on NVM

Byte-addressable devices have been attractive due to their ability to address the shortcomings of DRAM, including the high cost and limited capacity [19]. NVM technology provides a specific type of byte-addressable device with the ability to persist. Over the recent decade, numerous studies have focused on optimizing KV systems on NVM, with many attempting to redesign traditional structures [14, 15,

20] or design new kinds of indexes [21]. Most of them relied on simulations or were optimized for small values, generally preferring to place everything into NVM without considering a heterogeneous memory architecture.

Certain studies have partially utilized DRAM to hold (part of) the index [22–25] or work as a cache [26, 27]. SwapKV [28], in particular, provides a key-value store for hybrid memory systems by bringing hot data into DRAM while writing cold data into NVM, However, they introduce additional engineering costs due to their specific index designs. HeterMM differs from them by providing a more general and plugged-in framework that can be applied to different kinds of indexes.

Besides, FlatStore [8] employs an in-DRAM index and a horizontal batching technology to optimize small key-value pairs by combining small entries and avoiding small writes on NVM. In this paper, we go further by designing an efficient storage mechanism for heterogeneous memory and holding the hot data in DRAM to make system performance as close to the in-DRAM one as possible.

2.2 KV stores on hybrid storage systems

Besides the NVM, there have been various studies on KV stores across hybrid storage systems consisting of DRAM and disks or SSDs. The typical design is FASTER [29], which also consists of an in-memory index and a unified hybrid storage across the DRAM and the disk. However, on the one hand, it is targeted at disks or SSDs, which shows very different characteristics of NVM. For example, FASTER does not support reading from the disk directly and incurs asynchronous reading instead to hinder the extremely high overhead of disk reading. On the other hand, as will be discussed later, HeterMM divides the DRAM space

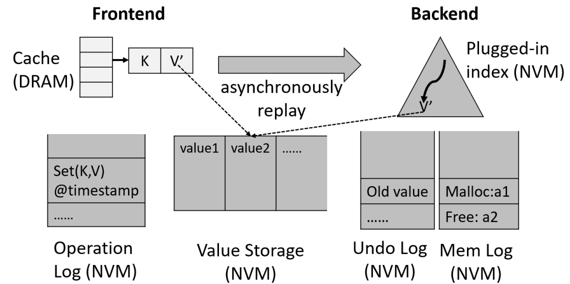


Fig. 1 The framework of TIPS.

into the write region and the read cache to further optimize the read operations, which is not supported by FASTER.

2.3 Techniques to convert volatile indexes persist

Several studies have proposed techniques to transition volatile indexes to NVM [16, 17, 30, 31], which could be utilized to apply existing indexes in NVM-based KV stores. However, they generally encounter the following limitations:

1) They require in-depth index-specific knowledge [17, 31], which increases engineering costs and introduces a propensity for errors.

2) They impose restrictions on the applicable indexes [17, 30, 31], limiting their applicability.

3) They all inadequately leverage the superior performance of DRAM and the unique characteristics of NVM, resulting in significant overheads.

In particular, TIPS [16] is the state-of-the-art converting framework, which provides tiered concurrency and crash consistency. As depicted in Fig. 1, TIPS applies an in-DRAM hash table as an index cache and an operation log for durability. Newly arrived data are inserted into the cached index in DRAM and replayed to the NVM index asynchronously utilizing the operation log. Once the operation log is replayed, the corresponding entry in the cache is allowed to be removed. For crash consistency, it applies an undo log to recover the index from

an unfinished modification. To avoid the memory leak in NVM, it applies a memory allocation log to record all allocated and freed addresses.

Needless to say, the in-DRAM index cache in TIPS helps it achieve a better performance compared to pure in-NVM ones. However, it still suffers from the following problems. Firstly, its in-DRAM hash table only works as an index with all its data restored in NVM, leading to at least one NVM operation for each read or write request. Secondly, it directly applies in-DRAM indexes to NVM without considering the difference between the two devices, resulting in lots of small and random access to NVM and inefficient replaying operations, which could further influence the throughput of the whole system. Thirdly, the extra undo log and memory allocation log in NVM would result in the write amplification on NVM, which increases the competition on the limited write throughput of NVM [32]. Besides, the DRAM cache only holds newly written data, which shows no effect on read-only data. In contrast, HeterMM is designed specifically to address these issues, with a primary focus on the effective and efficient management of data across DRAM and NVM.

3 System Design

3.1 Overall Architecture

The primary concern of a system on a heterogeneous memory architecture is the effective utilization of characteristics of different devices.

A typical KV store comprises an index to facilitate search operations and a data storage mechanism to hold the data. While some systems tightly integrate these components, the access characteristics of the index and data differ, especially regarding sequential and random access patterns. Random data writes can be mitigated by implementing an append-only store, and index updates can be avoided by in-place updates. However, preventing random writes on the index for inserting or deleting is often challenging, as they are dictated by the index design. As a result, hot random writes for data and indexes potentially reside in different areas.

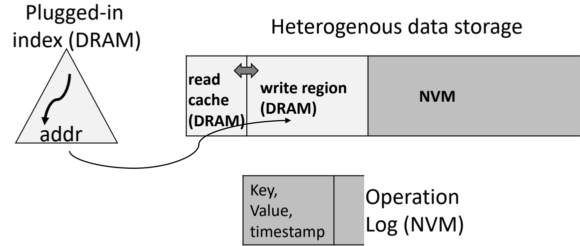


Fig. 2 The framework of HeterMM.

Based on the above observations and considering the performance diversity between sequential read and random operations of NVM, we prefer decoupling and managing the index and data separately within a KV store on heterogeneous memory architecture. As illustrated in Fig. 2, HeterMM is composed of a plugged-in in-DRAM index, a hotness-aware data storage mechanism on heterogeneous memory, and an operation log on NVM.

In particular, the index, which is the most frequently accessed and typically in a small unit and random order, is not friendly to NVM [32]. Moreover, data structures of the index optimized for DRAM may not perform as effectively on NVM due to their different characteristics. Therefore, we advocate for maintaining the index in DRAM to achieve system performance near reaching that of in-DRAM systems. This assumption that the index can fit into DRAM is widely endorsed by prior studies [8, 22]. Previous reports [33, 34] also show that value sizes are usually much larger than the key size. Specifically, the recent study from Twitter [34] reports that around 50% of workloads in their cluster have

```
Value_t* get_value(HeterMemory* hm, LogAddress addr);
LogAddress set_value(HeterMemory* hm, LogAddress addr,
    Key_t key, Value_t* newValue);
```

Fig. 3 APIs provided by HeterMM.

value sizes over 5x key sizes, and over 30% of the workloads have value sizes greater than 10x key sizes. Although NVM offers a larger capacity than DRAM, the difference is within tenfold [35]. Therefore, we argue that maintaining the index in DRAM is reasonable.

Specifically, newly written data resides in DRAM and each data is allocated a logical address to calculate its physical position. Old data is flushed to NVM in batches while the logical address remains the same during and after the data is flushed to NVM. The durability of data residing in DRAM is ensured by the operation log. On the one hand, data in DRAM is updated in place, which could be regarded as early compaction and reduces data volume flushed to NVM. On the other hand, data in NVM could be regarded as a checkpoint which could be used to cut off the operation log. Moreover, to optimize access to read-only data in NVM, we further divide the DRAM region into a read cache and a write region, which could be resized dynamically according to the workload.

3.2 Plug-In Programming Model

In order to combine with different kinds of indexes conveniently, HeterMM offers a set of plug-in APIs, as depicted in Fig. 3. They are used to encapsulate the value get/set methods in the original index.

Specifically, the *set_value* API is responsible for storing the data in memory (DRAM or NVM) and returning a logical address to the caller. The logical address works as the value in the original volatile index and is used to call the *get_value* API to get the actual position of the data during a *get* request. Fig. 4 illustrates an example of how to utilize HeterMM APIs to apply a pure in-DRAM hash table to heterogeneous memory stores. The only changes required are the replacement of the value get/set codes with the provided APIs.

```
int hash_set(Hash* h, Key_t key, Value_t value){
    HashBucket* bucket = get_bkt(h, key);
    int idx;
    pthread_rwlock_wrlock(&bucket->lock);
    bucket[idx].key = key;
#ifndef USE_HeterMM //The original code
    bucket[idx].value = value;
#else // Applying HeterMM
    bucket[idx].value =
     set_value(h->heterMemory, NULL, key, &value);
#endif
    pthread_rwlock_unlock(&bucket->lock);
}

Value_t* hash_get(Hash* h, Key_t key){
    HashBucket* bucket = get_bkt(h, key);
    int idx;
    Value_t* ret = NULL;
    pthread_rwlock_rdlock(&bucket->lock);
    if(bucket[idx].key == key){
#ifndef USE_HeterMM //The original code
        ret = &bucket[idx].value;
#else // Applying HeterMM
        ret = get_value(h->heterMemory,
     bucket[idx].value);
#endif
    }
    pthread_rwlock_unlock(&bucket->lock);
    return ret;
}
```

Fig. 4 Simplified code snippet of the *set* and *get* functions when utilizing HeterMM on a hash table.

4 Heterogeneous Memory Management

In this section, we provide an overview of HeterMM, including its memory management mechanism (in Section 4.1) and how the get/set APIs are implemented (in Section 4.2 and 4.3).

4.1 Memory Management

As Fig. 5 shows, HeterMM aims to retain hot data in DRAM and flush cold data into NVM. NVM and DRAM are managed as a unified flat-addressable

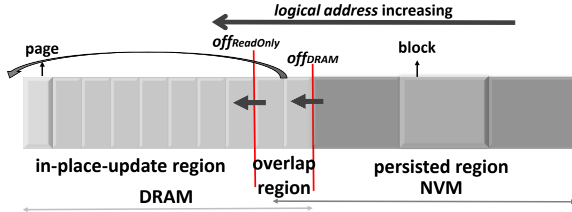


Fig. 5 The logical space unifying DRAM and NVM.

space but with separate management strategies to fit their distinct characteristics.

As highlighted in previous studies [11,36], NVM often suffers from the asymmetry of read/write operations and the disparity between sequential and random accesses. The limited write bandwidth of NVM in high concurrency conditions also significantly impacts the performance. Acknowledging these, HeterMM adopts append-only writing for the NVM and decouples writes on NVM from the front-end accesses to evade high concurrency.

Specifically, the whole data space is divided into three parts, applying different updating strategies:

1. The **in-place-update region**, where data resides in DRAM and can be updated in place.
2. The **overlap region**, where data is flushing into NVM, may reside in both DRAM and NVM and must be updated out-of-place.
3. The **persisted region**, where data resides in NVM and must be updated out-of-place.

Newly arrived or out-of-place updated data are appended to the in-place-update region while old data will move on to the overlap region or persisted region asynchronously and sequentially. Specifically, newly appended data is assigned a unique and increasing logical address, which remains unchanged unless the data is updated out-of-place. Two markers, $off_{ReadOnly}$ and off_{DRAM}, are used to distinguish the three storage regions. As shown in Fig. 5, data whose logical address greater than $off_{ReadOnly}$ resides in in-place-update region; those whose logical address between $off_{ReadOnly}$ and off_{DRAM} reside in overlap region; and others reside in persisted region.

Data in all three regions could be read directly and data in the overlap region is read from DRAM. In particular, the overlap region works as the middle tier to distinguish hot and cold data by providing hot data a chance to keep residing in the DRAM. This is because the old but hot data in it is highly likely to be brought back to the in-place-update region before it reaches the persisted region. Therefore, all accesses to the hot data can be served by the DRAM.

4.1.1 Memory Organization

HeterMM manages DRAM in pages, following traditional memory management strategies [37]. The DRAM space is divided into pages of size 2^k bytes. All these pages are organized into a ring. If the free space in DRAM falls below a threshold, *freePages*, old data in DRAM is flushed to NVM. The pages from which data has been flushed are then freed and put to the end of the ring, ready for new data.

NVM is organized into blocks of size 2^m bytes, where $m > k$[1)]. The space allocation and reclamation unit in NVM is a block. The data in NVM is accessible only if the whole block finishes the persisting, and its logical address remains the same when and after the data is flushing to NVM, without needing to modify the index. In HeterMM, each NVM block corresponds to an NVM file. When a new block is needed, a new file in NVM is created and mapped. To avoid file creation becoming a bottleneck and hindering the flushing processes, some files will be created in advance when the flushing threads are idle.

[1)] Empirically, $k = 20$ is an effective setting for page sizes and $m = 30$ is an effective setting for block sizes.

4.1.2 Data Flushing

In the background, flushing operations are performed to transfer old data from DRAM to NVM once the free space in DRAM drops below *freePages*[2)]. The flushing process consists of two stages: the persisting stage and the freeing stage.

During the persisting stage, old pages in DRAM are flushed and persisted in NVM. The $off_{ReadOnly}$ is increased just ahead of the overlap stage, indicating that data with a logical address less than $off_{ReadOnly}$ is ready for being flushed and persisted. Specifically, data in the overlap region, located between off_{DRAM} and $off_{ReadOnly}$, may exist in both devices. Therefore, data in the overlap region is read-only to avoid inconsistency between the two copies.

In the freeing stage, DRAM pages that have completed persisting are freed. The freed page is then moved to the end of the ring of DRAM pages, ready to accommodate incoming data. Before freeing a page, the off_{DRAM} marker is advanced to notify workers about changes in the physical data position.

4.2 Get Operation

When the *get_value* API is invoked, HeterMM retrieves the data based on the provided logical address. It first checks the off_{DRAM} to determine whether the data is located in DRAM or NVM. Owing to the byte-addressability of NVM, data in NVM can be read in the same way as data in DRAM. Specifically, the physical address of the data can be calculated based on the logical address *addr* as below:

$$
\begin{aligned}
physicalAddress &= \begin{cases} pages[P_{no}] + (addr \bmod (2^k - 1)) \\ \qquad if \quad addr \geq off_{DRAM} \\ blocks[B_{no}] + (addr \bmod (2^m - 1)) \\ \qquad if \quad addr < off_{DRAM} \end{cases} \\
P_{no} &= \frac{addr}{2^k} \quad \bmod pageNum \\
B_{no} &= \frac{addr}{2^m}.
\end{aligned} \tag{1}
$$

The physical address comprises the base address and the inner offset within a page or block. In this equation, *pages* and *blocks* represent arrays holding base addresses of DRAM pages and NVM blocks.

Notably, as DRAM pages may be recycled, the mapping between pages and logical addresses can change as new data arrives. However, before a page is freed and recycled, its corresponding logical address remains constant. In the equation, *pageNum* is the total number of pages in DRAM and also defines the size of the *pages* array. When a page is allocated, its corresponding range of logical addresses is determined, and then its base address is recorded in the *pages* array. Therefore, a logical address will always map to a definite slot (defined by P_{no} in the equation) in the *pages* array unless the $off_{ReadOnly}$ grows to greater than it.

4.3 Set Operation

The *set_value* API is responsible for setting new data or updating existing data with a logical address *addr*. Typically, new data shall provide an invalid logical address (indicated by *NULL* in Fig. 4) and HeterMM will allocate a new logical address for it. As illustrated in Fig. 5, DRAM and NVM are organized as a unified logical address space, with the logical address increasing along with the space allocation. When a new logical address is required, HeterMM maps it with the corresponding space in

[2)] Empirically, $freePages = 2^{m-k}$ is an effective setting.

the DRAM. Essentially, there is an active page in DRAM responsible for housing new data. If there is insufficient space on the active page, HeterMM applies for a free page which then becomes active.

In cases where the caller provides a valid logical address in the *addr*, it indicates that an update operation is taking place. Specifically, if the old data is located in the persisted or overlap regions, it is updated out-of-place. In this case, HeterMM allocates a new logical address, similar to an insert operation. If the old data resides in the in-place-update region, i.e., the original logical address is after the $off_{ReadOnly}$, the data is updated directly in place and the original logical address is returned.

5 Implementation

The plug-and-play APIs of HeterMM ensure its ease of use. This section introduces the details of its implementation designs that contribute to its exceptional performance and reliability in case of failure.

Firstly, to mitigate the potential performance bottleneck of random read operations on NVM in read-intensive workloads, HeterMM employs an adaptive read cache (detailed in Section 5.1), which can dynamically adjust its size based on the workload.

Secondly, there is a space reclamation mechanism discussed in Section 5.2. Given the append-only writing on NVM, where data is immutable and updated out-of-place, it is crucial to prevent old or deleted data from depleting the NVM resources. To this end, HeterMM applies a space reclamation mechanism, which reorganizes old and sparse blocks and frees them.

Then, an optional operation logging mechanism is introduced in Section 5.3. This helps maintain system consistency in the event of a system failure.

Finally, a thread-oriented allocation mechanism is introduced to enhance the efficiency and scalability of the whole framework in Section 5.4.

5.1 Read Cache

As depicted in Fig. 2, HeterMM divides the DRAM space into two parts: the write region, serving the write requests as outlined before, and the read cache, storing frequently accessed data by read requests.

Firstly, the performance of NVM, unlike traditional secondary storage devices such as HDDs or SSDs, is closer to that of DRAM. As a result, the advantages of a read cache are easy to be diminished by the associated overheads of access tracking and cache replacement. To overcome this, HeterMM adopts a sampled FIFO (First-In-First-Out) strategy, which has been proven effective [37]. In this strategy, where a read request is selected for every *SampleFreq* read requests that access data stored in NVM[3)]. The data read by the selected request is then copied into the read cache in DRAM.

Secondly, the read cache shares the DRAM space with the write region, leading to a trade-off between the size of the read cache and the space available for write operations. A larger read cache benefits read operations by accommodating more frequently accessed data in DRAM. However, it comes at the cost of reducing the space available for write operations, resulting in more out-of-place updates and additional flushing operations to NVM.

To address the second challenge, HeterMM employs a dynamic DRAM allocation strategy. Initially, the entire DRAM is allocated to the write region for insertions and out-of-place updates. When the workload requires no additional space for these operations, HeterMM reallocates unused write region pages to the read cache. Conversely, if the

[3)] Empirically, *SampleFreq* = 100 is an effective setting.

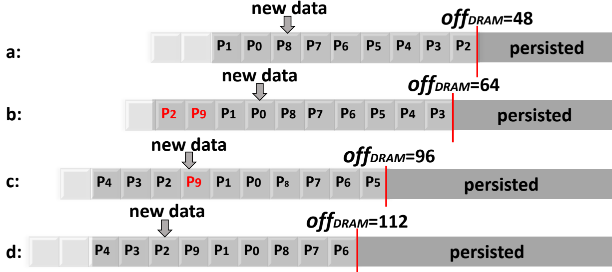


Fig. 6 Illustration of adding a new page into (**b** and **c**) and dropping a freed page (**d**) from the *pages* array, where $k = 4$.

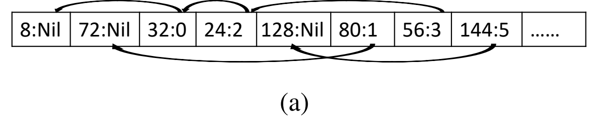
write region is short of spaces, HeterMM will retrieve pages from the read cache and reallocate them to the write region.

Specifically, to maintain the accuracy of Equation 1 when the write region size changes dynamically, the length of the *pages* array is set to the total number of DRAM pages, representing the maximum potential number of pages within the write region. Whenever a page is freed and becomes available for new allocations, HeterMM appends it to the end of the *pages* array.

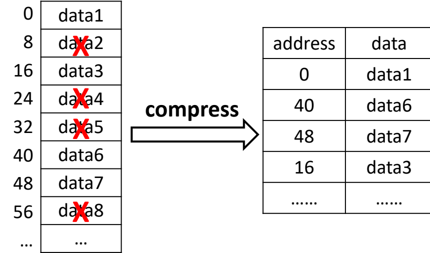
Consider a scenario where the page size is 2^4, as depicted in Fig. 6. Initially, the write region comprises 9 pages (**a**), while the *pages* array length is 11, indicating a total number of 11 pages in DRAM. Currently, the off_{DRAM} is 48 and new coming data is written to page P_8.

As new data comes, the system decides to absorb pages from the read cache into the write region. Therefore, before freeing page P_2, page P_9 is added to the write region. As a result, page P_9 is added to the end of *pages* array and after freeing page P_2, it is located after P_9. The resulted *pages* array is shown in Fig. 6 (**b**) where the off_{DRAM} increases to 64. After that, page P_9 becomes part of the ring buffer, just like the other pages (**c**).

Similarly, if a page is absorbed from the write region into the read cache, it is not placed at the end of the *pages* array (**d**). Instead, a similar but separate *pages* array, dedicated to the read cache, will be used to hold and manage it.



(a)



(b)

Fig. 7 Illustration of an invalid address array (**a**) and the process of compressing and reorganizing a block into a hash table (**b**), where the block size is 64B and data size is 8B.

Furthermore, when an update occurs on a cached tuple, an out-of-place update will be applied as the old resides in NVM. Therefore, the cached entry becomes invalid and waits to be freed and reclaimed as described above.

5.2 Space Reclamation

The append-only storage mechanism in NVM can result in a considerable amount of invalid data due to out-of-place updating and deletion. Therefore, HeterMM incorporates a space reclamation process to compress sparse blocks, which consists of two stages: the marking stage and the reorganizing stage.

In the marking stage, the invalid logical addresses due to out-of-place updating or deleting are recorded in an invalid address array. As shown in Fig. 7(a), addresses mapping to the same block are linked together. Meantime, each block keeps track of the total size of the invalid data within the block, which will be used later to determine whether the block needs to be reorganized. If the invalid space in an NVM block exceeds 50%, the daemon thread in-

vokes the reorganizing stage for it.

During the reorganizing stage, the invalid address array is accessed and the invalid data in the block is identified by the corresponding list. As illustrated in Fig. 7(b), the remaining valid data in the block is then reorganized and compressed. After finishing the reorganization, the hash table is flushed into a new file and replaces the old file.

The replacement is implemented by changing the base address in the *blocks* array, which is atomic. A bit in the base address is used to identify it is a compressed block. The old file will be freed after waiting a certain time[4] to ensure no worker is accessing it. When accessing a compressed block, the data is retrieved by querying the hash table.

Currently, HeterMM uses coalesced hashing [38] to implement the hash table, achieving 100% space utilization of the hash table. While compression introduces overhead for read operations, the read cache can alleviate this issue. Furthermore, it is plausible that data exhibits a locality in accessing time [29]. As data is arranged in the same order as it is allocated, it is likely that they will become invalid around the same time. Moreover, the compressed block generally contains older and less frequently accessed data, which makes the overhead of accessing the hash table acceptable.

5.3 Failure Recovery

Many key-value storage systems like Redis [3], typically employed as cache systems, can afford some data loss. However, failure recovery is critical in many other scenarios [16, 39]. The Write-Ahead-Log (WAL) is typically used for failure recovery but at the expense of substantial space and time overhead [29]. One common approach to mitigate these limitations is the use of checkpointing. In HeterMM, the use of NVM for WAL helps reduce the time overhead. Furthermore, HeterMM views the persisted region in NVM as a checkpoint, necessitating only operation logs for data stored in DRAM. We detail the log reclamation mechanism and the recovery process in the following.

5.3.1 Operation Log Reclamation

HeterMM manages operation logs in blocks. Each log block in HeterMM maintains a *maxAddress*, representing the highest logical address of the data recorded in the block. *maxAddress* serves to determine if all operation log entries within the block have been persisted and checkpointed. In other words, the corresponding data has entered the persisted region. Therefore, a block is deemed safe for reclamation and recycling when the off_{DRAM} surpasses its *maxAddress*.

A situation may arise where the off_{DRAM} does not advance in correspondence with extensive operation logs. It can occur when all updates fall within the in-place-update region because they create new logs without necessitating additional DRAM space. However, in this scenario, there should be many operation log entries for the same key. Therefore, they can be compressed as only the latest log entry is needed for recovery [8, 16].

5.3.2 System Recovery

In essence, the system comprises two elements: the in-DRAM index and data across heterogeneous memory. After a system event like a crash or shutdown, data within HeterMM reside in two places: the persisted region in NVM (cf. Fig. 5) and the operation logs, while data in the overlap region, which may partially persist in NVM, can be discarded as their

[4] 5 seconds in our setting, which far exceeds the request latency in our experiments.

operation logs are certainly not reclaimed. Furthermore, as multiple value versions of the same key may exist in the operation log, each key-value pair in HeterMM is assigned a timestamp to identify the latest version for a given key.

For failure recovery, all metadata of NVM blocks (including the data and the operation logs) resides in NVM and their modifications are atomic with the cache line flush (*clwb*) and memory fence instruction (*mfence*) [39]. Especially, to ensure the atomicity of operation log writing, it consists of two steps: (1) persisting the data while setting its timestamp as invalid; (2) setting and persisting the timestamp. Therefore, partially persisted data will not be considered during the recovery.

The recovery process, as shown in Algorithm 1, involves two stages: reinsertion of tuples (i.e., the key-value pair) from the operation log to the KV store (*Line 7 ∼ 12*) and reinsertion of logical addresses of tuples in the checkpoint (persisted region in NVM) into the in-DRAM index (*Line 13 ∼ 26*).

During operation log recovery, to address potential data loss caused by system crashes or shutdowns, the data that is not persisted must be reinserted into the system (*Line 12*). To identify whether a value should be inserted during recovery, a hash table (*ht* in Algorithm 1) in DRAM maintains the latest timestamp (*ts*) for each key. Data is inserted into the KV store only if the log entry timestamp exceeds the existing one. When an older log entry is accessed first, followed by a newer one, the older value is inserted into the KV store at first, and subsequently replaced by the newer value, by a regular update operation using the original *set* method.

During the recovery of the persisted region in NVM, HeterMM recalculates the logical address of each tuple (*Line 20*) and inserts it into the in-DRAM index to reconstruct the index. Therefore,

Algorithm 1: System recovery

```
   input : heterMemory: metadata of HeterMM;
   input : set: method to insert key-value pair;
   input : setAddr: method to insert key-address pair;
   input : index: metadata of the in-DRAM index;
 1 Function recovery():
 2     ht = {} ;
 3     if heterMemory.checkWAL() then
 4         recoverWAL()
 5     if heterMemory.checkPersist() then
 6         recoverNVM()
 7 Function recoverWAL():
 8     for log in heterMemory.wal.blocks do
 9         entry = ht.find(log.key);
10         if not entry or entry.ts < log.ts then
11             ht.set(log.key, log.ts, NIL);
12             set(index, log.key, log.value);
13 Function recoverNVM():
14     for block in heterMemory.nvmBlocks do
15         for tuple in block do
16             key = tuple.key;
17             ts =tuple.ts;
18             entry = ht.find(key);
19             if not entry or entry.ts < ts then
20                 newAddr = blockNo ×2^k + offset;
21                 if entry then
22                     addInvalid(entry.addr);
23                 ht.set(key, ts, newAddr);
24                 set(index, key, newAddr);
25             else
26                 addInvalid(entry.addr);
```

during this stage, HeterMM will not insert the value into the system inside the *set_value* API. Instead, it will return the given value, which is indeed the logical address (as indicated in *Line 24*), directly. Therefore, the logical address will be inserted into the in-DRAM index without re-storing data in the heterogeneous memory. Besides, unlike operation

Algorithm 2: Thread-oriented allocator

```
   input : size: the size to be allocated;
 1 Function allocate(size):
 2     if checkAlarm() then
 3         getNextPage();
 4     if curr.offset + size ≥ 2^k then
 5         getNextPage();
 6         return allocate(size);
 7     curr.offset += size;
 8     return curr.pageNo ×2^k + offset;
 9 Function getNextPage():
10     if offset < 2^k then
11         addr = curr.pageNo ×2^k + curr.offset;
12         addInvalid(addr);
13     pageNo = GlobalAllocated.fecthAdd(1);
14     curr.pageNo = pageNo;
15     curr.offset = 0;
```

log recovery, older and invalid data requires explicit marking to facilitate later space reclamation (*Line 22* and *26*).

Moreover, the algorithm can be easily transformed into a parallel version, enabling parallel processing of different log blocks and data blocks. The only required modifications involve adding appropriate locks on the hash table for concurrent accesses.

5.4 Thread-oriented Allocation

Multiple modules in HeterMM necessitate allocation, including allocation of a new logical address and the corresponding DRAM space for newly arrived data, allocation of a new cache entry, and allocation of an operation log entry. All these operations require the allocation of a globally unique position from the global pool. Although atomic operations like Compare-and-Swap (CAS) could implement this, they can be expensive and result in reduced scalability with multiple threads.

To enhance the efficiency of the allocation procedure, HeterMM employs a thread-oriented page-level allocator. Instead of assigning the space or the address from the global pool each time, a whole page is assigned to a thread at a time. Consequently, thread-local allocation replaces global allocation, significantly reducing the number of CAS operations. The allocation procedure is described in Algorithm 2. Specifically, each thread maintains a 64-bit variable named *curr*. Its high 32 bits represent the page occupied by the thread, while the lower 32 bits represent the offset inside the page to be allocated for the next allocation.

Another common feature of the allocation of logical addresses and cache entries is that they need to be reclaimed and reused cyclically. To avoid conflicts between the reclaim process and the workers, HeterMM regards the pages that have been assigned to a worker but have not finished the allocation as active pages. In contrast, inactive pages are those that have been allocated completely and the worker is working on another page. Then, HeterMM only allows inactive pages to be reclaimed.

However, as indicated in Fig.5, the pages are reclaimed in the logical address order, while the performance among different workers may vary. It is possible that a worker keeps a page active for an extremely long period due to specific workloads or other factors. This can potentially block the space reclaim process and impact other workers. To circumvent such circumstances, HeterMM applies an alarm clock mechanism. For each allocation type, a condition known as *checkAlarm* is defined. When the global space is close to being exhausted, the *checkAlarm* is evaluated. If the condition is met, the worker voluntarily relinquishes the current page and requests a new one, enabling the space reclamation process to proceed and preventing faster ones from being adversely affected.

Table 1 Workloads for evaluation.

Workload	Get	Set	Scan
Write-Heavy (YCSB-WH)	50%	50%	0%
Read-Heavy (YCSB-RH)	95%	5%	0%
Read-Only (YCSB-RO)	100%	0%	0%
Scan-Heavy (YCSB-SH)	0%	5%	95%

In the case of logical address allocation, Equation 2 defines its *checkAlarm* condition, which ensures that the currently used page is nearing the boundary of the in-place-update region as depicted in Fig. 5. For cache entry allocation, there is a variable named *dropPage* that represents the last reclaimed cache page, and the *checkAlarm* condition for cache entries is Equation 3, which signifies that the current page is expected to be reclaimed soon.

$$curr.P_{no} \times 2^k - off_{ReadOnly} \leq 2^k \tag{2}$$

$$curr.P_{no} - dropPage \leq 1 \tag{3}$$

In the context of relinquishing the current page and allocating new pages, there is a potential issue of wasted space on the old page. This issue has a lesser impact on cache entry. However, when allocating logical addresses, it can lead to a permanent space hole. To address this, in *Line 12* of the algorithm, the wasted spaces are identified and added to the invalid address array (as discussed in Section 5.2). During the re-organization process, these wasted spaces will be absorbed and eliminated.

6 Evaluation

To assess the performance of HeterMM, we conducted plenty of experiments to compare it with the state-of-the-art general frameworks and NVM-optimized KV stores.

6.1 Experimental Environment

Setup. We conducted experiments on a dual-socket machine running Linux kernel 5.4.0-126. Each socket was equipped with an Intel® Xeon® Gold 6326 CPU (2.90GHz) with 16 physical cores. Each physical core can be forked into 2 logical ones due to hyper-threading. The machine had 128GB DRAM and 1TB NVM (Intel® Optane™ Persistent Memory 200 Series) per socket. All NVM DIMMs were configured in App Direct mode, which exposes NVM and DRAM as two separate tiers of memory to programmers. To eliminate potential NUMA impacts, all experiments were performed on a single socket.

Workload. Our evaluation was based on the YCSB benchmark [18], which is a standard key-value store benchmark. The benchmark primarily consists of concurrent *get*, *set*, and *scan* requests. The request ratios for different evaluated workloads are shown in Table 1. The maximum scan length was set to 100, the same as in the TIPS [16] experiments. We configured the workload to follow the Zipfian distribution and adjusted the skewness by varying the Zipfian factor, z, where a higher z value indicated a more skewed workload.

Initially, we set z to 0.99, and each tuple consisted of an 8-byte key and a 64-byte value unless otherwise specified. We preloaded 200 million tuples into the system and measured the system throughput as our primary evaluation metric. All reported results are averages of 10 consecutive runs, each lasting 1 minute.

Systems for Comparison. To assess the efficiency and effectiveness of HeterMM, we combined it with two kinds of indexes: hash table and B+ tree. In particular, we combined HeterMM with a cache-line hash table (CLHT) [40], a lock-free hash table (LFHT) [41], and a B+ tree to implement hetero-

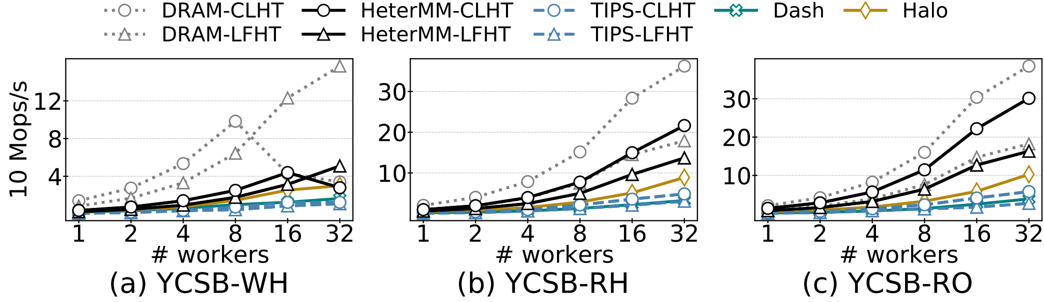


Fig. 8 Performance comparison among in-DRAM, HeterMM-based, TIPS-based, and NVM-optimized hash stores.

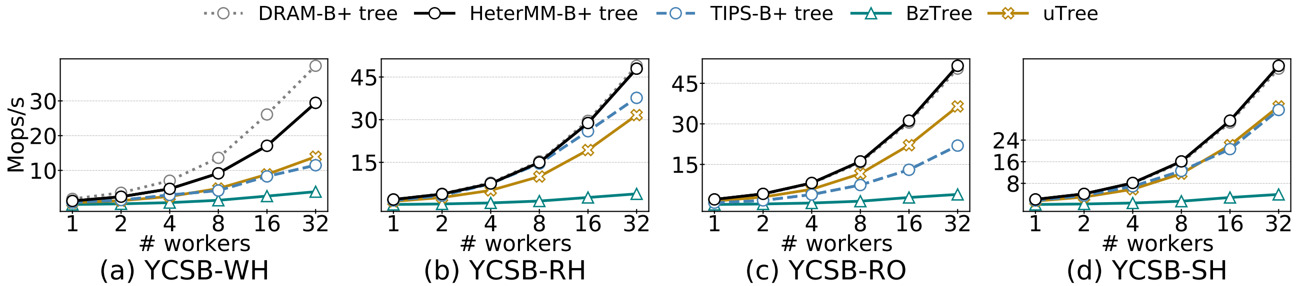


Fig. 9 Performance comparison among in-DRAM, HeterMM-based, TIPS-based, and NVM-optimized B+ tree stores.

geneous memory-based KV stores. We allocated 4GB of DRAM to HeterMM for its DRAM region, which includes the write region and the read cache, as depicted in Fig. 2. The allocated DRAM can accommodate approximately 25% of the dataset.

We compared HeterMM with two types of systems: general frameworks that apply existing indexes on NVM, and NVM-optimized KV stores. For the first type, we chose TIPS [16], which is a state-of-the-art general conversion framework that utilizes an in-DRAM hash table as the cache. We used TIPS to convert the same three indexes (CLHT, LFHT, and B+tree) into a persistent form, and its DRAM cache was configured to cache 100% of the index. For the second type of comparison systems, we chose Dash [42] and Halo [39] as representatives of systems that utilize hash tables, and BzTree [43] and μTree [44] as representatives using B+ trees. Among them, Halo and μTree are based on hybrid DRAM and NVM storage. In particular, Dash is designed as an index for 8 bytes values and we enhanced it with the PMDK's persistent allocator like the previous papers [25].

6.2 Performance Comparison and Analysis

To evaluate HeterMM, we assessed it alongside others using a variety of workloads, with varying worker numbers, workload skewness, and value sizes.

6.2.1 Varying Degree of Parallelism

Figures 8 and 9 illustrate the throughput of all comparison systems as the number of workers increases, allowing us to compare their performance and scalability. To better analyze the performance of these systems, we evaluated the pure in-DRAM stores that utilize CLHT, LFHT, and B+ tree. It is evident that for both hash-based and B+ tree-based systems, the in-DRAM stores achieve the highest performance. Additionally, the HeterMM-based KV stores outperform other NVM-based stores, even when applying the same index. Based on these re-

Table 2 Performance improvement of HeterMM compared to TIPS.

	YCSB-WH	YCSB-RH	YCSB-RO	YCSB-SH
CLHT	3.56x	4.19x	5.40x	#
LFHT	3.65x	4.64x	7.37x	#
B+ tree	2.06x	1.11x	1.51x	5.60x

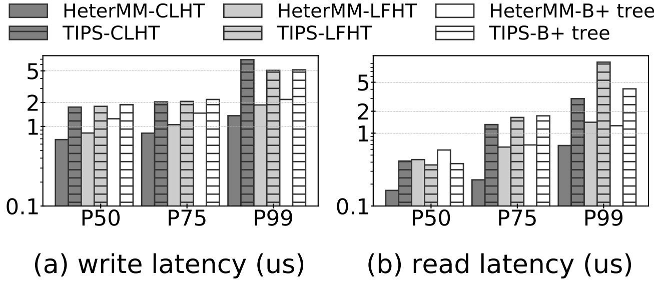


Fig. 10 Latency (in log scale) for the YCSB-WH workload with 16 workers.

sults, we make the following observations:

(a) HeterMM shows much less overhead than TIPS.

As depicted in the figures, HeterMM-based systems consistently outperformed TIPS-based ones when applying the same index. Specifically, the former demonstrated up to 3.65x better performance than the latter when utilizing CLHT, 7.37x better performance when utilizing LFHT, and 5.6x better when utilizing B+tree. Table 2 shows the performance comparison of systems using HeterMM and TIPS when applying 16 workers and Fig. 10 shows their latency for the YCSB-WH workload. They all highlight the significant advantages of HeterMM.

Specifically, these advantages can be attributed to the better utilization of DRAM and reduced access to NVM in HeterMM. Specifically, each write request results in less than three NVM writes in HeterMM, including two for the operation log and one for background flushing operations (in batch). However, it might result in five NVM writes in TIPS, including two for the operation log, one for persisting the value, one for modification of the index, and one for the undo log. As a result, TIPS suffers from the write amplification on NVM. Moreover, the object-oriented NVM space management in TIPS exhibits poorer performance compared to the block-oriented management in HeterMM.

For read requests, there is a high likelihood for HeterMM-based systems that the data resides in DRAM due to the in-place-update region and the read cache. Otherwise, the data will be read directly from NVM. In contrast, TIPS must access the in-NVM value, resulting in at least one NVM read. Besides, the embedded hash table in TIPS would prevent it from fully utilizing the highly optimized original indexes while HeterMM is designed to show the strengths of the original indexes.

For scan requests, the read cache in TIPS is skipped. TIPS always needs to traverse the B+ tree in NVM and the operation log to check potential results that have not been replayed. However, HeterMM only needs to traverse the B+ tree in DRAM and retrieve targeting data. As a result, the HeterMM-based system shows 5.6x better performance than the TIPS-based one with the YCSB-SH workloads.

(b) The performance of systems utilizing HeterMM is highly relative to the original index.

By designing an efficient data storage mechanism, HeterMM strives to leverage the strengths of the original indexes. Figures 8 and 9 demonstrate that the performance of HeterMM-based systems follows the same trend as the pure in-DRAM system applying the same index. Therefore, HeterMM-LFHT shows better scalability than HeterMM-CLHT because the same phenomenon happens to DRAM-LFHT and DRAM-CLHT.

Specifically, DRAM-CLHT exhibits performance degradation when there are too many workers. A similar trend is observed in HeterMM-CLHT but with later performance degradation and a lower per-

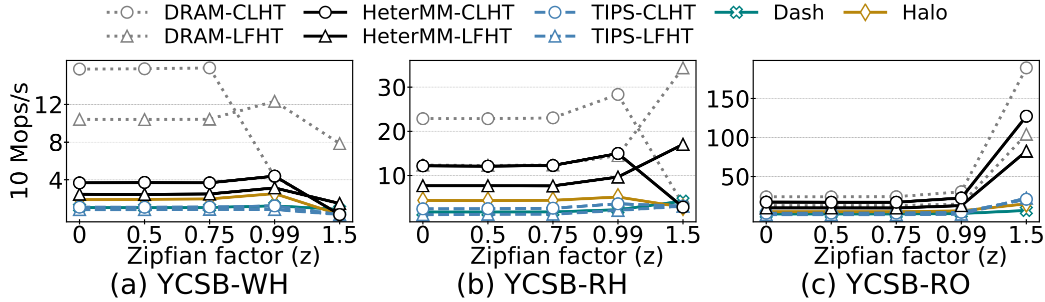


Fig. 11 Performance comparison among different hash stores with various skewness.

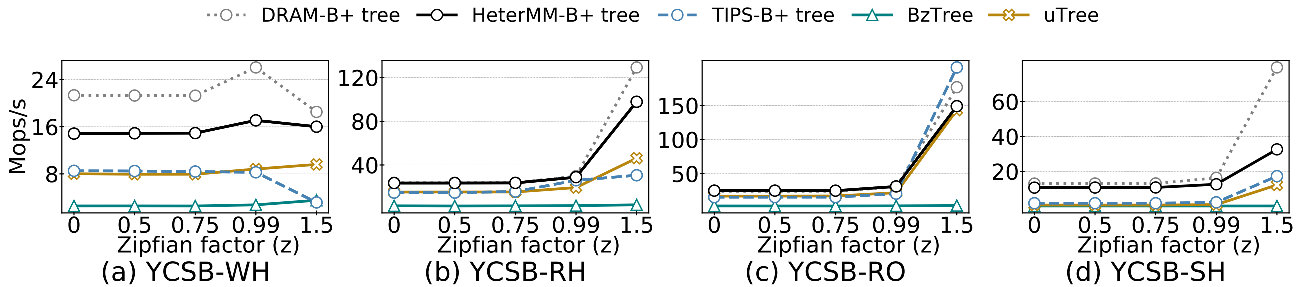


Fig. 12 Performance comparison among different B+ tree stores with various skewness.

formance peak. Moreover, HeterMM-CLHT performed better than HeterMM-LFHT with the YCSB-RH and YCSB-RO workloads, which also comply with the performance of pure in-DRAM systems. As it is usually hard for one index to fit all, HeterMM tries its best to hold the strengths of different indexes, and it is important to choose the proper one for a given scenario.

(c) HeterMM-based systems could achieve better performance than NVM-optimized systems.

Figures 8 and 9 demonstrate that the best performance among NVM-based KV stores, across various workloads, is achieved by systems utilizing HeterMM. This is because HeterMM maintains the hot data in DRAM and could better leverage the performance of DRAM. In contrast, Halo only puts the index in DRAM and μTree maintains the internal nodes of B+ Tree in DRAM, while Dash and BzTree almost put everything in NVM. The results highlight the importance of the full utilization of DRAM.

6.2.2 Varying Skewness

The skewness of data access often has a significant impact on the performance of KV stores. In this set of experiments, we varied the skewness of data accesses, represented by the Zipf factor z, while keeping the number of workers fixed at 16, which corresponds to the number of physical cores. The results are presented in Figures 11 and 12, showing that HeterMM-based systems outperformed other NVM-based systems. In general, all systems exhibited a similar trend as the skewness increased, indicating that the impact of skewness on the performance discrepancy among different systems was little, except in cases of extreme skewness, such as when the Zipfian factor was set to 1.5.

Notably, with the Zipfian factor as 1.5, the performance of all hash-based systems, specifically those utilizing the CLHT index, experienced a sharp drop for the YCSB-WH workloads. This can be attributed to the intensified competition for resources such

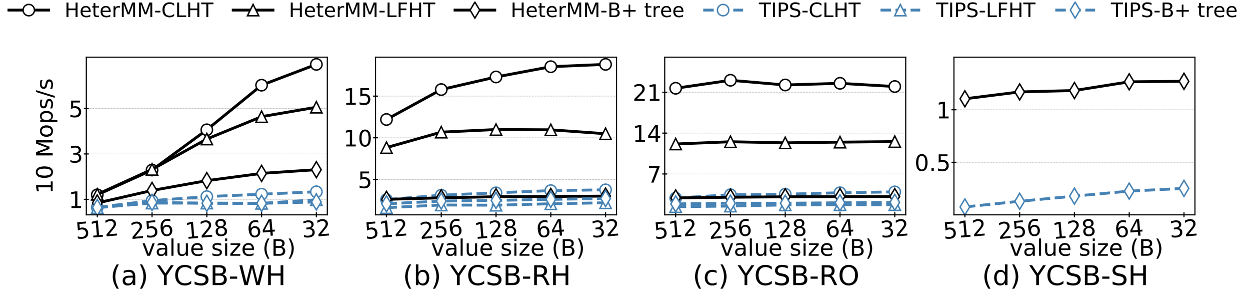


Fig. 13 Performance comparison among different systems with various value sizes.

as locks, which became the bottleneck of the systems. Consequently, the performance discrepancy between HeterMM-based and TIPS-based systems decreased. A similar phenomenon was observed in CLHT-based systems for the YCSB-RH workloads. However, this was not the case for LFHT-based and B+ tree-based systems due to the better scalability of these two indexes.

In contrast, for the YCSB-RO workloads, there is a significant performance improvement when the Zipfian factor increases to 1.5 for almost all systems. This improvement can be attributed to a more skewed workload, which is more favorable for CPU cache hit and DRAM hit ratios. Moreover, the performance improvement is more pronounced for the HeterMM-based systems compared to TIPS-based ones, as HeterMM can better leverage the strengths of DRAM-optimized indexes.

In particular, when the Zipfian factor increases to 1.5, the performance of TIPS-B+ tree was over that of HeterMM-B+ tree with the read-only workloads. This is because the gains of the in-DRAM hash table in TIPS could make up for its cost of more NVM reads.

In conclusion, the performance changes of different systems with varying skewness are primarily influenced by the characteristics of the original index. However, even in the worst-case scenario, HeterMM consistently outperformed TIPS.

6.2.3 Varying Value Size

The size of values in KV stores can significantly affect their performance. To evaluate this impact, we conducted experiments on HeterMM-based and TIPS-based systems with varying value sizes ranging from 32 bytes to 512 bytes. In these experiments, we adjusted the size of the DRAM region to ensure that it could accommodate approximately 25% of the data for each configuration. Results of different value sizes are presented in Fig. 13.

The performance of all systems generally improves as the value size decreases. This is because larger values result in increased overhead on operation logging and data storage. However, there are notable differences in the YCSB-RO workloads for HeterMM-based systems that utilize the CLHT as the index. Specifically, there are two distinct throughput peaks observed at a value size of 256 bytes and 64 bytes, respectively. This can be attributed to that the access granularity of the Optane device is 256 bytes and the CPU cacheline size is 64 bytes. Therefore, 256 bytes provides the best access performance for Optane NVM [45], while 64 bytes is optimal for DRAM. However, the impact on write-heavy workloads is not pronounced as HeterMM employs block-level writing and avoids random writes on NVM.

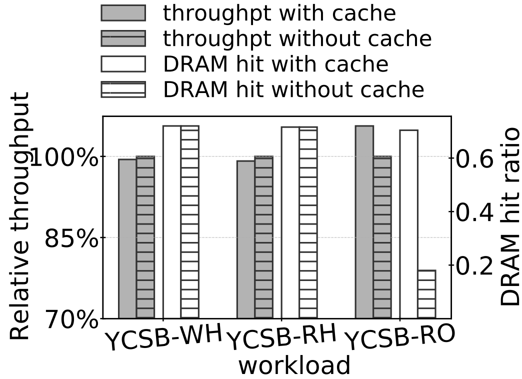


Fig. 14 Effects of the read cache on the throughput and DRAM hit ratio.

6.3 Design Analysis

In this section, we assess the effect of the read cache and the failure recovery and take the system based on CLHT as a representative for experiments.

6.3.1 Read Cache

To address potential performance limitations of read requests, a read cache is introduced in HeterMM. The read cache stores frequently accessed data in DRAM, improving read performance and mitigating the impact of slower NVM reads. The effectiveness of the read cache is measured using the DRAM hit ratio, which represents the proportion of read data served directly from the DRAM region. However, the introduction of the read cache also has some side effects as it competes with the write region for limited DRAM space, which can affect the overall efficiency of the system. Additionally, managing the read cache incurs additional overhead in terms of cache management operations.

The impact of the read cache is evaluated in terms of system throughput and DRAM hit ratios for different workloads, as illustrated in Fig. 14. The results indicate that the read cache introduces negligible overhead for the YCSB-WH and YCSB-RH workloads, as the DRAM distribution strategy employed (as described in Section 5.1) disables the read cache for such workloads. However, even without the read cache, the system could achieve a high DRAM hit ratio. It is because the frequent *set* requests will bring data in the overlap region back into the in-place-update region and therefore keep hot data in DRAM.

In contrast, the benefits of the read cache are evident in the YCSB-RO workloads, particularly in terms of the DRAM hit ratio. The read cache significantly increases the DRAM hit ratios from 18% to 70%, resulting in an approximate 5.6% improvement in throughput. These findings demonstrate that the read cache effectively improves performance for read-intensive workloads, while its impact on write-intensive workloads is minimal.

In conclusion, the unified logical space design in HeterMM, encompassing both DRAM and NVM, eliminates the necessity for a read cache in certain scenarios. However, it can still provide performance benefits for read-only workloads. The decision to utilize a read cache should be based on the specific workload requirements and the trade-off between performance improvements and the associated overhead.

6.3.2 Failure Recovery

To evaluate the effectiveness of the operation log mechanism, we conducted 50 experiments where we deliberately terminated the procedure using the SIGKILL signal, following a similar approach as in previous studies [16]. In all cases, the system was able to recover successfully. During these experiments, we employed 16 threads for the recovery process, and the time required for recovery of the 200 million tuples varied between 4s and 14s, depending on the volume of data stored in NVM and the operation logs. These results demonstrate

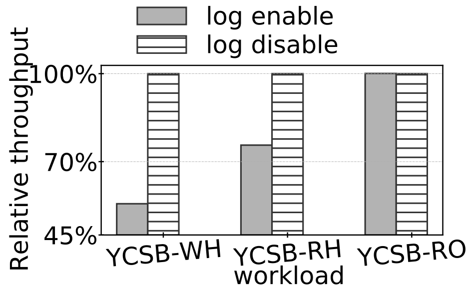


Fig. 15 Effects of the operation log on the throughput.

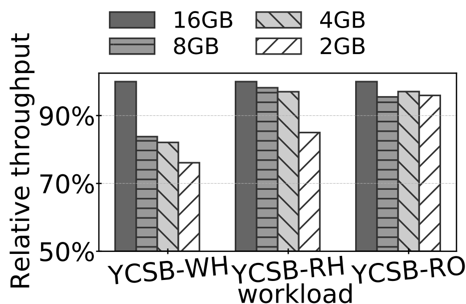


Fig. 16 Performance sensitivity of HeterMM-CLHT for varying DRAM sizes used for heterogenous data storage.

the reliability and efficiency of the recovery mechanism in handling unexpected failures.

To evaluate the overhead introduced by the operation log, we conducted experiments where we disabled the operation log and measured the throughput of the system. The results are shown in Fig. 15. As expected, the performance overhead is proportional to the ratio of *set* requests in the workload. However, the overhead remains acceptable, with a maximum decrease of 45% even for the write-heavy workload.

6.4 Sensitivity Analysis

A series of experiments were conducted to analyze the sensitivity of DRAM sizes used for heterogenous data storage, with the results presented in Fig. 16 for the HeterMM-CLHT as a representative. The impact of DRAM utilization is most pronounced in the YCSB-WH workloads. This is attributed to the larger DRAM capacity enabling the storage of more tuples, leading to increased in-place updates and reduced NVM flushing operations and space allocation overhead. However, no significant differences are observed in the YCSB-RH workload except when the DRAM size drops to 2GB. This is because write operations consistently bring hot data into DRAM, minimizing the reliance on DRAM size requirements. In the case of the YCSB-RO workload, a performance improvement is observed when the DRAM size reaches 16GB, as it can accommodate nearly all the data.

7 Discussion

The decision by Intel to wind down its Optane DIMM business has raised concerns about the future of byte-addressable storage research. Even though, we argue that NVM technology, initially proposed to address the scaling limitations and volatility of DRAM [7], remains relevant and necessary. Our assumptions regarding the NVM are mainly based on the NVDIMM-P standard [46], a widely adopted NVM standard [45]. It is reasonable to expect that future NVM products will adhere to this standard and share similar characteristics [47].

In addition to NVM, other devices have been studied to extend the capacity of DRAM [48, 49]. FlatFlash [49], for instance, leverages the byte-addressability provided by the PCIe interconnect and the internal memory inside SSD controllers. It builds a unified memory storage that uses SSD as an extension of DRAM. FlatFlash achieves latency in the range of hundreds of nanoseconds for sequential access and dozens of microseconds for random access, indicating that its random access speed is slower compared to most NVM devices. These various approaches provide alternative opportunities to extend the capacity of DRAM and contribute to the significance of exploring HeterMM.

8 Conclusion

We propose HeterMM, a versatile framework that leverages in-DRAM indexes in KV stores on heterogeneous memory. HeterMM incorporates a plug-in programming model, allowing for the integration of various types of indexes. By prioritizing the maintenance of both the index and hot data in DRAM, HeterMM maximizes the utilization of the superior performance of DRAM. Additionally, HeterMM addresses potential performance bottlenecks associated with NVM access by applying a hotness-aware storage mechanism. Our evaluation demonstrates that HeterMM outperforms existing state-of-the-art frameworks that convert in-DRAM indexes to persistent ones. Furthermore, HeterMM can surpass NVM-specific KV stores by carefully selecting the appropriate index for specific scenarios.

References

1. Qi X, Hu H, Guo J, Huang C, Zhou X, Xu N, Fu Y, Zhou A. High-availability in-memory key-value store using RDMA and optane DCPMM. Frontiers Comput. Sci., 2023, 17(1): 171603
2. Huang C, Hu H, Qi X, Zhou X, Zhou A. Rs-store: Rdma-enabled skiplist-based key-value store for efficient range query. Frontiers Comput. Sci., 2021, 15(6): 156617
3. Redis. https://redis.io/, 2023 (accessed August, 2023)
4. Leis V, Haubenschild M, Kemper A, Neumann T. Leanstore: In-memory data management beyond main memory. In: 34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018. 2018, 185–196
5. Neumann T, Freitag M J. Umbra: A disk-based system with in-memory performance. In: 10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings. 2020
6. Banakar V, Wu K, Patel Y, Keeton K, Arpaci-Dusseau A C, Arpaci-Dusseau R H. Wiscsort: External sorting for byte-addressable storage. Proc. VLDB Endow., 2023, 16(9): 2103–2116
7. Huang K, He Y, Wang T. The past, present and future of indexing on persistent memory. Proc. VLDB Endow., 2022, 15(12): 3774–3777
8. Chen Y, Lu Y, Yang F, Wang Q, Wang Y, Shu J. Flatstore: An efficient log-structured key-value storage engine for persistent memory. In: Larus J R, Ceze L, Strauss K, eds, ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020. 2020, 1077–1091
9. Gugnani S, Kashyap A, Lu X. Understanding the idiosyncrasies of real persistent memory. Proc. VLDB Endow., 2020, 14(4): 626–639
10. Yang J, Kim J, Hoseinzadeh M, Izraelevitz J, Swanson S. An empirical guide to the behavior and use of scalable persistent memory. login Usenix Mag., 2020, 45(3)
11. Xiang L, Zhao X, Rao J, Jiang S, Jiang H. Characterizing the performance of intel optane persistent memory: a close look at its on-dimm buffering. In: Bromberg Y, Kermarrec A, Kozyrakis C, eds, EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022. 2022, 488–505
12. Tian C, Liu H, Liao X, Jin H. Ucat: heterogeneous memory management for unikernels. Frontiers Comput. Sci., 2023, 17(1): 171204
13. Chen T, Liu H, Liao X, Jin H. Resource abstraction and data placement for distributed hybrid memory pool. Frontiers Comput. Sci., 2021, 15(3): 153103
14. He Y, Lu D, Huang K, Wang T. Evaluating persistent memory range indexes: Part two. Proc. VLDB Endow., 2022, 15(11): 2477–2490
15. Hu D, Chen Z, Wu J, Sun J, Chen H. Persistent memory hash indexes: An experimental evaluation. Proc. VLDB Endow., 2021, 14(5): 785–798
16. Ramanathan M K, Kim W, Fu X, Monga S K, Lee H W, Jang M, Mathew A, Min C. TIPS: making volatile index structures persistent with DRAM-NVMM tiering. In: Calciu I, Kuenning G, eds, 2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021. 2021, 773–787
17. Lee S K, Mohan J, Kashyap S, Kim T, Chidambaram V. Recipe: converting concurrent DRAM indexes to persistent-memory indexes. In: Brecht T, Williamson C, eds, Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019. 2019, 462–477
18. Cooper B F, Silberstein A, Tam E, Ramakrishnan R, Sears R. Benchmarking cloud serving systems with YCSB. In: Hellerstein J M, Chaudhuri S, Rosenblum M, eds, Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana,

USA, June 10-11, 2010. 2010, 143–154
19. Intel . Intel reports second-quarter 2022 financial results, 2022
20. Liu J, Chen S, Wang L. Lb+trees: Optimizing persistent index performance on 3dxpoint memory. Proc. VLDB Endow., 2020, 13(7): 1078–1090
21. Zhou X, Shou L, Chen K, Hu W, Chen G. Dptree: Differential indexing for persistent memory. Proc. VLDB Endow., 2019, 13(4): 421–434
22. Oukid I, Lasperas J, Nica A, Willhalm T, Lehner W. Fptree: A hybrid SCM-DRAM persistent and concurrent b-tree for storage class memory. In: Özcan F, Koutrika G, Madden S, eds, Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016. 2016, 371–386
23. Xia F, Jiang D, Xiong J, Sun N. Hikv: A hybrid index key-value store for DRAM-NVM memory systems. In: Silva D D, Ford B, eds, 2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017. 2017, 349–362
24. Li Z, Tan Z, Chen J. HASDH: A hotspot-aware and scalable dynamic hashing for hybrid DRAM-NVM memory. In: 39th IEEE International Conference on Computer Design, ICCD 2021, Storrs, CT, USA, October 24-27, 2021. 2021, 154–161
25. Benson L, Makait H, Rabl T. Viper: An efficient hybrid pmem-dram key-value store. Proc. VLDB Endow., 2021, 14(9): 1544–1556
26. Zhu J, Huang K, Zou X, Huang C, Xu N, Fang L. HDNH: a read-efficient and write-optimized hashing scheme for hybrid DRAM-NVM memory. In: Sun X, Shende S, Kalé L V, Chen Y, eds, ICPP 2021: 50th International Conference on Parallel Processing, Lemont, IL, USA, August 9 - 12, 2021. 2021, 47:1–47:10
27. Huang Y, Pavlovic M, Marathe V J, Seltzer M I, Harris T, Byan S. Closing the performance gap between volatile and persistent key-value stores using cross-referencing logs. In: Gunawi H S, Reed B, eds, 2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018. 2018, 967–979
28. Cui L, He K, Li Y, Li P, Zhang J, Wang G, Liu X. Swapkv: A hotness aware in-memory key-value store for hybrid memory systems. IEEE Transactions on Knowledge and Data Engineering, 2023, 35(1): 917–930
29. Chandramouli B, Prasaad G, Kossmann D, Levandoski J J, Hunter J, Barnett M. FASTER: A concurrent key-value store with in-place updates. In: Das G, Jermaine C M, Bernstein P A, eds, Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018. 2018, 275–290
30. Memaripour A S, Izraelevitz J, Swanson S. Pronto: Easy and fast persistence for volatile data structures. In: Larus J R, Ceze L, Strauss K, eds, ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020. 2020, 789–806
31. Friedman M, Ben-David N, Wei Y, Blelloch G E, Petrank E. Nvtraverse: in NVRAM data structures, the destination is more important than the journey. In: Donaldson A F, Torlak E, eds, Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020. 2020, 377–392
32. Xiang L, Zhao X, Rao J, Jiang S, Jiang H. Characterizing the performance of intel optane persistent memory: a close look at its on-dimm buffering. In: Bromberg Y, Kermarrec A, Kozyrakis C, eds, EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022. 2022, 488–505
33. Atikoglu B, Xu Y, Frachtenberg E, Jiang S, Paleczny M. Workload analysis of a large-scale key-value store. In: Harrison P G, Arlitt M F, Casale G, eds, ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '12, London, United Kingdom, June 11-15, 2012. 2012, 53–64
34. Yang J, Yue Y, Rashmi K V. A large-scale analysis of hundreds of in-memory key-value cache clusters at twitter. ACM Trans. Storage, 2021, 17(3): 17:1–17:35
35. Optane. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/optane-persistent-memory-200-series-brief.html, 2023 (accessed August, 2023)
36. Yang J, Kim J, Hoseinzadeh M, Izraelevitz J, Swanson S. An empirical guide to the behavior and use of scalable persistent memory. In: Noh S H, Welch B, eds, 18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020. 2020, 169–182
37. Raybuck A, Stamler T, Zhang W, Erez M, Peter S. Hemem: Scalable tiered memory management for big data applications and real NVM. In: Renesse v R, Zeldovich N, eds, SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021. 2021, 392–407
38. Vitter J S. Implementations for coalesced hashing.

Commun. ACM, 1982, 25(12): 911–926

39. Hu D, Chen Z, Che W, Sun J, Chen H. Halo: A hybrid pmem-dram persistent hash index with fast recovery. In: Ives Z, Bonifati A, Abbadi A E, eds, SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022. 2022, 1049–1063
40. David T, Guerraoui R, Trigonakis V. Asynchronized concurrency: The secret to scaling concurrent search data structures. In: Özturk Ö, Ebcioglu K, Dwarkadas S, eds, Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2015, Istanbul, Turkey, March 14-18, 2015. 2015, 631–644
41. Michael M M. High performance dynamic lock-free hash tables and list-based sets. In: Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '02. 2002, 73–82
42. Lu B, Hao X, Wang T, Lo E. Scaling dynamic hash tables on real persistent memory. SIGMOD Rec., 2021, 50(1): 87–94
43. Arulraj J, Levandoski J J, Minhas U F, Larson P. Bztree: A high-performance latch-free range index for non-volatile memory. Proc. VLDB Endow., 2018, 11(5): 553–565
44. Chen Y, Lu Y, Fang K, Wang Q, Shu J. utree: a persistent b+-tree with low tail latency. Proceedings of the VLDB Endowment, 2020, 13(12): 2634–2648
45. Huang W, Ji Y, Zhou X, He B, Tan K. A design space exploration and evaluation for main-memory hash joins in storage class memory. Proc. VLDB Endow., 2023, 16(6): 1249–1263
46. JEDEC . DDR4 NVDIMM-P BUS PROTOCOL. https://www.jedec.org/system/files/docs/JESD304-4-01.pdf, 2021
47. Benson L, Papke L, Rabl T. Perma-bench: Benchmarking persistent memory access. Proc. VLDB Endow., 2022, 15(11): 2463–2476
48. Huang J, Badam A, Qureshi M K, Schwan K. Unified address translation for memory-mapped ssds with flashmap. In: Marr D T, Albonesi D H, eds, Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015. 2015, 580–591
49. Abulila A H M O, Mailthody V S, Qureshi Z, Huang J, Kim N S, Xiong J, Hwu W W. Flatflash: Exploiting the byte-accessibility of ssds within a unified memory-storage hierarchy. In: Bahar I, Herlihy M, Witchel E, Lebeck A R, eds, Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019. 2019, 971–985