HPDK: A Hybrid PM-DRAM Key-Value Store for High I/O Throughput

Bihui Liu, Zhenyu Ye, Qiao Hu, Yupeng Hu, *Senior Member, IEEE*, Yuchong Hu, *Member, IEEE*, Yang Xu, and Keqin Li, *Fellow, IEEE*

***Abstract*—This paper explores the design of an architecture that replaces Disk with Persistent Memory (PM) to achieve the highest I/O throughput in Log-Structured Merge Tree (LSM-Tree) based key-value stores (KVS). Most existing LSM-Tree based KVSs use PM as an intermediate or smoothing layer, which fails to fully exploit PM's unique advantages to maximize I/O throughput. However, due to PM's distinct characteristics, such as byte addressability and short erasure time, simply replacing existing storage with PM does not yield optimal I/O performance. Furthermore, LSM-Tree based KVSs often face slow read performance. To tackle these challenges, this paper presents HPDK, a hybrid PM-DRAM KVS that combines level compression for LSM-Trees in PM with a B$^+$-tree based in-memory search index in DRAM, resulting in high write and read throughput. HPDK also employs a key-value separation design and a live-item rate-based dynamic merge method to reduce the volume of PM writes. We implement and evaluate HPDK using a real PM drive, and our extensive experiments show that HPDK provides 1.25-11.8 and 1.47-36.4 times higher read and write throughput, respectively, compared to other state-of-the-art LSM-Tree based approaches.**

***Index Terms*—B^+-tree, key-value store, log-structured merge trees, persistent memory.**

I. Introduction

THE rapid growth of large-scale cloud computing applications, such as mobile internet, e-commerce, and social networking platforms, has led to an influx of massive real-time data access requests. This has resulted in a significant increase in throughput requirements for read and write operations [1], [2]. Yahoo! has reported a shift in their typical workloads, with reads and writes now comprising similar proportions [3].

LSM-Tree based Key-Value Stores (KVSs) [4], [5], [6], [7] with Persistent Memories (PM) have recently emerged as crucial components to support various data-intensive applications in data centers. PM offers byte-addressability, large capacity, and non-volatility, which opens new opportunities for building high-throughput KVSs. However, there are two main challenges in LSM-Tree based KVSs that hinder further improvements in throughput. Firstly, LSM-Tree based KVSs exhibit high write throughput but slow read performance, as read operations necessitate scanning multiple files. Secondly, most existing studies [8], [9], [10], [11] combine PM with disks, such as Solid State Disks (SSDs) and Hard Disk Drives (HDDs). The differences between PM and SSD are listed in Table I. While data is stored on the disk, PM is used as an intermediate or buffering layer to improve system latency. Accessing data on SSDs results in higher latency compared to PM; thus, replacing SSDs with PM for data storage may be more effective. However, the differing characteristics of byte addressability and rapid erasure time for PM means that simply replacing SSDs is not the best solution.

TABLE I
The Differences Between PM and SSD

	PM	SSD
Storage	Non-Volatile	Non-Volatile
Read/Write Granularity	Load/Store Instructions Cache Line	I/O Command Block

To address these challenges, this paper proposes HPDK[1], a high I/O throughput KVS based on a hybrid PM-DRAM system. The main architecture incorporates a hybrid index with an LSM-tree in PM, a B$^+$-tree in DRAM, and PM replacing the disk for storing all persistent data. B$^+$-tree based KVSs [12], [13] have high read throughput but poor write throughput. Thus, HPDK employs LSM-Tree to handle incoming key-value pairs for low write latency while maintaining a B$^+$-tree in DRAM to index key-value pairs in PM and improve read performance. Furthermore, HPDK compresses LSM-Tree levels to eliminate write amplification generated by the compaction process. Since there is no compaction process, expired key-value pairs affect space utilization and system throughput. To address this, HPDK implements a live rate-based dynamic merge algorithm for garbage collection of data blocks with many expired key-value items. Finally, HPDK adopts a key-value separation scheme to reduce PM writes and extend PM's service life.

[1]The source code of HPDK is https://github.com/QiaoHU-HNU/HPDK.git

Manuscript received 29 April 2023; revised 12 February 2024; accepted 9 March 2024. Date of publication 18 March 2024; date of current version 10 May 2024. This work was supported by Hunan Province Outstanding Youth Fund Project under Grant 2022JJ10018. Recommended for acceptance by S. Guo. *(Corresponding authors: Qiao Hu; Yupeng Hu.)*

Bihui Liu, Zhenyu Ye, Qiao Hu, Yupeng Hu, and Yang Xu are with the College of Computer Science and Electronic Engineering, Hunan University, Changsha, Hunan 410082, China (e-mail: bihuiliu@hnu.edu.cn; yezhenyu@hnu.edu.cn; huqiao@hnu.edu.cn; yphu@hnu.edu.cn; xuyangcs@hnu.edu.cn).

Yuchong Hu is with the School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, Hubei 430074, China (e-mail: yuchonghu@hust.edu.cn).

Keqin Li is with the Department of Computer Science, State University of New York, New Paltz, NY 12561 USA (e-mail: lik@newpaltz.edu).

Digital Object Identifier 10.1109/TC.2024.3377914

We evaluate HPDK using the db_bench micro-benchmark [7] for read/write throughput and YCSB macro-benchmark [14] suite for real-world workloads released with three representative LSM-Tree based KVSs. Our extensive experiments demonstrate that HPDK provides 1.25-11.8 and 1.47-36.4 times higher read and write throughput respectively, compared to other methods.

In summary, the main contributions of this work are as follows:

1) We are the first that propose a hybrid LSM-Tree based KVS architecture in a PM-DRAM environment, completely replacing disks with PM. The hybrid KVS architecture combines an LSM-tree in PM and a B^+-tree in DRAM, aiming to maximize I/O throughput within the PM-DRAM system.
2) Our approach includes a meticulous design of a level-compression scheme and a dynamic merge scheme for LSM-Tree. These schemes effectively reduce write amplification during garbage collection. Moreover, we introduce a key-value separation scheme for the B^+-tree, which significantly decreases PM writes.
3) To demonstrate the exceptional I/O throughput achieved by HPDK, we implement it into an actual PM-DRAM system and conduct extensive experiments.

The rest of this paper is organized as follows. We motivate our work in Section II. Then we present the design and implementation of HPDK in Section III and Section IV, respectively. Next we evaluate HPDK in Section V. The related work and limitations are discussed in Section VI and Section VII separately. Finally we make a conclusion in Section VIII.

II. Background and Motivation

In this section, we discuss the LSM-Tree and PM, as well as our considerations and motivations for building a throughput-optimized KVS on PM.

A. LSM-Trees and LevelDB

LSM-Tree is a persistent storage data structure that provides high write throughput for update-frequent applications. In LSM-Tree, index changes are first deferred and buffered in DRAM, then flushed to disk, merging and sorting level by level. Due to its high put throughput, LSM-Tree and its variants are widely used in KVS [15], [16], [17], [18]. LevelDB [7] is one of the popular LSM-Tree-based KVS that is extended by Google Bigtable [4]. The LevelDB architecture is shown in Fig. 1. Its main data structure consists of a fixed-size buffer MemTable and an Immutable MemTable in DRAM, a multi-level data organization structure for disk. The capacity of each level is configured to be ten times larger than its previous level. The key-value pairs in each level are ordered except level 0. At level 0, only the key-value pairs inside each SSTable are ordered without sorting the SSTable, which improves flush throughput from the DRAM to the disk.

For the PUT operation, LevelDB first appends the newly written key-value pairs to the Write-Ahead-Logging(WAL) and then adds it to MemTable. MemTable uses a skip list to sort all

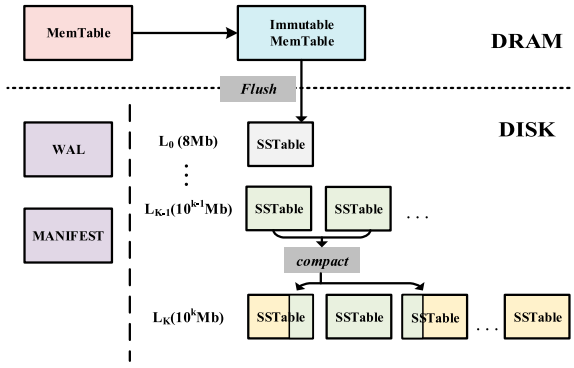

Fig. 1. LevelDB architecture. *This figure shows the LevelDB architecture. The memory component consists of a MemTable and an Immutable MemTable; the disk components include multi-levels of SSTables and some component files with metadata. Flush operation occurs when there is an Immutable MemTable in memory. Compact operation is conducted level by level to merge SSTables from the lower to higher levels in the background.*

buffered key-value pairs ordered by key. When the MemTable is full, LevelDB marks it as Immutable MemTable and then flushes it to the disk of level 0 as SSTable, and the WAL is deleted. SSTables need to be compact to higher levels when any level reaches the limit size. The compact process consists of three steps. First, LevelDB reads the key-value pairs in level i and level i+1 into memory(i>0). Second, LevelDB sorts and reorganizes valid key-value pairs into SSTable. Finally, LevelDB writes the SSTable with valid key-value pairs back to the level i+1. Those victim SSTables and the overlapped SSTables will become invalid and then delete from the disk. In LevelDB, an inserted key-value pair needs to be continuously sorted, merged and written to disk, and it will generate a considerable write amplification.

For the GET operation, LevelDB first searches MemTable and Immutable MemTable in DRAM in order; if not found, then searches the multi-level from level 0 to level n in order. For level 0, LevelDB will search all of the SSTable; for levels 1-n, LevelDB only needs to search candidate one because they are ordered. If the key-value pair is not found at one level, it will search the next level until the last level. The search operation ends once the key-value pair is found and the result is returned. In the worst case, to obtain a key-value pair LevelDB needs to read all the SSTables at level 0 and one SSTable in each remaining level.

LevelDB also maintains the metadata of the SSTable in the current LSM-Tree in a file called MANIFEST. The metadata includes a list of SSTable files for each level and a range of keys for each SSTable. During the compact process, the metadata changes of the SSTable are first logged and recorded in MANIFEST when the compact is finished, and then the expired SSTable is deleted. When the system crashes, LevelDB can recover to a consistent KVS even if the crash occurs during the compact process.

Although LevelDB has the advantages of high update throughput and consistency, it still has the disadvantages of slow read throughput and high I/O amplification. To build a high-throughput KVS base on LSM-Tree, we need to solve these problems.

B. Persistent Memory

Persistent Memory is an emerging computer storage device with non-volatile and byte-addressable that can provide data persistence while achieving I/O throughput comparable to DRAM. Data on PM are available upon power-up, and applications do not need to spend time warming up the cache. They can access the data as soon as the memory is mapped. Compared to DRAM, PM has non-volatile as well as the ability to provide more data to a multi-core environment with the same area of memory slot [9], [19]. Compared to traditional disks, PM offers high throughput, low I/O latency, low power consumption, and small volumes [20]. Systems that include PM can provide faster boot times, faster access to large in-memory data sets, and lower total ownership costs, outperforming traditional SSD based data center configurations. However, PM has a shorter erase time than traditional disks.

C. Motivation and Challenges

PM offers new opportunities for constructing high I/O throughput KVSs. However, directly incorporating PM into existing KVSs may not fully leverage PM's advantages and could even be counterproductive. KVSs designed for DRAM and common disk structures may become inefficient when implemented within a PM-DRAM architecture. Such non-optimized KVSs could waste space and bandwidth and even shorten PMs' lifetimes.

There have been many KVS designs based on LSM-Trees for PM [8], [9], [10], [21]. Most of these designs employ PM as buffering or smoothing layers to enhance system performance. For example, in NoveLSM [10], PM is a middle layer to store level 0 and level 1 data; HiKV [9] maintains a hybrid index in a hybrid memory system of DRAM and PM for a KVS and supports rich key-value operations; SLM-DB [21] uses PM as DRAM. However, they do not exploit the advantages of PM to build a fully optimized KVS for high throughput. Moreover, most of the existing KVSs which based on LSM-Trees have an inherent I/O amplification structure. In LevelDB [7], an inserted key-value pair needs to be constantly merged and sorted, and written to disk by background compact. For a n level LSM tree from LevelDB, its write amplification can be as high as $10n$ [22], [23], [24]. It dramatically affects the lifetime of the PM device if we use PM to directly store these data because PM devices have a shorter erasure lifetime than SSD. For GET operations, LevelDB also needs to search multiple SSTables in most cases, significantly affecting read throughput. Hence, it is crucial to design a new structure for LSM-Trees based KVS which can fully utilize PM and ensure high I/O throughput compared to other designs.

There are several challenges in building a high-throughput LSM-Trees based KVS for PM-DRAM architecture.

- *For PM.* Utilizing PM's large capacity and non-volatility, we must select the appropriate PM usage to maximize system I/O throughput. Additionally, PM's shorter erasure time compared to other hardware impacts its lifetime if excessive writing occurs.

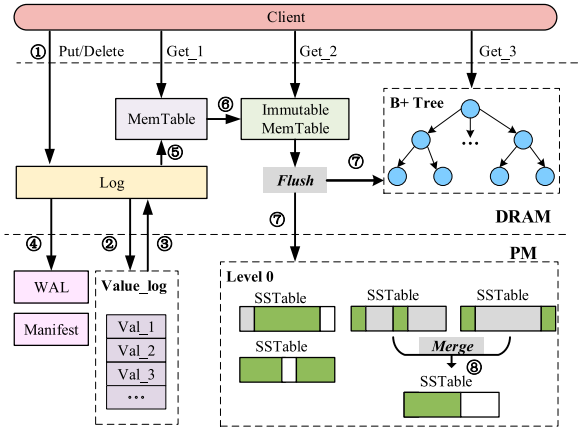

Fig. 2. HPDK architecture. *This figure shows the architecture of HPDK. The memory component consists of a MemTable, an Immutable MemTable, and a B^+-tree; the PM components include a single level 0, a value area, and some component files with metadata. Flush operation occurs when there is an Immutable MemTable in memory, and it will add key-address pairs into level 0 and B^+-tree at the same time. Merge operation run when the expired key-address pair reach the garbage collection condition we defined.*

- *For Key-Value Store.* Introducing PM into traditional KVS architectures requires KVS redesign and optimization for throughput. Also, considering PM's shorter lifetime, addressing potential I/O amplification structures in KVS for PM is necessary to extend PM's life without affecting I/O throughput.
- *For Overall System.* When building a high-throughput PM-based KVS, we must enable crash recovery to ensure availability and implement a garbage collection strategy to improve PM space utilization.

III. HPDK Design

HPDK is a KVS based on LSM-Tree and B^+-tree and designed for PM to achieves high I/O throughput with low read/write latency and low read/write amplification. This section presents the design of the HPDK. Fig. 2 shows the overall architecture of HPDK, which consists of a DRAM component and a PM component.

HPDK is a high-throughput KVS for PM-DRAM architecture with the following design principles:

1) HPDK stores all persistent data on PM to accelerate access time for persistent data, further enhancing I/O throughput.
2) HPDK is based on LevelDB and compresses its levels, retaining only level 0 to dramatically reduce write amplification and improve write throughput.
3) Similar to [9], [21], [25], HPDK incorporates a B^+-tree in DRAM to enhance read throughput. The B^+-tree stores all live key-address pairs in level 0, while data is stored in the B^+-tree's leaves.
4) To improve throughput for large value sizes, HPDK adopts a key-value separation design like [23] for key-address pairs. It stores the value in a designated area and the key with the value's address as the key-address pairs.

The B$^+$-tree also stores the address instead of the value. In most cases, the address size is smaller than the value [26], making the B$^+$-tree more compact.

In summary, HPDK designs and implements a dynamic compaction algorithm based on key-address item live rates for garbage collection. The algorithm merges SSTables with live data rates lower than a preset threshold, reclaims PM space, and ensures dirty data is cleared promptly, enhancing the device's space utilization. We will describe these designs in detail in the following sections.

A. Usage of PM

As emerging hardware, PM has the advantages of large storage capacity, non-volatility, and fast access speed. However, compared to DRAM, PM has a higher write latency ($7\times -22\times$), a lower bandwidth ($5\times -10\times$), and a lower endurance ($10^4 \times -10^7\times$) [9], [19]. To build a fully optimized KVS for I/O throughput, we directly use PM to store persistent data instead of SSD. This speeds up the overall data access speed of the system and improve the throughput of the KVS in this way. To store persistent data on PM instead of SSD, we should reduce the write to PM without affecting system throughput because the erasure time for PM is shorter than SSD. SSD. We can also improve PM's space utilization to save cost. We achieve this with the following design.

B. Level Compression

The main problem with LSM-Tree architectures like LevelDB is write and read amplification. Write (Read) amplification is the ratio between the amount of data written to (read from) the underlying storage device and the amount of data requested by the client. In LSM-Trees, write amplification mainly results from level design requiring constant merging and sorting data from low to high levels, causing the same data to be written repeatedly and generating significant write amplification. As SSTables are stored in PM in our design, we need to reduce write amplification. As shown in Fig. 2, we compress the level of LevelDB, making all SSTables flushed to PM at level 0. With only one level, the database no longer needs to frequently execute compact operations, dramatically reducing write amplification in the KVS. This also saves CPU and DRAM resources, as compaction operations require reading data into DRAM and involve many computation processes. Level compression does not directly improve database throughput but benefits the system by eliminating expensive operations like compaction.

C. B$^+$-Tree in DRAM

When we compress the level of LSM-Tree, the write amplification of the system is alleviated. However, when performing the read operation, if the read fails in memory, it will search the whole level 0. Because the SSTable in level 0 is stored unordered among them, it needs to scan all the SSTables to perform the search, which will generate a considerable read amplification and have a long latency. To solve this problem, as shown in Fig. 2, we keep a B$^+$-tree in DRAM, in which the latest version of key-address items in the PM is stored. When performing a read operation, we first search the MemTable and Immutable MemTable in DRAM as LevelDB; if the search is not success, we only need to search the B$^+$-tree instead of scanning all of the SSTable, which reduces the read amplification and improves the system read throughput.

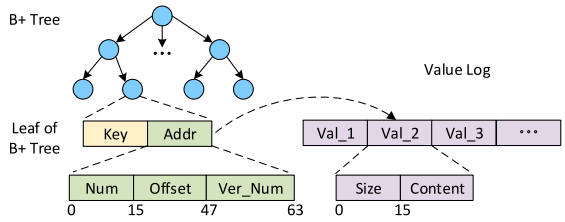

Fig. 3. HPDK data layout. *This figure shows the data layout of B$^+$-tree and value log in HPDK. Key and value addresses are stored in the leaf of B$^+$-tree while values are appended to a separate value log. The layout of the key-address project in the SSTable is the same as in the B$^+$-tree.*

In HPDK, when there is an Immutable MemTable, data is flushed to level 0 and written to the B$^+$-tree simultaneously. During the flush operation, the system creates a background thread to perform these operations. In the background thread, HPDK creates a new PM pool as a new SSTable, writes key-address pairs from the Immutable MemTable to it sequentially, and inserts them into the B$^+$-tree simultaneously. During the B$^+$-tree writing process, if the key does not exist, it is inserted directly at the appropriate location; if the key already exists, the value in the B$^+$-tree is updated to the current value; if comes a delete operation, the key-address item will be removed from the tree. After all key-address items are written to the SSTable and inserted into the B$^+$-tree, the metadata of the SSTable is updated and appended to the MANIFEST file as a log structure. Finally, the Immutable MemTable is deleted, and the thread ends.

D. Key Value Separation

In our design, the B$^+$-tree is stored in DRAM. In order to save space in the DRAM, we implement key-value separation like [23] for key-address items in HPDK. We write the values in an exceptional value append log and use shorter addresses to point them instead so that only the keys and corresponding addresses of values are stored. In our experiments, the data layout in the leaf of B$^+$-tree is as shown in Fig. 3. The address consists of a 6-byte value address (with a 2-byte pool number and a 4-byte offset) and a 2-Byte pool number. The first 6 bytes are used to locate the values, and the last 2 bytes are used to verify the validity of a key-address item from an SSTable. The layout of key-address items in SSTable is the same as those in B$^+$-tree, except that they do not have the last 2 Byte. The value item in the value log consists of a 4 Byte value size and the value content with variable length. In practical scenarios, the value size is larger than 8 bytes in most cases [26]; therefore, using the key-value separation reduces the amount of data stored in DRAM and reduces the amount of data written during B$^+$-tree balancing.

When a write operation comes in, HPDK first writes its value to the specified value area and returns its address, then uses the

returned address as the value of the key-address item for the write operation, and the record in the WAL is the key-address pair. In HPDK, value log is a PM pool with size of 128 MB, and it appends data at the end of the pool like a log structure. When the current pool is full, a new pool is created and data will be appended to the new pool. Only one pool accepts the write operations at the same time. Creating new pools continuously will eventually exhaust the memory capacity. To address this issue, we have implemented a simple approach that allows us to reuse existing pools. Each pool now keeps track of the current number of valid items it contains, enabling us to reuse a pool when the number of valid items reaches zero.

But it would be better to design a garbage collection method. For value area garbage collection, we can use a bitmap to record the invalid area and write new items in, which will not impact our I/O throughput. We plan to implement this in future work.

E. Live-Item Rate Based Dynamic Merge Algorithm

Since all SSTables are kept in one level, without a proper garbage collection mechanism, many invalid key-address items will occupy PM space in an update-intensive workload. It will result in lower PM space utilization and further affect system throughput. HPDK provides a live-item rate based dynamic merge algorithm to perform garbage collection.

We maintain the ratio of valid key-address pairs to all key-address pairs stored in each SSTable for selection purposes. We call this ratio as the live-item rate. If the live-item rate for an SSTable is lower than a predefined threshold, the SSTable contains excessive garbage that should be collected to improve disk space utilization. When an SSTable is created, the total number of key-address pairs stored in it is computed, and initially, the number of valid key-address pairs is equal to the total number of key-address pairs in the SSTable. When a key stored in the SSTable is updated with a fresh value, the key with the fresh value is stored in a new SSTable file. As a result, when we update a pointer to the new location object for the key in the B+-tree, we decrease the number of valid KV pairs in the original SSTable. Using these two numbers, we can compute the live-item rate for each SSTable.

For the merge process, HPDK maintains a merge candidate set of SSTables. During database startup, a background process is configured to maintain this set and judge when metadata changes (flush operation) occur. If the ratio of valid items to total items in a single SSTable reaches a preset threshold, the SSTable will be added to the merge candidate set. When the size of the merge candidate set reaches the predefined threshold, the merge process will be run.

We have a background thread that does the merge process. Algorithm 1 is the pseudo-code that describes the merging process. The input of this algorithm is the merge candidate set. Lines 1-6 check whether the SSTable in the merge candidate set contains valid key-address items. If all the key-address items in the SSTable are invalid, the SSTable can be freed directly without any other changes. Lines 7-9 means that if the merge candidate set is empty, this algorithm can be finished.

Algorithm 1 Live-item Rate Based Dynamic Merge Algorithm

```
Input: merge_candidate: dynamic merge candidate set
 1: for pool in merge_candidate do
 2:     if pool.valid == 0 then
 3:         pool.clear();
 4:         merge_candidate.remove(pool);
 5:     end if
 6: end for
 7: if merge_candidate.size == 0 then
 8:     return ;
 9: end if
10: result = merge_sort(merge_candidate);
11: iter = MakeInputIterator(result);
12: iter.SeekToFirst();
13: while iter.valid() do
14:     item = BTreeSearch(iter.key);
15:     if item != nullptr and item.addr == iter.value then
16:         write to new SSTable;
17:         update B+-tree;
18:     end if
19:     iter.next();
20: end while
21: InstallMergeResult();
22: merge_candidate.clear();
```

If the merge candidate set is not empty, SSTables in the merge candidate set should be reordered. since the key-address pairs in a single SSTable are ordered, lines 10 and 11 use merge sorting to sort multiple SSTables and create an iterator that can traverse all key-address items. Lines 12 position the iterator to the first item, and lines 13-20 handle each key-address item. The pseudo-code shows that for each key-address item in SSTable, we first have a search operation for the key in the B^+-tree. If the key exists and the address is the same as the address of the currently processed key-address items, the key-address item is valid and then written to a new SSTable.

When the key-address item is written to a new SSTable, the SSTable pool number of the corresponding item stored in the B^+-tree is changed, which will lead to data inconsistency in the database if the B^+-tree is not updated. So we write the item to a new SSTable while updating the $Addr$ part which is shown in Fig. 3 to keep the data in the B^+-tree consistency. After all key-address items are processed, the metadata of SSTables will be updated in lines 21 and 22, including SSTable additions and deletions, metadata update, merge candidate set updates, old space free, etc. Since our algorithm updates the data in the B^+-tree during handling every key-address item, if there are changes to the valid key-address items of the new SSTable during the merging process, it will also make the metadata incorrect. We add a check operation when updating the metadata to solve this problem. When a write operation to the new SSTable occurs before the merge process result installs, we will record these operations and add a check to the corresponding metadata in the InstallMergeResult() function on line 21 to process the related metadata when updating them.

In our merge algorithm, suppose that the number of SSTables in the merge candidate set is k, the number of key-address pairs in each SSTables is n, and the number of key-address items in B$^+$-tree is m. Then we need to search n times in the B$^+$-tree for each merge operation. Therefore, the complexity of our algorithm is $O(k \cdot n \cdot log(m))$. Because k is always a constant, then the time complexity of our merge algorithm is $O(n \cdot log(m))$. It means that the time complexity of our merging algorithm is related to the key-address items in the merge candidate set and the total amount of valid persistent data in the system.

In summary, HPDK stores all persistent data in PM to obtain high I/O throughput, reduces the write amplification by compressing the level of LSM-Tree, and reduces the read amplification to make the read faster by keeping a B$^+$-tree in DRAM. Then, we have a key-address separation design in HPDK to further reduce the write amplification and the total amount of data in DRAM. Finally, we design a merge algorithm for SSTables, which continuously merges those SSTables whose valid key-address items are below our predefined threshold during the HPDK life cycle to improve PM's space utilization.

IV. Implementation

To verify the efficiency of HPDK's design strategies, we implement HPDK based on LevelDB(version 1.23) [7], a state-of-the-art LSM-Tree-based KV store from Google. We use the Persistent Memory Development Kit (PMDK) [27] interface to access a 128 GB Intel AEP Persistent Memory. PMDK builds on the Direct Access (DAX) feature available in Linux and Windows, allowing applications to directly access Persistent Memory by memory-mapping files on a PM-aware file system.

A. Key-Address Store Operations

HPDK provides a standard set of interfaces for clients, including GET/PUT/DELETE [7]. The numbering in Fig. 2 indicates the data flow for those interface.

GET(k) means to obtain the value with key k in the KVS. To get the value for the given key k from KVS, HPDK does GET_1, GET_2, and GET_3 that in Fig. 2 in order. GET_1 searches in the current MemTable in DRAM, GET_2 searches on Immutable MemTable in DRAM, if k does not exist, GET_3 is executed. GET_3 searches in the B$^+$-tree in DRAM. If any phase finds k successfully, the corresponding value is retrieved from the value area in PM according to the read address and returned to the client. If any of the three phases does not find k, it is returned to the client as '$notfound$'. During the whole execution of the GET(k), HPDK returns the result to the client without reading any SSTable.

PUT(k, v) means to write a key-address pair (k, v) to the KVS. It does INSERT(k, v) insert the pair (k, v) to the KVS if the key k does not exist and UPDATE(k, v) update the value of key k to v if it does. To write a given key-address pair (k, v) to KVS, the data flow inside HPDK is represented in Fig. 2 as ① - ⑧. ① receives a PUT operation for a given key-address pair (k, v). ②,③ writes the value v to the value area and returns the address that includes the location information of the written value v, and replaces v with the address for subsequent operations. ④ log the now operation to the WAL for failure recovery in case of a system crash. ⑤ put the pair (k, $addr$) as (k, v) in MemTable in DRAM and returns the result of the current operation. ⑥ - ⑧ is the background data flow. When the MemTable is full, ⑥ set the MemTable to Immutable MemTable and create a new MemTable to continue receiving writes. Once there is an Immutable MemTable in the DRAM, the flush process of Immutable MemTable is triggered. ⑦ flush Immutable MemTable to level 0 in PM as SSTable and write it to the B$^+$-tree in DRAM at the same time. When SSTable in level 0 has too many invalid items, ⑧ merge algorithm of SSTable will be triggered to merge multiple SSTables where invalid items exceeding the predefined threshold into one SSTable, to improve the space utilization of PM.

DELETE(k) means to remove the key-address pair, where the key (k) is from the KVS. It can be implemented by PUT(k, v) by setting the v to a delete flag.

B. Crash Recovery

HPDK provides crash-consistency guarantees for data in PM and DRAM similar to those in LevelDB. Like LevelDB, the data recently written to WAL in HPDK is not committed (i.e., fsync()) by default because a commit is expensive. However, the latency of the data commit per write in PM is smaller than in SSD, so if a strong consistency guarantee is needed, HPDK can also achieve commit per write with less overhead based on PM. If it is a normal shutdown during recovery, we can simply rebuild the whole system using the MANIFEST file records.

HPDK utilizes the LevelDB recovery mechanism. Compared with LevelDB, HPDK adds more SSTable metadata information. Such as the merge candidate list (including the add, delete, and update of the list), the number of valid key-address pairs in each SSTable, and the total number of key-address pairs in each SSTable. In addition, HPDK adds the recovery information of B$^+$-tree and the value area. The recovery information of B$^+$-tree is all key-address pairs information in all leaf nodes. The recovery information for the value area is the number of valid records, the total number of records contained in each value pool, and the current write position of each pool. Information about the SSTable is incrementally recorded in the MANIFEST file. The information of the B$^+$-tree and the value area is recorded in separate files. Since the B$^+$-tree is relatively large and very expensive to record, the B$^+$-tree information will only be recorded after merging the SSTable.

When the system starts, it will first check the MANIFEST file and the recovery files of B$^+$-tree and value area to determine whether the system exited normally last time. If it is a normal exit, HPDK uses the MANIFEST file to execute the recovery procedure like LevelDB. For B$^+$-tree and value area, HPDK will reconstruct it based on the recovery file record. If the system is an abnormal exit, HPDK will first recover the normal data part, just like the normal exit, and then re-execute the records in WAL to recover the whole system except the B$^+$-tree. Regarding the B$^+$-tree, HPDK will first rebuild part of the B$^+$-tree using the recovery file and then read the SSTable sequentially, which is generated after the recovery file is last recorded

to rebuild the whole B$^+$-tree. Since all SSTables belong to the same level, the key-address pair in the newer SSTable is also newer, so the B$^+$-tree will be consistent with before. HPDK also detects if a merge process has not been completed, if so, HPDK must restart the merge. When a merge process begins, we create a merge log file with a record in it, and the merge log file will be deleted after the merge process. For PM data corruption caused by hardware errors, data recovery, and fault tolerance features such as data replication and erasure coding can be used [28].

V. Evaluation

HPDK is a PM-based key-value store (KVS) with high throughput. In this section, we first evaluate the read and write throughput of three representative LSM-Tree based KVSs: LevelDB [7], NoveLSM [10], SLM-DB [21], HPDK-KS (HPDK with only the key-value separation of our four techniques), and HPDK using db_bench micro-benchmark [7]. Next, we assess LevelDB and HPDK's throughput using the YCSB macro-benchmark [14] suite. Finally, we examine the impact of the other three techniques of HPDK. All experiments are executed as single-threaded workloads as LevelDB is not optimized for multi-threaded workloads (HPDK is implemented based on LevelDB). We deploy HPDK on Optane DC Persistent Memory.

A. Methodology

Our experiments use a machine with an Intel Xeon Platinum 8222L CPU (3.0GHz), a Samsung 981A SSD of 512G, 32GB DRAM, and Intel AEP Persistent Memory of 128G. Ubuntu 20.04.2 LTS with Linux kernel version 5.8 is used for the machine. We use a 128GB PM to run HPDK. We employ PMDK [27] to manage PM as a layout with some pool and configure a DAX-enabled ext4 file system to access PM. In the default setting, every PM pool size is set to 8MB.

We evaluate the throughput of HPDK and compare it with NoveLSM and LevelDB(version 1.23) over varying value size and fixed key size with 16 Byte. The size of MemTable is set to 4MB to suit the PM pool size, and a fixed key size of 16 bytes is used. Note that in HPDK, all SSTable files are stored on PM. For LevelDB, all parameters are set to default values. For HPDK, The live-item rate threshold is set to 0.5, if we increase this threshold, HPDK will perform dynamic merge more actively. In all experiments, NoveLSM, LevelDB and HPDK are with the same setting.

B. Results With PM in LevelDB Directly

To evaluate the efficiency of PM in HPDK, we first run a modified version of LevelDB, i.e. LevelDB + PM, which directly uses PM as a block device and access it with a standard POSIX interface to replace SSD without other modifications as in HPDK. We run the modified version of LevelDB with a fixed key size of 16 bytes with a varying value size of 100 Byte and 1 KB, respectively. As in Fig. 4, we can observe that the write throughput of LevelDB+PM is 52% – 157% higher when the value size is from 1 KB to 16 KB, and the read throughput is slightly (4% to 10%) lower than LevelDB. So directly using PM as a block device to store SSTable files instead of SSD can get a positive result to write throughput because PM has low write latency. So we run LevelDB in PM in the following experiment. In Fig. 4, the throughput is normalized to those of LevelDB.

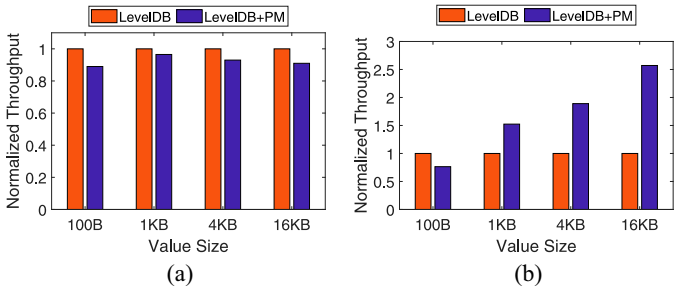

Fig. 4. Read and write throughput comparison. *The x-axis represents various value sizes, and the y-axis represents the normalized throughput to LevelDB with the same setting. (a) Read. (b) Write.*

C. Results With Micro-Benchmarks

Fig. 5 shows the operation throughput with HPDK for random read, random write, and overwrite workloads compared to LevelDB, NoveLSM and SLM-DB. To evaluate the influence of key-value separation, we also have a modified version of HPDK, i.e. HPDK-KS, which only has the key-value separation of the four techniques of HPDK. For all KVS, we first do a random write workload to create a database and then do a random read in that database. Then, we open the database that creates in random write and writes the same data as the random write workload does to the database for the overwrite workload, all workloads run with 1,000,000 items. Every workload starts after the previous one is finished avoiding other effects. From the results we make the following observations:

For a random read operation, compared to LevelDB and SLM-DB, HPDK consistently has higher performance ranging from 25% to 11× when the value size is from 100 Byte to 16 KB, respectively. Although compared to NoveLSM, the read throughput of HPDK is lower when the value size is 100 Byte, with the increase of value size, the throughput of HPDK is much higher. For the random write operation, HPDK provides higher throughput than LevelDB, NoveLSM and SLM-DB for values sizes from 100B to 16KB. Compared to these three approaches, HPDK provides up to 36.4× throughput improvement. For overwrite workload, HPDK has similar throughput with other three methods when the value size is 100 Byte. But its performance surges with the increasing of value size and it outperforms other three methods by up to 45.3× throughput improvement.

The reason that HPDK outperforms other methods is the implementation of the key-value separation scheme. From the throughput of HPDK-KV, we can see that the key-value separation design of HPDK prevents the system throughput from being affected by the value size of the key-value pair. Thus HPDK can provide a significant throughput improvement of I/O operations when the value size increases. However, as shown

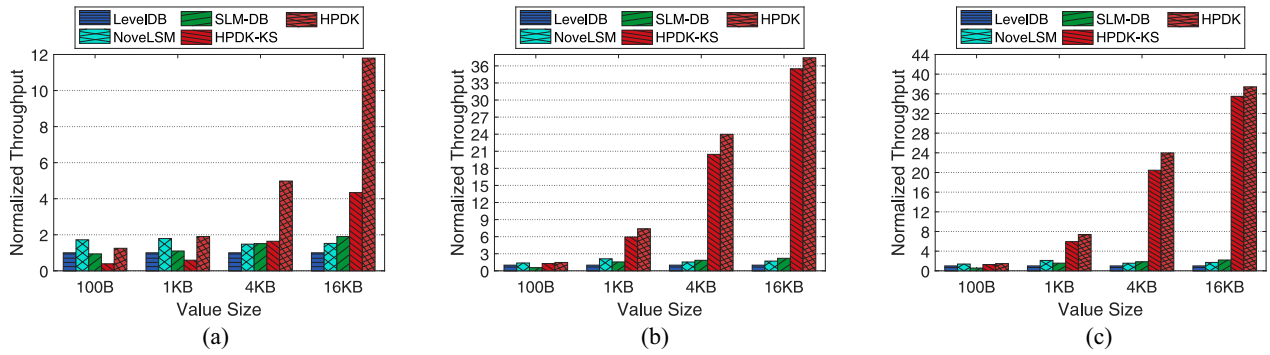

Fig. 5. Normalized throughput of HPDK compares to LevelDB, NoveLSM and SLM-DB with the same setting for db_bench. *The x-axis represents various value sizes, and the y-axis represents the normalized throughput to LevelDB. HPDK-KS is HPDK only with the key-value separation of our four techniques. (a) Random read. (b) Random write. (c) Overwrite.*

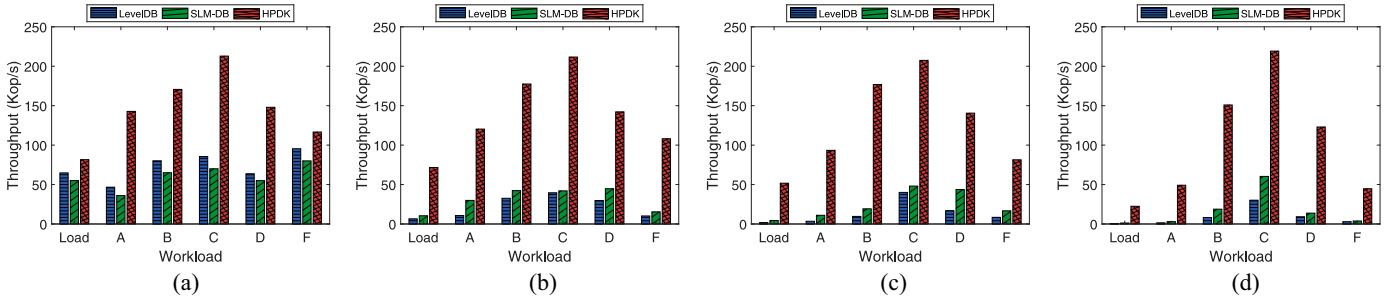

Fig. 6. YCSB throughput of HPDK compares to LevelDB and SLM-DB in various value sizes with the same setting. *The x-axis represents data loading and different workloads, and the y-axis represents throughput (Kop/s). (a) Value size: 100 Byte. (b) Value size: 1 KB. (c) Value size: 4 KB. (d) Value size: 16 KB.*

in Fig. 5(c), the throughput of HPDK decreases in overwrite workloads compared to HPDK-KS. This decrease is primarily due to HPDK's dynamic merging algorithm. In the overwrite workload, HPDK must run the dynamic merge all the time, saving disk space but reducing the throughput. In LevelDB, writing duplicate key-address pairs do not affect the number of compression operations, so HPDK has about 7% throughput degradation compared with HPDK-KS in overwrite workload.

D. Results With YCSB

Yahoo! Cloud Serving Benchmark (YCSB) [14] is a tool for the basic testing of KVSs. It provides a framework and a basic set consisting of six workloads close to real to the real scene to evaluate the throughput of KVSs. We use YCSB to compare the throughput of HPDK, LevelDB and SLM-DB in a 20GB database with the value of 100 Byte, 1 KB, 4 KB, and 16 KB, respectively.

Fig. 6 shows the throughput of HPDK with different value sizes. Since the original version of LevelDB does not implement range query, we do not run workload E. From Fig. 6, we can learn that HPDK has better throughput than LevelDB in data loading and the five workloads. When the value size is 100 Byte, the throughput of HPDK is more than twice that of LevelDB and SLM-DB. As the value size increases, the throughput improvement becomes obvious. This is mainly because the key-value separation design of HPDK makes the effect of value size small. When LevelDB throughput decreases due to increase value size, the advantages of HPDK become noticeable.

In the experiment, we first load fixed-size data by inserting fixed quantity key-address pairs and then execute workloads A, B, C, D, and F in sequence. Workload A composed of 50% reads, 50% updates, workload B performs 95% reads and 5% updates, workload C performs 100% reads, workload D composed of 95% reads and 5% inserts and workload F performs 50% reads and 50% read-modify-writes. For these workloads, A, B, C, and F use $zipfian$ distribution, and workload D performs all read for the latest keys.

From Fig. 6(a), we can learn that when the value size is 100 Byte, the throughput of HPDK is higher than LevelDB in data loading and all workloads, in data loading, the throughput of HPDK is 25.87% higher than LevelDB. In workload A, workload B, workload C, workload D, and workload F, the throughput of HPDK are 205.89%, 112.95%, 148.85%, 132.31%, and 22.21% respectively, higher than LevelDB. Fig. 6(b), 6(c), and 6(d) shows that when the value size increases, the throughput improvement of HPDK compared to LevelDB was increased, in data loading, the throughput of HPDK is 9×–46× higher than LevelDB, and 10×–32×, 4×–17×, 4×–6×, 3×–12× and 9×–14× higher in workload A, B, C, D, and F respectively when value size from 1KB increase to 16KB.

Table II shows the comparison of average latency and 99th percentile latency between HPDK and LevelDB in the YCSB workloads with different value sizes, from which we can see

TABLE II
AVERAGE AND 99THPERCENTILE LATENCY (μs) OF YCSB BENCHMARK ON HPDK AND LEVELDB

		Workload A				Workload B				Workload C		Workload D				Workload F			
		Read		Update		Read		Update		Read		Insert		Read		Read		RMW	
		Ave	99th	Ave	99th	Ave	99th	Ave	99th	Ave	99th	Ave	99th	Ave	99th	Ave	99th	Ave	99th
100 B	LevelDB	16	86	27	1074	13	61	7	26	12	55	12	36	16	53	9	49	12	55
	HPDK	**7**	**12**	**7**	**11**	**6**	**23**	**8**	**25**	**5**	**10**	**11**	**20**	**6**	**13**	**6**	**11**	**12**	**21**
1 KB	LevelDB	42	219	146	1195	32	120	11	38	25	125	18	71	34	105	24	202	173	1236
	HPDK	**7**	**17**	**9**	**17**	**5**	**11**	**8**	**15**	**5**	**12**	**13**	**23**	**7**	**14**	**6**	**11**	**13**	**23**
4 KB	LevelDB	152	1494	428	1531	83	912	512	1564	25	109	35	103	61	451	30	196	210	1377
	HPDK	**7**	**15**	**14**	**18**	**5**	**12**	**11**	**17**	**5**	**13**	**16**	**25**	**7**	**12**	**6**	**18**	**19**	**24**
16 KB	LevelDB	218	3486	1132	1385	97	1062	615	1705	33	140	519	1472	87	730	49	237	636	1672
	HPDK	**7**	**15**	**34**	**29**	**5**	**11**	**31**	**30**	**5**	**12**	**42**	**63**	**6**	**14**	**6**	**13**	**39**	**42**

The average and 99thPercentile latency(μs) of YCSB benchmark on HPDK and LevelDB. RMW means Read-Modify-Write.

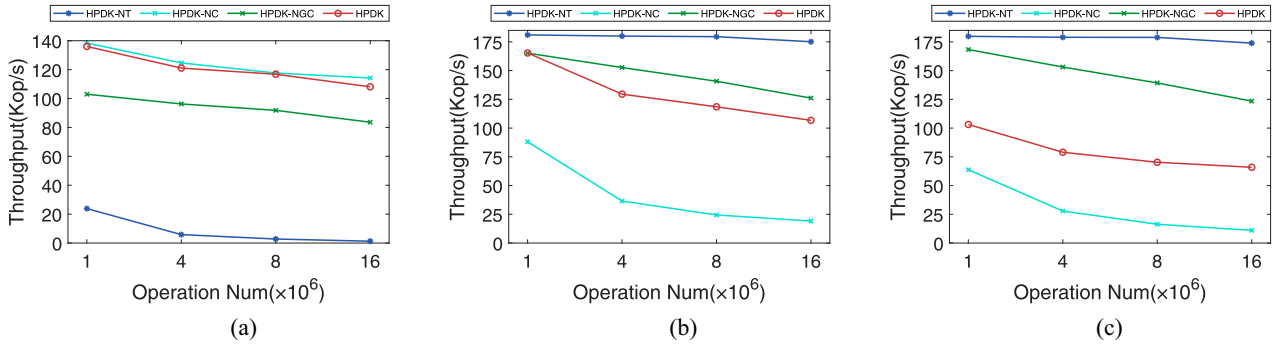

Fig. 7. Impact of different techniques in HPDK on system throughput. *The x-axis represents the operation number, and the y-axis represents the throughput (Kop/s). HPDK-NT is HPDK without the B$^+$-tree, HPDK-NC is HPDK without level compression, and HPDK-NGC is HPDK without the dynamic merge algorithm. (a) Random read. (b) Random write. (c) Overwrite.*

that the average latency and tail latency of HPDK in each workload is much lower than that of LevelDB. In HPDK, only the write latency is affected by the change of value size because the persistence time of large values is higher than that of small ones. Benefitting from the design of HPDK, the value size does not affect the read latency.

E. Impact of Different Techniques

We use four techniques in HPDK to improve the throughput of various aspects of the system: level compression, in-memory B$^+$-tree, key-value separation, and live-item rate based dynamic merge algorithm. We evaluated the throughput optimization of key-value separation on the system in Section V-C, so in this section, we compare the impact of the other three techniques on HPDK. Since the value size in key-address pairs has almost no effect on HPDK, we fix the value size to 100 Byte with a fixed key size of 16 Byte and use increasing operation number to evaluate the change in system throughput with growing data.

The variation of system throughput with the number of operations is shown in Fig. 7. HPDK-NT is HPDK without the B$^+$-tree, HPDK-NC is HPDK without level compression, and HPDK-NGC is HPDK without the dynamic merge algorithm.

From Fig. 7(a), we can observe that the in-memory B$^+$-tree in HPDK significantly improves the read throughput of the system, and the throughput improvement increases from 5 times to 85 times as the operation number increase from 1,000,000 to 16,000,000. The throughput improvement is mainly because the level compression design in HPDK makes the read latency high. It makes the SSTable all located at level 0, which means that if a read operation is not found in memory, it needs to read all the SSTable until it finds the current items. The in-memory B$^+$-tree only needs one search operation in the tree. Another technique to improve the system read throughput is the dynamic merging algorithm in HPDK, mainly achieved by reducing the number of useless SSTables, thus reducing the latency of a single read operation from the side. The dynamic merging algorithm in HPDK improves the read throughput by about 30% and is not affected by the number of operations increase. The impact of level compression in HPDK on the system read throughput is negligible, only 1% to 4%, which is also caused by the compact of HPDK-NLC reducing a certain number of SSTables.

From Fig. 7(b), we can observe that the improvement of write throughput in HPDK is mainly due to the technique of level compression. The actual low throughput in HPDK-NLC is primarily because the compact operation will cause a write stall,

increasing write operations' latency. The level compression of HPDK allows the system to perform garbage collection using the dynamic merge algorithm with less overhead without a compact process which significantly improves the write throughput of the system. However, the dynamic merging algorithm in HPDK also degrades the system throughput somewhat, which mainly depends on the execution frequency of the merging algorithm. From Fig. 7(c), the dynamic merging algorithm degrades the overall system throughput by about 45% in the least HPDK-friendly overwrite workload. The write throughput impact of B^+-tree in memory in HPDK is Larger as the number of operations increases, and the throughput decrease increase from 8% to 39% as the operation number increase from 1,000,000 to 16,000,000.

For the overwrite workload, as shown in Fig. 7(b) and 7(c), the throughput of HPDK-NGC and HPDK-NT is almost the same as the random write operation. However, this same throughput means that the throughput of the overwrite workload is reduced mainly by the dynamic merging algorithm and the in-memory B^+-tree in HPDK. The impact of the dynamic merge algorithm on the throughput primarily comes from the resource consumption of the merge process. In the merging process, HPDK needs to search and update the B^+-tree, which consumes a lot of system resources and thus reduces the system throughput. Furthermore, the dynamic merge algorithm will be triggered frequently in overwrite workloads, resulting in a significant reduction in system throughput. The overwrite workload also reduces the throughput of HPDK-NLC, mainly because the overwrite workload trigger more background compact processes than random writes.

In summary, all four techniques in HPDK have different impacts on system throughput; level compression improves the throughput of random write. Still, it reduces read throughput by eliminating the compact operations background. In-memory B^+-tree significantly improves read throughput but requires additional memory space and affects write throughput. Key-value separation improves the read throughput and the write throughput for large values. Finally, the dynamic merge algorithm improves the read throughput of the system while saving hard disk space. Still, it has a particular impact on the write throughput, and the more frequent the trigger, the more significant the effect on the system, mainly because the merge process requires a search and update operation for the B^+-tree.

VI. Related Work

HPDK is a high I/O throughput KVS based on a hybrid PM-DRAM system, which aims to achieve high throughput and low write amplification. We first discuss existing works on PM. NoveLSM [10] uses PM as a middle layer to store level 0 and level 1 data, but it is also employs the architecture of LevelDB, resulting in higher write amplification compared to HPDK. SLM-DB [21] uses PM as DRAM removes the WAL of LevelDB and directly stores the MemTable on PM. It maintains a B^+-tree index, and has a single-level architecture like HPDK. However, it does not use DRAM to store data, which affects system throughput. SLM-DB also performs garbage collection by compacting SSTables with a high overlap rate, leading to higher overhead than HPDK. HiKV [9] maintains a hybrid index in a hybrid memory system of DRAM and PM for a KVS and supports rich key-value operations. HiKV maintains a hash index in PM for efficient reading and writing and a B^+-tree index in DRAM to support range query. However, it does not improve write throughput, and the resizing overhead for the hash index is also high. uDepot [29] maintains an index structure in DRAM and leverages the throughput of PM devices to realize a cache server with throughput close to DRAM at a lower cost. FlatStore [25] matches the persistence granularity in PMs, decouples the KVS into a volatile index and log-structured storage, and manages the index and small-sized KVs with a per-core OpLog to better support batching. Bullet [30] uses cross-referencing logs to smooth the throughput differences between PM and DRAM through the producer-consumer model, making their throughput similar. PapyrusKV [8] is designed for distributed HPC architecture. LevelHash [19] optimizes the hash for the PM to reduce the amount of writing.

Next, we discuss research aiming at improving LSM-Trees throughput by reducing the write amplification caused by compaction. PebblesDB [15] reduces write amplification by decreasing data rewriting rather than dividing data into small blocks. It divides SSTables into blocks at each level, ensuring that blocks do not overlap. SSTables can be directly divided during compaction and put into different blocks according to the key range, reducing or eliminating writes during compaction. Like HPDK, WiscKey [23] also employs key-value separation, reducing write amplification during compression when the value size is large. HashKV [31] is optimized based on WiscKey,using key-value separation and adding hash-based data grouping, dynamic reserved space allocation, hotness awareness, and selective key-value separation. This optimizes throughput for update-intensive workloads and garbage collection. Both WiscKey and HashKV partially reduce write amplification, but they still perform expensive background compaction operations competing with clients. LWC-Tree [16] eliminates I/O amplification by appending data to SSTables and only merging metadata. It retains aggregated metadata of the underlying overlapping tables in each SSTable, reducing the small random disk reading in the execution process. SlimDB [22] redesigns the index block to a three-level compact index, using a Multi-Level Cuckoo Filter, and proposes a stepped-merge algorithm to optimize read, write throughput, and memory usage for semi-sorted data. GearDB [17] achieves high throughput and high space utilization through a new on-disk data layout that reorganizes every zone for one level, a compaction window performing gear compaction, and the gear compaction performing garbage collection-free on HM-SMR. TRIAD [18] reduces the time and space required for compaction and flush operations by partitioning hot and cold data, delaying the SSTable compaction to enough overlap between SSTables, and changing the role of WAL in LSM-Tree and using it like SSTable. SILK [32] reduces high tail latency without affecting the original throughput of the system by prioritizing deal operations close to DRAM.

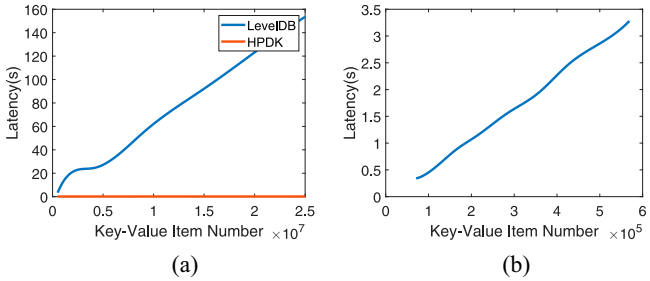

Fig. 8. Limitations of HPDK. *The x-axis represents key-address item number, and the y-axis represents the latency(s). (a) The startup latency of HPDK and LevelDB. (b) The latency of rebuild B$^+$-tree in DRAM in HPDK.*

Kreon [33] uses random I/O instead of CPU overhead and I/O amplification. It uses an index within each level to eliminate the need for sorting large segments and employs a custom memory-mapped I/O path to reduce the cost of I/O.

Finally, several studies [2], [24], [34], [35], [36] provide high throughput for KVS with different methods. Other works [12], [37] are based on B$^+$-tree or its variants, proposing optimization schemes or optimal persistent data structures. HPDK, based on PM, modifies the LSM-Tree and combines it with B+-tree to achieve high throughput, reducing the amount of data written while attaining high read and write throughput.

VII. Limitations

HPDK is a KVS based on PM with high throughput, low latency, and low write/read amplification. However, Since it keeps a B$^+$-tree in DRAM, it needs to rebuild it in DRAM every time it is started, which will consume some time. When HPDK adopts the design of key-value separation, the rebuilding time of B$^+$-tree is only related to the number of key-address items. As shown in Fig. 8(a), The startup time increases linearly with the number of key-address items. When the number of valid key-address items in KVS is 25,000,000, it takes about 2.5 minutes to restart the whole HPDK, while for LevelDB, the system startup is only related to the amount of data in WAL, which is less than the amount of data in one SSTable. Additionally, to evaluate the additional latency required to recover the B$^+$-tree after a crash, we also tested the time to rebuild the B$^+$-tree from SSTable. As shown in Fig. 8(b), as with normal startup, the time required to rebuild the B$^+$-tree in DRAM is proportional to the number of key-address items. When we rebuild about 570,000 key-address items from 8 SSTables to the B$^+$-tree in DRAM, it takes about 3.28 seconds, which is also close to the time we need to rebuild the B$^+$-tree during normal startup.

The cost of PM (Persistent Memory) is another constraint in implementing our method, as it is significantly higher compared to NVMe (Non-Volatile Memory Express). However, our method is well-suited for data centers where high performance holds greater importance than cost. For instance, a military data center would be a suitable choice for implementing our method.

HPDK is the first that adopts a hybrid LSM-Tree based KVS architecture which totally replacing Disk by PM. Also HPDK improves the slow read performance which exists in traditional LSM-Tree based KVS.

VIII. Conclusion

This paper presents HPDK, a hybrid LSM-Tree based KVS architecture designed for PM-DRAM environments. HPDK is designed to achieve high throughput through low read and write latency using four techniques: level compression for LSM-Tree in PM, key-value separation design, a B$^+$-tree index in DRAM, and a merge algorithm for SSTable in Persistent Memory. We implement HPDK on a real Persistent Memory. Experimental results show that HPDK outperforms LevelDB and SLM-DB in read and write throughput. For example, when value sizes are 100 Bytes, 1 KB, 4 KB, and 16 KB, HPDK improves read and write throughput by up to 36.4×. These results confirm the effectiveness of our design for high I/O throughput in PM-DRAM systems.

References

[1] X. Wu, Y. Xu, Z. Shao, and S. Jiang, "LSM-trie: An LSM-tree-based ultra-large key-value store for small data items," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC 15)*, 2015, pp. 71–82.
[2] J. Chen et al., "HotRing: A hotspot-aware in-memory key-value store," in *Proc. 18th USENIX Conf. File Storage Technol. (FAST 20)*, 2020, pp. 239–252.
[3] R. Sears and R. Ramakrishnan, "bLSM: A general purpose log structured merge tree," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2012, pp. 217–228.
[4] F. Chang et al., "BigTable: A distributed storage system for structured data," *ACM Trans. Comput. Syst. (TOCS)*, vol. 26, no. 2, pp. 1–26, 2008.
[5] B. Zhang, H. Gong, and H. C. David, "PMDB: A range-based key-value store on hybrid NVM-storage systems," *IEEE Trans. Comput.*, vol. 72, no. 5, pp. 1274–1285, May 2023.
[6] S. Chen, Y. Liang, and M. Yang, "KVSTL: An application support to LSM–tree based key-value store via shingled translation layer data management," *IEEE Trans. Comput.*, vol. 71, no. 7, pp. 1598–1611, Jul. 2022.
[7] LevelDB, "A fast and lightweight key/value database library," 2011. [Online]. Available: https://github.com/google/leveldb
[8] J. Kim, S. Lee, and J. S. Vetter, "PapyrusKV: A high-performance parallel key-value store for distributed NVM architectures," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2017, pp. 1–14.
[9] 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. (USENIX ATC 17)*, 2017, pp. 349–362.
[10] S. Kannan, N. Bhat, A. Gavrilovska, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Redesigning LSMs for nonvolatile memory with NoveLSM," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC 18)*, 2018, pp. 993–1005.
[11] H. Chen, C. Ruan, C. Li, X. Ma, and Y. Xu, "SpanDB: A fast, cost-effective LSM-tree based KV store on hybrid storage," in *Proc. 19th USENIX Conf. File Storage Technol. (FAST 21)*, 2021, pp. 17–32.
[12] H. Cha, M. Nam, K. Jin, J. Seo, and B. Nam, "B3-tree: Byte-addressable binary B-tree for persistent memory," *ACM Trans. Storage (TOS)*, vol. 16, no. 3, pp. 1–27, 2020.
[13] Y.-P. Liang, T.-Y. Chen, C.-H. Chi, H.-W. Wei, and W.-K. Shih, "Enabling a B+-tree-based data management scheme for key-value store over SMR-based SSHD," in *Proc. 57th ACM/IEEE Des. Automat. Conf. (DAC)*, Piscataway, NJ, USA: IEEE Press, 2020, pp. 1–6.
[14] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. 1st ACM Symp. Cloud Comput.*, 2010, pp. 143–154.
[15] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham, "PebblesDB: Building key-value stores using fragmented log-structured merge trees," in *Proc. 26th Symp. Operating Syst. Princ.*, 2017, pp. 497–514.
[16] T. Yao et al., "A light-weight compaction tree to reduce I/O amplification toward efficient key-value stores," in *Proc. 33rd Int. Conf. Massive Storage Syst. Technol. (MSST)*, 2017, pp. 1–13.

[17] T. Yao et al., "GearDB: A GC-free key-value store on HM-SMR drives with gear compaction," in *Proc. 17th USENIX Conf. File Storage Technol. (FAST 19)*, 2019, pp. 159–171.
[18] O. Balmau et al., "TRAID: Creating synergies between memory, disk and log in log structured key-value stores," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC 17)*, 2017, pp. 363–375.
[19] P. Zuo, Y. Hua, and J. Wu, "Write-optimized and high-performance hashing index scheme for persistent memory," in *Proc. 13th USENIX Symp. Operating Syst. Des. Implementation (OSDI 18)*, 2018, pp. 461–476.
[20] J. Yang, D. B. Minturn, and F. Hady, "When poll is better than interrupt," in *Proc. 10th USENIX Conf. File Storage Technol. (FAST)*, vol. 12, 2012, pp. 3–3.
[21] O. Kaiyrakhmet, S. Lee, B. Nam, S. H. Noh, and Y.-r. Choi, "SLM-DB: Single-level key-value store with persistent memory," in *Proc. 17th USENIX Conf. File Storage Technol. (FAST 19)*, 2019, pp. 191–205.
[22] K. Ren, Q. Zheng, J. Arulraj, and G. Gibson, "SLIMDB: A space-efficient key-value storage engine for semi-sorted data," *Proc. VLDB Endowment*, vol. 10, no. 13, pp. 2037–2048, 2017.
[23] L. Lu, T. S. Pillai, H. Gopalakrishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Wisckey: Separating keys from values in SSD-conscious storage," *ACM Trans. Storage (TOS)*, vol. 13, no. 1, pp. 1–28, 2017.
[24] L. Marmol, S. Sundararaman, N. Talagala, and R. Rangaswami, "NVMKV: A scalable, lightweight, FTL-aware key-value store," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC 15)*, 2015, pp. 207–219.
[25] Y. Chen, Y. Lu, F. Yang, Q. Wang, Y. Wang, and J. Shu, "FlatStore: An efficient log-structured key-value storage engine for persistent memory," in *Proc. 25th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2020, pp. 1077–1091.
[26] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *Proc. 12th ACM SIGMETRICS/Perform. Joint Int. Conf. Meas. Model. Comput. Syst.*, 2012, pp. 53–64.
[27] PMem.io, "Persistent memory programming," 2024. [Online]. Available: https://pmem.io/pmdk/
[28] Y. Hu, S. Song, S. Xiao, Q. Xu, N. Xiao, and Z. Qin, "A dominating error region strategy for improving the bit-flipping LDPC decoder of SSDs," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 62, no. 6, pp. 578–582, Jun. 2015.
[29] K. Kourtis, N. Ioannou, and I. Koltsidas, "Reaping the performance of fast NVM storage with udepot," in *Proc. 17th USENIX Conf. File Storage Technol. (FAST 19)*, 2019, pp. 1–15.
[30] 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. (USENIX ATC 18)*, 2018, pp. 967–979.
[31] H. H. Chan et al., "HashKV: Enabling efficient updates in KV storage via hashing," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC 18)*, 2018, pp. 1007–1019.
[32] O. Balmau, F. Dinu, W. Zwaenepoel, K. Gupta, R. Chandhiramoorthi, and D. Didona, "SILK: Preventing latency spikes in log-structured merge key-value stores," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC 19)*, 2019, pp. 753–766.
[33] A. Papagiannis, G. Saloustros, P. González-Férez, and A. Bilas, "An efficient memory-mapped key-value store for flash storage," in *Proc. ACM Symp. Cloud Comput.*, 2018, pp. 490–502.
[34] B. Lepers, O. Balmau, K. Gupta, and W. Zwaenepoel, "KVell: The design and implementation of a fast persistent key-value store," in *Proc. 27th ACM Symp. Operating Syst. Princ.*, 2019, pp. 447–461.
[35] J. Im et al., "PinK: High-speed in-storage key-value store with bounded tails," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC 20)*, 2020, pp. 173–187.
[36] A. Anwar et al., "Customizable scale-out key-value stores," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 9, pp. 2081–2096, Sep. 2020.
[37] D. Hwang, W.-H. Kim, Y. Won, and B. Nam, "Endurable transient inconsistency in byte-addressable persistent B+-tree," in *Proc. 16th USENIX Conf. File Storage Technol. (FAST 18)*, 2018, pp. 187–200.

Bihui Liu received the bachelor's degree in accounting from the Central South University, Changsha, China, in 2014, and the master's degree in accounting from the University of Miami, USA, in 2016. She is currently working toward the Ph.D. degree in energy and power with the College of Information Science and Engineering, Hunan University, Changsha, China. Her research interests include cyber security, data security, and artificial intelligence. She wants to find better ways to solve the complicated problems of data security for companies and for ordinary people.

Zhenyu Ye received the B.E. degree in network engineering from the College of Information Engineering, Xiangtan University, China, in 2018. He is currently working toward the Ph.D. degree with the College of Computer Science and Electronic Engineering, HNU of computer science and technology. His research interests include key-value store and system security.

Qiao Hu received the B.S. degree from Hunan University, in 2011, the M.Sc. degree from Wuhan University, in 2013, and the Ph.D. degree from the Department of Computer Science, City University of Hong Kong, in 2017. Now he works as an Assistant Professor with Hunan University. His research interests include RFID security and privacy, cloud computing, and wireless communication security.

Yupeng Hu (Senior Member, IEEE) received the M.S. and Ph.D. degrees in computer science from Hunan University, China, in 2005 and 2008, respectively. He was the Dean with the Department of Cyberspace Security. He was with the National University of Defense Technology as a Postdoctoral Researcher, from 2011 to 2016. He was with the Department of Computer Science and Engineering, UT-Arlington, as a Visiting Scholar, from 2015 to 2016. He was also with the IBM China Development Laboratory as an Academic Visitor, in 2012. He is currently a Professor with the College of Computer Science and Electronic Engineering, Hunan University. His research interests include big data and storage systems security, erasure coding, AI security, and network and system security. He has published more than 80 journal articles, book chapters, and refereed conference papers.

Yuchong Hu (Member, IEEE) received the B.S. degree in computer science and technology from the School of the Gifted Young, University of Science and Technology of China, Anhui, China, in 2005, and the Ph.D. degree in computer science and technology from the School of Computer Science, University of Science and Technology of China, in 2010. He is currently an Associate Professor with the School of Computer Science and Technology, Huazhong University of Science and Technology. His research interests include improving data reliability (e.g., erasure coding) and data reduction (e.g., deduplication) in cloud storage, and big data storage systems.

Yang Xu (Member, IEEE) received the Ph.D. degree in computer science and technology from Central South University, China. From 2015 to 2017, he was a Visiting Scholar with the Department of Computer Science and Engineering, Texas A& M University, USA. He is currently an Associate Professor with the College of Computer Science and Electronic Engineering, Hunan University, China. His research interests include distributed computing, cloud computing, blockchain, artificial intelligence, and trustworthy/privacy computing. He has published over 50 articles in international journals and conferences, including IEEE TRANSACTIONS ON SERVICES COMPUTING, IEEE TRANSACTIONS ON INTELLIGENT TRANSPORTATION SYSTEMS, IEEE TRANSACTIONS ON INDUSTRIAL INFORMATICS, IEEE TRANSACTIONS ON CLOUD COMPUTING, IEEE TRANSACTIONS ON EMERGING TOPICS IN COMPUTING, IEEE/ACM TRANSACTIONS ON COMPUTATIONAL BIOLOGY AND BIOINFORMATICS, and IEEE TRANSACTIONS ON NETWORK SCIENCE AND ENGINEERING. He was the awardee of the Best Paper Award of IEEE International Conference on Internet of People (IoP 2018).

Keqin Li (Fellow, IEEE) is a SUNY Distinguished Professor with the State University of New York. He is among the world's top five most influential scientists in parallel and distributed computing in terms of single-year and career-long impacts based on a composite indicator of the Scopus citation database. He received the IEEE TCCLD Research Impact Award from the IEEE CS Technical Committee on Cloud Computing in 2022 and the IEEE TCSVC Research Innovation Award from the IEEE CS Technical Community on Services Computing in 2023. He won the IEEE Region 1 Technological Innovation Award (Academic) in 2023. He is a Member of the SUNY Distinguished Academy. He is an AAAS Fellow, an AAIA Fellow, and an ACIS Founding Fellow. He is an Academician Member of the International Artificial Intelligence Industry Alliance. He is a Member of Academia Europaea (Academician of the Academy of Europe).