



# Revisiting Log-Structured Merging for KV Stores in Hybrid Memory Systems

Zhuohui Duan
Huazhong University of Science and Technology
Wuhan, 430074, China
zhduan@hust.edu.cn

Jiabo Yao
Huazhong University of Science and Technology
Wuhan, 430074, China
jiaboyao@hust.edu.cn

Haikun Liu
Huazhong University of Science and Technology
Wuhan, 430074, China
hkliu@hust.edu.cn

Xiaofei Liao
Huazhong University of Science and Technology
Wuhan, 430074, China
xfliao@hust.edu.cn

Hai Jin
Huazhong University of Science and Technology
Wuhan, 430074, China
hjin@hust.edu.cn

Yu Zhang
Huazhong University of Science and Technology
Wuhan, 430074, China
zhyu@hust.edu.cn

## ABSTRACT

We present MioDB, a novel LSM-tree based *key-value* (KV) store system designed to fully exploit the advantages of byte-addressable *non-volatile memories* (NVMs). Our experimental studies reveal that the performance bottleneck of LSM-tree based KV stores using NVMs mainly stems from (1) costly data serialization/deserialization across memory and storage, and (2) unbalanced speed between memory-to-disk data flushing and on-disk data compaction. They may cause unpredictable performance degradation due to write stalls and write amplification. To address these problems, we advocate byte-addressable and persistent skip lists to replace the on-disk data structure of LSM-tree, and design four novel techniques to make the best use of fast NVMs. First, we propose one-piece flushing to minimize the cost of data serialization from DRAM to NVM. Second, we exploit an elastic NVM buffer with multiple levels and zero-copy compaction to eliminate write stalls and reduce write amplification. Third, we propose parallel compaction to orchestrate data flushing and compactions across all levels of LSM-trees. Finally, MioDB increases the depth of LSM-tree and exploits bloom filters to improve the read performance. Our extensive experimental studies demonstrate that MioDB achieves 17.1× and 21.7× lower $99.9^{th}$ percentile latency, 8.3× and 2.5× higher random write throughput, and up to 5× and 4.9× lower write amplification compared with the state-of-the-art NoveLSM and MatrixKV, respectively.

## CCS CONCEPTS

• **Computer systems organization** → **Heterogeneous (hybrid) systems**; • **Information systems** → **Storage management**.

## KEYWORDS

Non-Volatile Memory, Key-Value Store, Log-Structured Merge, Skip List, LSM-tree Compaction

**ACM Reference Format:**
Zhuohui Duan, Jiabo Yao, Haikun Liu, Xiaofei Liao, Hai Jin, and Yu Zhang. 2023. Revisiting Log-Structured Merging for KV Stores in Hybrid Memory Systems. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3575693.3575715

## 1 INTRODUCTION

*Log-Structured Merge* (LSM) tree based key-value stores, such as BigTable [4], LevelDB [16], HBase [2], and RocksDB [13] have become a critical component in modern data-center applications. LSM-tree based KV stores stage write operations in memory and flush them to SSD/HDD in a batch, and thus enforce sequential writes to achieve high throughput and low latency. However, these KV stores often suffer from high *Write Amplification* (WA) [8, 36], periodical write stalls [3, 43], and barely satisfactory read performance [1, 46], because LSM-tree based KV stores organize multiple levels of on-disk files where KV pairs are compacted (i.e., merge-sorted) many times across multiple levels of LSM-trees. Although data compaction reduces data redundancy and improves the performance of querying, it may cause severe WA and write stalls that may periodically degrade application performance to nearly zero. Previous studies have mainly focused on reducing WA or improving the throughput of LSM-trees for SSDs [15, 26, 37, 39]. However, there remain challenges to solve these problems completely in traditional DRAM/SSD systems.

The emergence of NVM technologies such as Intel Optane *DC Persistent Memory Modules* (DCPMMs) [18] provides vast opportunities to build low-latency, high-throughput, and cost-efficient KV stores. NVMs offer DRAM-like performance, disk-like persistence,

[*]Zhuohui Duan and Jiabo Yao contributed equally to this work. Haikun Liu is the corresponding author. All authors are with National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China.


*ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada*

ACM ISBN 978-1-4503-9916-6/23/03…$15.00
https://doi.org/10.1145/3575693.3575715

high density, low cost per bit, near-zero standby power consumption, and byte-addressability [9, 11, 12, 35]. Compared with SSD, NVMs show extremely lower access latency (up to 100×) and much higher bandwidth (up to 10×) [21, 27]. These features make NVM a promising candidate to replace SSDs, and offer new opportunities to optimize the performance of LSM-tree based KV stores.

There have been some recent studies on exploiting NVMs for LSM-tree based KV stores [20, 21, 25, 43]. SLM-DB [20] maintains a single level LSM-tree and a $B^+$-tree index in NVM to improve the performance of read and write operations. NoveLSM [21] redesigns the storage architecture of LevelDB, and uses NVM as an extension of DRAM buffer for on-disk SSTables [4]. MatrixKV [43] replaces the first level of LSM-tree with a matrix-like data structure in NVMs, and exploits a column compaction scheme to reduce periodical write stalls. Although these earlier efforts are able to improve the performance of LSM-trees with NVMs, they can not completely solve **a critical problem: the data serialization and compaction in traditional LSM-trees are too slow, and thus may block data flushing from DRAM to NVM**. As a result, when burst writes are accumulated in the DRAM buffer and the top level of the LSM-tree, applications would suffer periodical write stalls. These stalls often result in long tail latency and severe performance fluctuation, and thus may violate *service level agreements* (SLAs) of many latency-sensitive applications. On the other hand, data compaction across all levels of LSM-trees also causes severe WA which also degrades the system performance and aggravates worn-out of storage devices.

In this paper, we first make comprehensive experimental studies on state-of-the-art LSM-tree based KV stores using NVMs, i.e., NoveLSM [31] and MatrixKV [27]. Based on our observations, we reveal that the root cause of write stalls is mainly attributed to the performance bottleneck of the data compaction in the top two levels of the LSM-tree. To address this problem, we propose MioDB, a LSM-tree based KV store designed to fully exploit the advantages of byte-addressable NVMs. We replace the on-disk data structure (SSTable) in LSM-trees with multi-level skip lists to manage KV pairs in the NVM. We propose four novel techniques to best utilize the byte-addressable NVM in MioDB. First, we propose one-piece flushing to minimize the cost data serialization from DRAM to NVM. As the NVM and the DRAM buffer use the same data structure, MioDB only needs to update pointers in skip lists during the one-piece flushing, and thus significantly reduces the cost of data relocation. Second, we design an elastic NVM buffer with a LSM-like structure to handle burst writes. This multi-level elastic buffer enables fast and asynchronous skip-list compactions to minimize write stalls. Unlike traditional LSM-trees, the capacity of each level in MioDB is not limited. Thus, data compaction involved in the top two levels would not block data flushing from DRAM to NVM. To reduce the compaction cost and WA, we propose two-stage data compactions including a zero-copy compaction and a lazy-copy compaction. In the former stage, MioDB merges small skip lists into larger ones by only updating pointers in skip lists, without moving KV pairs. In the latter stage, MioDB copies and inserts all fresh KV pairs into a large skip list, and performs garbage collection to reclaim memory consumed by outdated KV pairs. Third, we propose parallel compaction to orchestrate data flushing and compactions across all levels of LSM-trees, so that data can flow into the bottom level quickly. Finally, MioDB increases the depth of LSM-tree and exploits bloom filters to further improve the read performance of LSM-trees. Overall, we have made the following contributions.

- We design multi-level (LSM-like) skip lists to fully utilize the byte-addressable NVM for KV stores, and exploit one-piece flushing to reduce the cost of data serialization/deserialization across DRAM and NVM.
- We propose zero-copy compaction in combination with lazy-copy compaction to reduce write stalls and WA.
- We propose parallel compaction to orchestrate data flushing and compactions across all levels of LSM-trees, and thus further reduce write stalls and improve the query performance.
- We implement MioDB based on LevelDB and open the source code [28]. We evaluate MioDB with extensive workloads. Experimental results show that MioDB achieves 17.1× and 21.7× lower $99.9^{th}$ percentile access latencies, 8.3× and 2.5× higher random write throughput, and up to 5× and 4.9× lower write amplification compared with state-of-the-art NoveLSM [31] and MatrixKV [27], respectively.

The remaining of this paper is organized as follows. Section 2 presents the background of NVMs and LSM-trees, and then introduces two typical LSM-tree based KV stores using NVMs, i.e., NoveLSM and MatrixKV. Section 3 analyzes the limitations of NoveLSM and MatrixKV, and then presents our design principles. Section 4 describes our design and implementation of MioDB. Section 5 presents experimental results. Section 6 introduces the related work, and we conclude in Section 7.

## 2 BACKGROUND

Here, we first introduce NVMs and LSM-trees, and then discuss limitations of existing LSM-tree based KV stores using NVMs.

### 2.1 Non-Volatile Memory

The advent of Intel Optane DCPMM [18] has finally made NVMs commercially available. They can offer much larger capacity than DRAM at lower cost and energy consumption [41]. Its performance is about two orders of magnitude higher than SSD [21, 27]. Despite the advantages of Intel Optane DCPMM, there is still significant performance gap between DRAM and NVM [18, 41]. We use Flexible I/O Tester (FIO v3.7) [14] to measure the performance of DRAM and NVM, and find that Intel Optane DCPMM shows much lower sequential/random bandwidth than DRAM for both read/write operations. Particularly, the random write throughput of Intel Optane DCPMM is almost 7 times lower than that of DRAM [41]. Thus, when KV store systems are deployed in hybrid memory systems, it is still beneficial to stage write requests in a DRAM buffer to avoid random writes to the slow NVM. *Log-structured Merge Tree* (LSM-tree), which is widely used in DRAM-SSD architectures, is still reasonable for hybrid DRAM/NVM systems [10, 38, 45].

### 2.2 Log-Structured Merge Tree

*Log-Structured Merge Tree* (LSM-tree) is a write-friendly block-based data structure [33] implemented in many popular KV stores, such as LevelDB [16], RocksDB [13], and PebblesDB [36]. Taking the typical LevelDB as an example, we illustrate its structure in Figure 1(a). LevelDB buffers KV updates in a sorted skip list (called MemTable) that resides in fast and volatile DRAM, and then stores KV pairs in

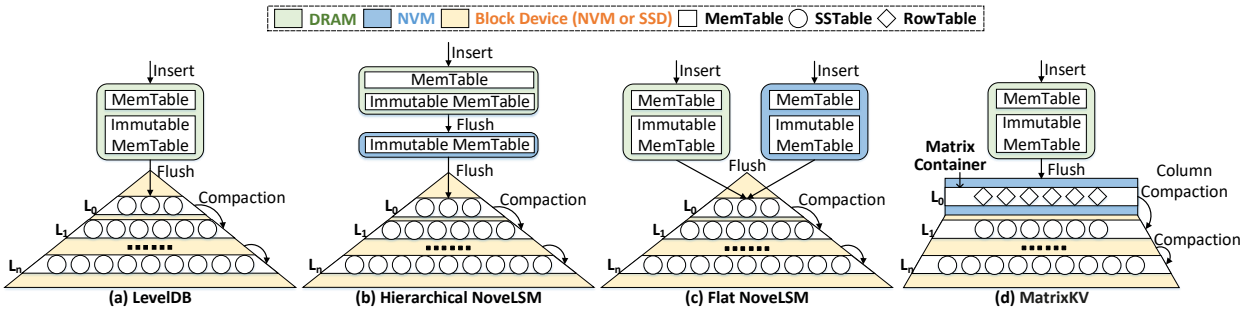


**Figure 1: The storage hierarchy of LSM-tree based key-value stores using NVM**

multiple levels of on-disk data structures called *Sorted String Table* (SSTable). A skip list is a multidimensional linked list that enables fast searches and in-place updates without traversing all elements in the linked list [34]. Popular LSM-tree based KV stores all use skip lists to stage writes in MemTables. For failure recovery, a KV pair should be first appended to a *write-ahead log* (WAL) before it is written to the MemTable. When the MemTable is full, LevelDB makes it immutable and creates a new MemTable. A background thread flushes data in the immutable MemTable to an on-disk SSTable. In this way, LSM-trees optimize write performance of KV stores by converting random writes into a batch of sequential writes on block devices. Each level in the LSM-tree has limited capacity, but a lower level contains about 10 times more SSTables than its previous level. SSTables are compacted (sort-merged) from level $L_0$ to levels ($L_1$, $L_2$,..., $L_n$) when the size of a level grows beyond its limit. For a read operation, the data is searched from the DRAM buffer and the LSM-tree hierarchically, and thus the read latency increases with the number of levels.

## 2.3 LSM-Tree Based KV Stores Using NVMs

Although LevelDB and its extensions (e.g., RocksDB) offer relative high throughput of write operations, they still suffer from periodical write stalls and costly WA [43]. Write stalls are mainly caused by data compacting from $L_0$ to $L_1$, and data flushing from immutable MemTable to on-disk SSTables [21, 43]. WA stems from data compaction across multiple levels, and increases with the number of levels in LSM-trees. Recently, there have been a few efforts [21, 43] to bridge the performance gap between DRAM and SSD in LSM-trees by using NVMs. We elaborate these proposals in the following.

**NoveLSM** [21] proposes two storage architectures to use NVM in a LSM-tree based KV store. The first architecture uses NVM as a secondary buffer to receive immutable MemTables from DRAM, as shown in Figure 1(b). We thereby call this architecture as hierarchical NoveLSM. KV pairs in the DRAM-based immutable MemTable are first flushed into the large NVM-based MemTable one by one. As the NVM-based MemTable is also a large skip list, this architecture avoids data serialization during data flushing from DRAM to NVM. However, when the large NVM-based MemTable is flushed into SSD, the KV store still suffers from serialization/deserialization costs. The second architecture uses NVM as an extension of DRAM buffer to enlarge the capacity of MemTables, as shown in Figure 1(c). We thereby call this architecture as flat NoveLSM. It makes NVM-based MemTable mutable so that inserts are directly updated to the persistent skip list in-place. This design avoids logging KV updates in the persistent disk and reduces WA. However, these two architectures of NoveLSM only defer flushing MemTables to SSTables. When the NVM-based MemTables become full, the $L_0$ of LSM-tree is quickly filled up with MemTables flushed from NVM. The costly $L_0$-to-$L_1$ compaction may block the data flushing, and even causes longer write stalls.

**MatrixKV** [43] is a state-of-the-art LSM-based KV store using NVMs. It focuses on reducing application write stalls and WA. As shown in Figure 1(d), MatrixKV redesigns the persistent data structure of the level $L_0$ in LSM-trees, and proposes a fine-grained compaction scheme to mitigate write stalls. MatrixKV reorganizes data flushed from the skip lists and stores it in a matrix called matrix container. MatrixKV only compacts some columns of multiple rows in the matrix container, and thus substantially reduces the amount of data compacted. Although MatrixKV can improve the speed of $L_0$-to-$L_1$ compaction and mitigate the possibility of write stalls, the costly data serialization during flushing MemTables may still cause write stalls. On the other hand, a large size of $L_0$ increases the cost of data deserialization upon read operations. Moreover, the design choice of wider levels in the LSM-tree may degrade the read performance.

Although these studies have made earlier efforts to use NVMs for LSM-tree based KV stores, they have not addressed the write stall and WA problems completely because the byte addressability of NVM is not fully exploited during data flushing and compaction in LSM-trees.

# 3 OBSERVATIONS AND DESIGN PRINCIPLES

In this section, we first analyze key issues of state-of-the-art LSM-tree based KV stores [27, 31] using hybrid memories, and then present our design principles of MioDB.

## 3.1 Key Observations

To understand the challenges of using hybrid memories for LSM-tree based KV stores, we conduct a set of experiments to evaluate state-of-the-art NoveLSM [31] and MatrixKV [27] on a server described in Section 5. We choose the flat NoveLSM in our experiments because its performance is better than the hierarchical NoveLSM [31]. We use a total 80 GB dataset containing about 20 million KV pairs of the same size. For each KV pair, the key size is 16 bytes and the value size is 4 KB. Both NoveLSM [31] and MatrixKV [27] are evaluated under the same system settings.

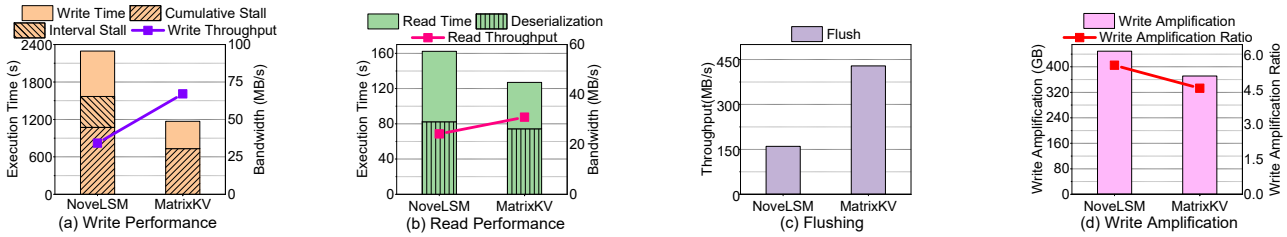


**Figure 2: The execution time of write/read, throughput of flushing, and WA in NoveLSM and MatrixKV**

There are two kinds of stalls during MemTable flushing. If the immutable MemTable is still being flushed while the MemTable becomes full, new write requests are not served until the remaining data in the immutable MemTable are completely flushed to SSTables in the $L_0$. These stalls observed by applications is called interval stalls [13]. To avoid long-term interval stalls, KV stores usually limit the speed of writes by intentionally creating a short period of stall between write requests. The total time of these short-period stalls is called cumulative stalls [13].

Figure 2(a) shows the total execution time of writing the dataset to NoveLSM and MatrixKV. For NoveLSM, both interval stalls and cumulative stalls are amazingly long because the large-size MemTable (4 GB) in the NVM significantly increases the cost of flushing. As the $L_0$-to-$L_1$ compaction is rather slow, and often blocks the MemTable flushing, causing a long period of write stalls. MatrixKV speeds up the MemTable (64 MB) flushing by optimizing the $L_0$-to-$L_1$ compaction, and eliminates the interval stalls. As a result, the write throughput of MatrixKV is significantly improved by near two times compared with NoveLSM. However, the cumulative stall still accounts for 62.5% of total write time in MatrixKV. This implies there is a vast room to improve the write throughput of KV stores.

Figure 2(b) shows the total execution time of reading one million KV pairs from NoveLSM and MatrixKV. Since MatrixKV reduces the depth of LSM-tree and provides on-DRAM indexes for the matrix container ($L_0$), its read performance is higher than that of NoveLSM. However, both NoveLSM and MatrixKV need to deserialize data in the SSTables for read operations. The deserialization costs are as high as 50.7% and 58.6% of the total read time for NoveLSM and MatrixKV, respectively. This implies that data deserialization is the root cause of poor read performance in LSM-tree based KV stores.

Figure 2(c) shows the throughput of data flushing when writing the dataset to NoveLSM and MatrixKV. MatrixKV can flush data much faster than NoveLSM because data copying from DRAM to NVM is more efficient than that from DRAM to SSD. We believe that the cost of data serialization can be further reduced by making the data structure of LSM-trees compatible with the MemTable.

Figure 2(d) shows the *write amplification* (WA) caused by SSTable compactions in NoveLSM and MatrixKV. The WA ratio is defined as the ratio of on-disk data traffic to the data volume written by users. NoveLSM and MatrixKV show 6.6× and 5.6× WA, respectively. This implies that there is still a large room to reduce WA for LSM-tree based KV stores.

The above observations demonstrate that the performance bottleneck of LSM-trees are mainly attributed to data flushing from MemTables to SSTables and $L_0$-to-$L_1$ compactions. Clearly, SSTables are no longer efficient for byte-addressable NVMs due to the following reasons. (1) Data serialization (MemTable-to-SSTable) during flushing has a significant impact on the write throughput of KV stores. (2) Read operations also cause costly data deserialization from SSTables to main memory. (3) Data compaction across multiple levels of the LSM-tree results in severe WA. Both data serialization/deserialization and compaction lead to periodical application stalls and degrade the performance of KV stores. As a result, it is essential to redesign the persistent data structure in LSM-trees for hybrid memory systems.

## 3.2 Design Principles

MioDB exploits NVMs' byte addressability, large capacity, and persistence to build a high-performance KV store. As the random write bandwidth of NVM is an order of magnitude lower than that of DRAM, we still use a small size of DRAM as a write buffer to achieve high throughput and low latency, but design an elastic NVM buffer to enable in-memory data compaction. Based on the above observations, we advocate several design principles in MioDB to fully exploit the advantages of NVMs.

**Principle 1: the in-NVM data structure should be compatible with the DRAM buffer to minimize the cost of data serialization/deserialization.** To fully exploit the byte-addressability of NVMs, we advocate persistent skip lists in NVMs (called PMTables) to replace on-disk SSTables, so that all levels of LSM-trees in the NVM have the same data structure with the DRAM-based MemTable. In this way, the costly data serialization/deserialization can be avoided during the flushing and read operations. We can significantly improve the speed of flushing and $L_0$-to-$L_1$ compaction, and thus eliminate the performance bottleneck of LSM-tree based KV stores.

**Principle 2: data flushing should be orchestrated with the $L_0$-to-$L_1$ compaction so that interval stalls can be completely eliminated.** In our design, the $L_0$-to-$L_1$ compaction of skip lists becomes very fast because it only needs to update pointers in the skip lists without moving data. Thus, the PMTable flushing should be as quickly as the $L_0$-to-$L_1$ compaction. In NoveLSM, although the cost of serialization/deserialization is significantly reduced during data flushing, the KV pairs in the small DRAM-based MemTable should be copied and inserted into a very large skip list in the NVM one by one. It is often costly to find the insertion location and copy KV pairs with multiple *memcpys*. There is still room to further reduce the cost of persisting the skip lists to NVM.

**Principle 3: data compaction between skip lists should be fast enough and do not cause WA problems.** The byte-addressability feature of NVMs provides an opportunity to directly

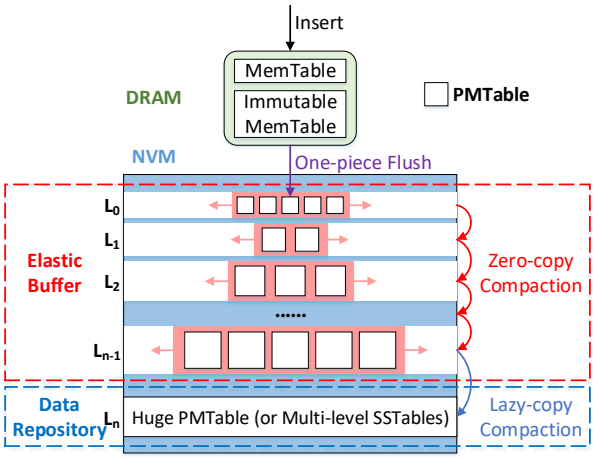


**Figure 3: The architecture of MioDB**

compact overlapping skip lists by only updating the pointers, without moving or updating KV pairs. Thus, the compaction does not cause WA. However, we should carefully orchestrate data compactions across all levels so that a $L_i$-to-$L_{i+1}$ compaction would not block the $L_{i-1}$-to-$L_i$ compaction. It is essential to perform PMTable compactions in parallel for all levels, and eliminate write stalls even for write-intensive workloads.

**Principle 4: write performance optimizations should not compromise the read performance of KV stores.** Although LSM-tree based KV stores are designed particularly to optimize the write performance, the read performance should not be overlooked. Using a single and large skip list in the NVM can offer good read performance, however, it is often costly to insert a KV pair into the large skip list because of sorting. In contrast, maintaining multiple small skip lists in a single level allows the MemTable flushing very quickly, but degrades the read performance. Thus, it is essential to make a tradeoff between the performance of reads and writes.

## 4 DESIGN AND IMPLEMENTATION

In this section, we present the design and implementation details of MioDB.

### 4.1 MioDB Architecture

Figure 3 shows the system architecture of MioDB with a DRAM-NVM hierarchy. Similar to previous LSM-based KV stores, MioDB also uses DRAM-based MemTables to stage write requests. However, MioDB replaces traditional SSTables with skip lists (PMTables) to manage persistent data in the NVM. Here, we still use a tree-like structure to illustrate PMTable compactions across multiple levels. However, the logical structure of MioDB is completely different from LSM-trees. In the NVM, MioDB maintains an elastic buffer composed of multiple PMTables in levels $L_0$ to $L_{n-1}$, and a data repository in which a huge PMTable or multi-level SSTables are used to store data during a lazy-copy compaction.

Unlike LSM-trees in which the number of SSTables in each level is limited, MioDB can accommodate unlimited PMTables in each level, without blocking the one-piece flushing or cross-level PMTable compactions. However, if one level contains too many small and unsorted PMTables, the performance of range queries may become poor. To address this problem, MioDB performs zero-copy compactions for all levels except $L_n$ in parallel. The zero-copy compaction only changes pointers of duplicate nodes in skip lists, without moving KV pairs themselves. The outdated data is reclaimed after the lazy-copy compaction. Thus, zero-copy compaction can significantly reduce the compaction cost and WA. For each level, a thread compacts two PMTables into a larger PMTable and inserts it into the underlying level. Thus, the logical sizes of PMTables in each level are usually different. The zero-copy compaction in each level can be performed in parallel. This allows PMTables flow into $L_{n-1}$ quickly. For PMTables in $L_{n-1}$, we copy all KV pairs and insert them into $L_n$, and then reclaim memory consumed by outdated KV pairs in $\{L_0, L_1, ..., L_n\}$. We call this $L_{n-1}$-to-$L_n$ compaction as a lazy-copy compaction. Finally, the huge PMTable stores all unique and sorted KV pairs.

Unlike NoveLSM [31], MioDB uses multi-level skip lists instead of a single big skip list as the elastic buffer because it is very costly to merge a MemTable (a small skip list) into a big skip list. For example, if an 80 GB dataset is stored in a big skip list, there are total $80GB/4KB = 20M$ data nodes. For each insertion/update, $log(20M) \approx 24$ nodes are searched in the worst case. Assume a MemTable contains $m$ KV pairs, at most $24 * m$ lookups and $m$ `memcpy`s are required if these KV pairs are flushed into the big skip list. These random NVM references can significantly slow down the MemTable flushing, resulting in periodical write stalls and long tail latency. Moreover, when the big skip list is flushed to the $L_0$ of the LSM-tree finally, the tail latency of writes may increase significantly. Because the $L_0$-to-$L_1$ compaction is very slow in traditional LSM-trees, it may block the flushing of the big skip list due to limited storage space available in $L_0$. This is the reason why NoveLSM only uses a moderate-sized skip list (4 GB) in the NVM as a buffer for on-disk SSTables.

Since MioDB uses a huge PMTable as the data repository in the bottom to store compacted KV pairs finally, a fundamental challenge should be addressed to eliminate write stalls during the flushing, i.e., **how to merge a moderate-sized (64 MB) MemTable into the huge PMTable in a very short time**? To address this problem, MioDB exploits multi-level skip lists as a buffer of the huge PMTable, and copies the entire MemTable to $L_0$ rapidly via a single `memcpy` (called one-piece flushing). Then, the costly merging operation is postponed to the zero-copy compaction stage which can be performed asynchronously with the one-piece flushing. In this way, the latency of skip-list merging is hidden by the background compaction in multiple levels, and we have a potential to completely eliminate write stalls.

**Advantages of multi-level PMTables.** The multi-level elastic buffer offers many opportunities to optimize the performance of KV stores. First, the multi-level architecture enables fast data flushing from the DRAM buffer. Second, unlike traditional LSM-trees in which the cross-level SSTable compaction is performed only when a level becomes full, MioDB can perform the zero-copy compaction once there are two PMTables in each level, without suffering the decision-making cost of selective compaction [20]. Third, since the PMTable compaction in each level is completely independent of other levels, it is possible to enable zero-copy compactions in

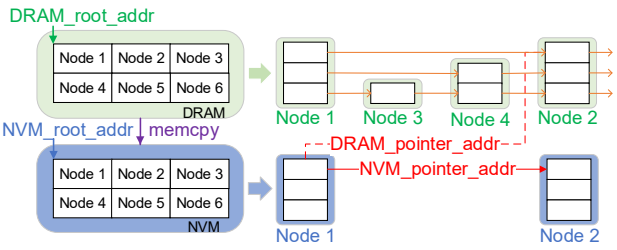


**Figure 4: An illustration of one-piece flushing**

parallel. Fourth, the parallel PMTable compaction allows a large amount of small PMTables flow into the large PMTables in the lower levels quickly. Because querying a key in a single large PMTable is much faster than querying it in multiple small PMTables, this design can also improve the performance of querying in MioDB. At last, since there are multiple PMTables in different levels, we have a potential to speed up querying by assigning bloom filters to PMTables.

**Supporting Memory/Storage Hierarchy.** Although the data repository can directly use NVMs to store the huge PMTable (a skip list) for high performance, large-capacity SSDs are still expected to co-exist with NVMs in existing LSM-tree based KV stores for recent years. To provide backward compatibility for many existing applications, we extend MioDB to support the DRAM-NVM-SSD hierarchy. We can also use multi-level SSTables to replace the huge PMTable, and still exploit the traditional approach in LSM-trees for data compaction. In this architecture, the elastic buffer can still sustain I/O bursts to eliminate most write stalls, and substantially reduce write amplification through zero-copy compaction before the data is finally flushed into the SSD. Since it is often slow to flush PMTables to SSD, a number of relatively large PMTables may be accumulated in $L_{n-1}$ of the NVM buffer. Although KV pairs in each PMTable are sorted, there are often overlapping key ranges among different PMTables. Since a read or scan operation may have to query all PMTables in $L_{n-1}$ in the worst case, the tail latency of reads is roughly linear with the number of PMTables retrieved. Thus, burst writes also have a negative impact on the performance of reads. However, when these PMTables accumulated in $L_{n-1}$ have been flushed to multi-level SSTables in the SSD, the latency of reads would become stable.

## 4.2 One-Piece Flushing

When the MemTable in the DRAM is full, we should make it immutable and flush it to the NVM. In traditional LSM-tree based KV stores, KV pairs in the MemTable should be serialized and copied to the on-disk SSTable one by one. Although the hierarchical NoveLSM uses a big skip list in NVMs to store data flushed from the DRAM-based MemTable, it is very costly to find the insertion location and to merge KV pairs into the big skip list one by one, as described in Section 4.1.

We propose one-piece flushing to persist immutable MemTables into NVMs efficiently. In MioDB, we allocate a large contiguous memory space of the same size for both DRAM MemTables and NVM PMTables. This offers an opportunity to physically copy a MemTable to a PMTable through one *memcpy*, without suffering the cost of data location and merging. However, the skip list contains many pointers linked to other data nodes, and these pointers still point to the data nodes in the DRAM (the dashed line) after the one-piece flushing, as shown in Figure 4. These pointers should be updated correctly in the persistent skip list. Because only the starting address of the MemTable is changed after the one-piece flushing, all data nodes in the PMTable have the same address offset relative to the MemTable. We can update all pointers in the PMTable according to the relative address.

Although the pointer swizzling during the one-piece flushing is costly, it can be performed in the background, without blocking read/write operations. When a MemTable becomes full, MioDB makes it immutable and creates a new MemTable to stage writes. A background thread flushes the immutable MemTable from DRAM to NVM, and then swizzles pointers in the PMTable. During this time, the original immutable MemTable in the DRAM can still serve read requests, and it is not reclaimed until the pointer swizzling is finished.

To eliminate the cost of pointer swizzling, an alternative choice is to use offset-based (relative) pointers in skip lists. This pointer uses an offset and its current location to calculate the destination address which it points to. However, because MioDB compacts skip lists by only updating pointers without moving data nodes, it still has to traverse multidimensional linked lists to calculate the relative address of each data node. Since using offset-based pointers may increase the cost of pointer updating during the zero-compaction, we still use absolute pointers in MemTables and perform the pointer swizzling in the background.

## 4.3 Zero-Copy Compaction

We exploit the byte-addressability feature of NVM and the data structure of skip lists to design a zero-copy compaction scheme for the elastic buffer. The compaction of PMTables only needs to update the pointers in skip lists without moving data nodes, and thus significantly reduces WA and the compaction cost.

We call existing tables in a level as oldtables, and a newly-inserted table as a newtable. In the elastic buffer, we select the oldest two PMTables in each level to compact at a time, and insert the merged PMTable into the next level. Figure 5(a) shows the initial state of a newtable and an oldtable. The data nodes in the PMTable are sorted by the *Key* in an ascending order. *Seq* represents the sequence number of the KV pair. For simplicity, we call a data node with *Key* $x$ and *Seq* $y$ as node $N_{xy}$. A larger sequence number implies the data is newer. Data nodes with the same *Key* are sorted by the *Seq* in a descending order. During the zero-copy compaction, we traverse the newtable and insert nodes into the oldtable one by one.

Figure 5(b) illustrates how a unique data node is merged into the oldtable. We take $N_{b6}$ as an example. First, we use an insertion mark (a pointer variable) to record the address of $N_{b6}$, and remove $N_{b6}$ from the newtable. The insertion mark allows $N_{b6}$ still readable during the compaction. Second, the pointers in the predecessor node of $N_{b6}$ (i.e., the head node) should point to the successor nodes of $N_{b6}$. Since the height of $N_{b6}$ is 2 and the successor nodes of each level are $N_{d7}$ and $N_{d5}$, the head node only needs to change the pointers of these two levels. Third, we traverse the oldtable

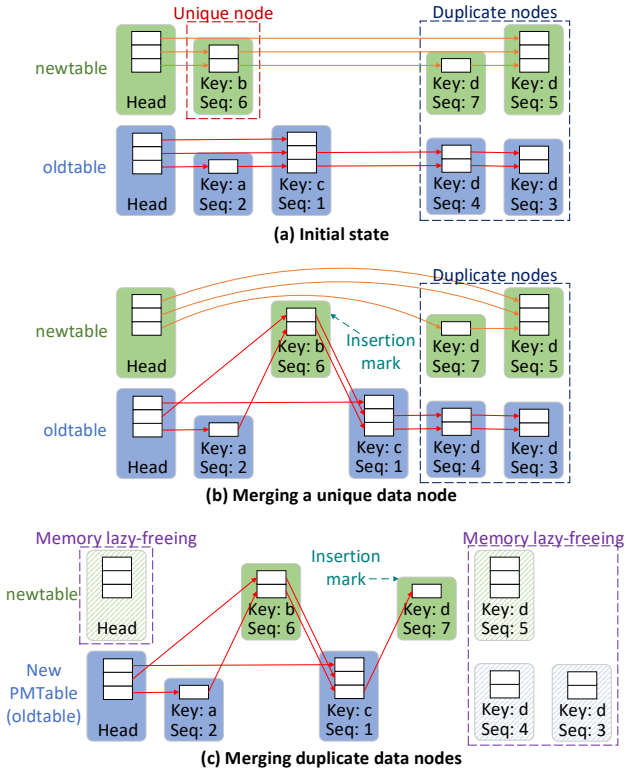


**Figure 5: An illustration of zero-copy compaction**

according to *Key* and *Seq* to find a node whose *Key* is larger than $N_{b6}$. In our example, this node is $N_{c1}$. We allow $N_{b6}$ to be a predecessor of $N_{c1}$, and then insert $N_{b6}$ into the oldtable successfully.

Figure 5(c) illustrates how duplicate data nodes are merged into the old table, which is deemed as a new PMTable in the next level. If a node in two PMTables has multiple versions, we call this node a duplicate node. After the node $N_{b6}$ is compacted, we continue to traverse the newtable and try to process $N_{d7}$. Since $N_{d5}$ has the same *Key* as $N_{d7}$ but is older, it should be "deleted" from the newtable directly. We first record the address of $N_{d7}$ in the insertion mark, and then the pointers in the predecessor node of $N_{d7}$ (i.e., the head node) should point to the successor nodes of $N_{d5}$. However, since $N_{d5}$ has not a successor node, the pointers of the head node in the newtable are set as NULL. Next, we insert $N_{d7}$ into the oldtable according to the *Key* and *Seq*. We traverse the oldtable to find a node whose *Key* is larger than $N_{d7}$, or its key is the same as $N_{d7}$ but has a smaller *Seq*. In our example, this node is $N_{d4}$. Thus, $N_{d7}$ is inserted in the front of $N_{d4}$. We then find potential redundant nodes after $N_{d7}$ in the oldtable. As $N_{d4}$ and $N_{d3}$ are older than $N_{d7}$, we also remove those nodes by simply setting the pointers in $N_{d7}$ as NULL. Finally, $N_{d7}$ is merged into the new PMTable, the older nodes $N_{d5}$, $N_{d4}$, $N_{d3}$ and the head node of the newtable are logically "deleted". At this time, the "newtable" becomes null and is marked as reclaimable. The memory resource consumed by the "deleted" nodes will be freed lazily after a lazy-copy compaction in the level $L_{n-1}$. This lazy memory freeing mechanism can substantially reduce the cost of data compaction.

**Supporting Concurrent Compaction and Queries.** To limit the impact of zero-copy compaction on the performance of queries, we exploit atomic writes to update pointers in a lock-free manner, without blocking concurrent queries on the two PMTables. However, since a data node usually involves multiple pointers in skip lists while an atomic write can only update a pointer once, there are two special cases that may affect the result of querying during the zero-copy compaction. We take $N_{b6}$ in Figure 5 as an example.

**Case 1**: *$N_{b6}$ has been picked from the newtable but not yet added to the oldtable.* At this time, if a query has already traversed $N_{c1}$ in the oldtable, even $N_{b6}$ will be inserted into the oldtable later, it cannot be found. To address this problem, we use an insertion mark to record the address of $N_{b6}$. When the newtable have been retrieved, we first consult the insertion mark and then traverse the oldtable. In this way, $N_{b6}$ can be found no matter it has been inserted into the oldtable or not.

**Case 2**: *when a query thread is still consulting $N_{b6}$ in the newtable, $N_{b6}$ is inserted into the oldtable immediately.* In this case, the next hop of the query may jump to $N_{c1}$ in the oldtable, and the successor nodes of $N_{b6}$ in the newtable will not be traversed. To solve this issue, we skip the node stored in the insertion mark when the newtable is being queried. Then, the query can go through the newtable entirely. The data node that is being compacted will be queried in the insertion mark or the oldtable.

Overall, to guarantee the accuracy of a query, the query thread should traverse the newtable, the insertion mark, and the oldtable sequentially. Thus, a query would never omit the data node that is being compacted. In this way, MioDB enables lock-free updating of pointers during the zero-copy compaction.

## 4.4 Lazy-Copy Compaction

Although the elastic buffer can effectively handle burst writes, the redundant data in PMtables across all levels consumes a considerable amount of NVM space. On the other hand, a high degree of data redundancy also degrades the performance of queries. We propose lazy-copy compaction to collect garbage (outdated KVs) in PMTables across all levels, and use a huge skip list as a data repository to store the remaining data. The lazy-copy compaction is conducted as follows: 1) We traverse the PMTables in $L_{n-1}$ one by one from the tail to the head. For each KV pair in a PMTable, if there is not an old value for the same key in $L_n$, we copy the node to a newly-created node and insert it into the $L_n$. We note that only the newest node in $L_{n-1}$ is copied, the others with the same key are deleted directly. 2) If there exists an older data node in the data repository, we in-place update it with the newest one in $L_{n-1}$. 3) Since data in $L_{n-1}$ is newer than that in $L_n$, we traverse the data repository from the insertion position and delete older nodes directly. 4) After all PMTables in $L_{n-1}$ are merged into $L_n$, we can reclaim the memory allocated to PMTables that are null (marked as reclaimable) in all levels.

Unlike the zero-copy compaction, the $L_{n-1}$-to-$L_n$ compaction should physically copy the newest KV pairs in the $L_{n-1}$ and insert them into the huge PMTable, and thus the cost of lazy-copy compaction is higher than that of zero-copy compaction. However, it does not cause write stalls because the multi-level elastic buffer can handle burst writes efficiently.

### 4.5 Parallel Compaction

To minimize write stalls and improve query performance, the data in each level should be merged into the underlying level as soon as possible. As we only compact PMTables in a single level, data compactions in different levels are independent. Thus, we propose parallel compaction to speed up data compaction for all levels.

Although RocksDB has already provided an earlier implementation of parallel compaction, it shows very limited impact on the reduction of write stalls. The root cause stems from the limited capacity of SSTables in each level. When the level $L_i$ is full, the $L_{i-1}$-to-$L_i$ compaction is blocked till the $L_i$-to-$L_{i+1}$ compaction is completed. On the other hand, SSTables in a single level of traditional LSM-trees are sorted and not overlapped, and thus a compaction often involves multiple SSTables in two adjacent levels. If there is an overlapping range of keys in two SSTables, one SSTable in the $L_i$ may involve in $L_{i-1}$-to-$L_i$ and $L_i$-to-$L_{i+1}$ compactions at the same time. Thus, two cross-level compactions cannot be performed concurrently. This conflict limits the parallelism of compaction.

In MioDB, the elastic buffer can guarantee non-blocking compaction because the capacity of each level is not limited. Moreover, because MioDB always chooses two oldest PMTables in a single level for compaction, the compaction in each level is completely independent of compactions in other levels. Thus, we can use one thread to perform compaction for each level in parallel. For levels between $L_0$ and $L_{n-2}$, once there are two PMTables in a single level, they are compacted immediately.

### 4.6 Read Optimizations

In each PMTable, data in the skip list are sorted. Thus, an intra-PMTable query is very fast. However, multiple PMTables often have overlapping key ranges even in a single level. A query thread has to retrieve all PMTables one by one in the worst case. To improve the read performance, we advocate two optimizations as follows.

1) *Increase the capacity of a single PMTable to exploit the benefit of fast intra-PMTable querying*. To achieve this goal, we increase the number of levels in MioDB. Because of the parallel compaction among $L_0$ and $L_{n-2}$, the size of PMTables increases exponentially with the number of levels after they are compacted. Traditional LSM-trees need to carefully configure the depth of levels to make a tradeoff between the WA and the performance of querying. In MioDB, because zero-copy compaction only updates the pointers in skip lists, a large number of levels introduce trivial WA while significantly enlarging the size of PMTable in the low level. This design can avoid querying data from multiple small PMTables, and achieve more benefit from fast querying a few large PMTables.

2) *Use bloom filters to reduce the range of querying*. Bloom filters can effectively reduce the number of PMTables scanned during a query. We exploit *OR* operations to implement a mergeable bloom filter. Except the data repository, we assign a fix-sized bloom filter to each PMTable to facilitate the merging of bloom filters. These bloom filters are created when PMTables are initialized, and are merged accompanying with PMTable compactions. When the size of PMTable becomes large, the performance of bloom filters becomes poor due to false positives. Thus, we carefully set the number of levels in the elastic buffer according to the false positive rate of bloom filters.

### 4.7 Crash Consistency and Recovery

Similar to LSM-tree based KV stores such as LevelDB and NoveLSM, MioDB exploits *write-ahead logging* (WAL) to guarantee failure recovery for data written to the DRAM buffer. KV pairs are first appended to a persist log in the NVM sequentially, and then are inserted into the DRAM-based MemTable. In this way, the insertion of KV pairs that often incurs random memory accesses can be performed in the fast DRAM. When the data in the DRAM buffer is flushed to the NVM, MioDB can directly copy data to the PM without requiring a separate log because the log for the DRAM buffer can be still used to guarantee crash consistency. During the zero-copy compaction, because we use atomic writes to update pointers in PMTables and do not move data nodes, MioDB can still guarantee data consistency upon system crashes.

To guarantee failure recovery during the zero-copy compaction, MioDB maintains an insertion mark and some state variables in the NVM to indicate whether a PMTable table is still being merged. The insertion mark records the current data node that is being merged. When we restart the system from a crash, MioDB can continue the uncompleted compaction if the two PMTables are not marked as reclaimable. If the insertion mark is not NULL, MioDB first inserts the marked node into the newtable and then continues the compaction. For read operations, we query KV pairs according to the insertion mark and the state of PMTables. Three corner cases should be carefully considered to guarantee the correctness of querying. 1) For the newtable, when a data node is only removed from the high levels of the skip list, we can still query this node in the low levels of the skip list. 2) When a data node is completely removed from the newtable and not yet added in the oldtable, we can query this node in the insertion mark. 3) When a data node is already added to some low levels of the oldtable, we can query this node in the insertion mark or in the low levels of the oldtable.

## 5 EVALUATION

In this section, we conduct extensive experiments to evaluate the design insight of MioDB. Our experiments use a server equipped with two 20-core 2.10GHz Intel Xeon Gold 6230 CPUs, 28 MB LLC, 64 GB 2666 MHz DDR4 DRAM, and eight 128 GB Intel Optane DC PMMs. The operating system is CentOS 7 with Linux kernel 5.1.1.

At first, we compare MioDB with the state-of-the-art open-sourced NoveLSM [31] and MatrixKV [27] when they are deployed in an in-memory mode. All KV stores use 64 MB DRAM-based MemTables, and NoveLSM and MatrixKV use 64 MB SSTables with an amplification factor of 10. The default key-value sizes in our dataset are 16 bytes and 4 KB, respectively. The bloom filters in MioDB are configured as 16 bits per key. To limit the tail latency of writes, both NoveLSM and MatrixKV use a moderate-sized NVM buffer (8 GB) to store the MemTable and $L_0$, respectively. We follow their configurations in our experiments. However, all SSTables in NoveLSM and MatrixKV are stored in NVM without using SSD. Moreover, we also do not limit the capacity of DRAM tablecache used for SSTables in NoveLSM and MatrixKV. At last, we also compare the SSD-supported MioDB with NoveLSM and MatrixKV when they all store SSTables in SSDs, and other settings are the same as the in-memory mode. We deploy all three KV stores in

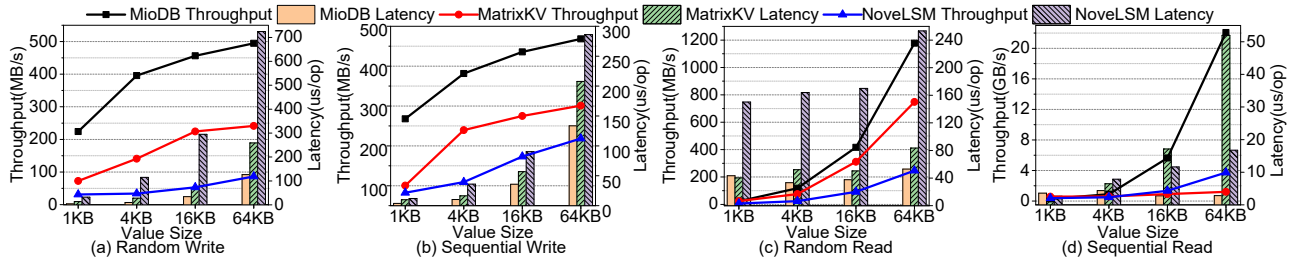


**Figure 6: The performance of in-memory MioDB, MatrixKV, and NoveLSM using micro-benchmarks**

the same server and use the same hardware configuration for each experiment.

## 5.1 Microbenchmarks

At first, we evaluate MioDB using a micro-benchmark–db_bench [7] released by LevelDB. We measure the random/sequential write throughput by writing 80 GB KV pairs in the database. We also measure the random/sequential read performance by querying one million KV pairs from the database.

**Write Performance.** Figure 6(a) shows the random write throughput and latency of MioDB, MatrixKV, and NoveLSM, all deployed in an in-memory mode. For these KV stores, the write throughput are roughly linear to the value size increasing from 1 KB to 64 KB. Compared to MatrixKV and NoveLSM, MioDB can improve the random write throughput by up to 3.1× and 11.6× (2.5× and 8.3× on average), respectively. MioDB can also improve the sequential write throughput by 1.5× and 2.8× on average, respectively, as shown in Figure 6(b). The reason is that both MatrixKV and NoveLSM suffer from costly data serialization and long write stalls during data flushing or $L_0$-to-$L_1$ compaction, while MioDB can eliminate these costs by redesigning the structure of LSM-trees and enabling many optimizations. In MioDB, the random write latency approximates to the sequential write latency because all write operations are first cached in the DRAM-based MemTable, and there is not any write stall during data flushing and compaction. As a result, the random write throughput in MioDB is almost equivalent to the sequential write throughput.

**Read Performance.** Figure 6(c) shows that MioDB can improve the random read throughput by 1.3× and 4.4× on average compared with MatrixKV and NoveLSM, respectively. Figure 6(d) shows MioDB also improves the sequential read throughput by 6.7× and 3.3× on average, respectively. Both random and sequential read latencies in MioDB increase slightly when the value size increases exponentially. The reason is that MioDB avoids costly data deserialization and most queries are responded by the huge PMTable in the $L_n$. Thus, the read throughput in MioDB is roughly improved by 4× with the increase of value size. In contrast, MatrixKV and NoveLSM all suffer from higher cost of data deserialization when the value size increases, resulting in higher read latency and lower read throughput than MioDB.

**Cost Analysis.** We further analyze the costs of three KV stores, as shown in Table 1. MioDB and MatrixKV completely eliminate interval stalls that are perceived by client users. Moreover, MioDB significantly reduces cumulative stalls by 96.2% and 97.4% compared with MatrixKV and NoveLSM, respectively. This is the reason why

**Table 1: Different costs in MioDB, MatrixKV, and NoveLSM**

| Cost | MioDB | MatrixKV | NoveLSM |
|---|---|---|---|
| Interval Stalls (s) | 0 | 0 | 496.9 |
| Cumulative Stalls (s) | 28.1 | 731.3 | 1071.3 |
| Deserialization (s) | 0 | 74.3 | 82.3 |
| Flushing (s) | 13.6 | 191.0 | 511.8 |
| Write Amplification | 2.9× | 5.6× | 6.6× |

MioDB significantly improves the write performance. Since MatrixKV and NoveLSM still use SSTables in NVMs, data queries suffer from costly data deserialization which accounts for 58.5% and 50.7% of total read time. In contrast, MioDB eliminates the deserialization cost using DRAM-NVM compatible data structures. As MioDB performs data flushing 13× and 36.6× faster than MatrixKV and NovelSM, respectively, the write throughput of MioDB is significantly improved. Moreover, MioDB significantly reduces WA by 2.7× and 3.7× compared with MatrixKV and NoveLSM, respectively.

## 5.2 Real-World YCSB Workloads

We also evaluate MioDB using a real-world benchmark–YCSB [7]. We first load 80 GB data into the KV store, and then evaluate workloads A-F with one million read/write operations. The data access patterns all follow a zipfian distribution (99% skewness) except D. Workload A has 50%-50% read-update ratio. Workload B performs 95% reads and 5% updates. Workload C is read-only. Workload D performs 5% inserts and 95% reads for the most recently inserted records. Workload E performs 95% scans (range queries) and 5% inserts. Workload F performs 50% reads and 50% read-modify-writes.

The default value size in YCSB is 1 KB, while NoveLSM and MatrixKV both use 4 KB values for the best performance. Here, we measure both 1 KB and 4 KB value sizes. We also compare MioDB with a special configuration of NoveLSM, i.e., NoveLSM-NoSST. It uses a single and big skip list to store all data in NVM, without using on-disk SSTables. Figure 7 shows the performance of NoveLSM, MatrixKV, NoveLSM-NoSST, and MioDB, all deployed in an in-memory mode. In the following, we first analyze experimental results for 4 KB value size.

For *load* (insert-only) operations, the throughput of MioDB (about 80 KIOPS) is 12.1× and 2.8× higher than that of NoveLSM and MatrixKV, respectively. The throughput of MioDB is also 2.2× higher than NoveLSM-NoSST for *load* operations because NoveLSM-NoSST has to insert KV pairs into the big skip list one by one (costly for sorting). For write-dominant workloads (i.e., A and F), MioDB significantly improves their throughput by up to 2.3× and 5.2×

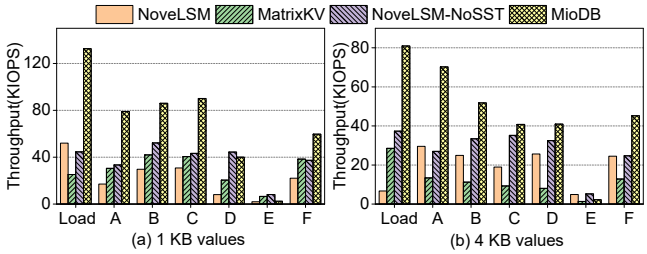


**Figure 7: The throughput (KIOPS) of YCSB workloads running on KV stores deployed in an in-memory mode**

**Table 2: Tail latencies of workload A in YCSB**

| KV Size | KV stores | Latency ($\mu s$) | | | |
|---|---|---|---|---|---|
| | | avg. | 90% | 99% | 99.9% |
| 4 KB | NoveLSM | 223.7 | 617.2 | 698.2 | 764.3 |
| | MatrixKV | 38.8 | 51.9 | 73.7 | 973.6 |
| | **MioDB** | **15.7** | **19.2** | **28.4** | **44.7** |
| 1 KB | NoveLSM | 23.5 | 30.0 | 53.1 | 119.6 |
| | MatrixKV | 41.3 | 51.5 | 76.4 | 484.0 |
| | **MioDB** | **14.0** | **17.1** | **26.6** | **44.5** |

compared with NoveLSM and MatrixKV, respectively. Although NoveLSM can perform in-place updates to the large MemTable (4 GB) in NVM, the cost of inserting/updating a 4KB KV is rather high because it requires at most *log*(4GB/4KB)=20 lookups to locate the insert position. In contrast, MioDB stages all writes in a small MemTable (64 MB) in DRAM. Since the random write throughput of DRAM is about 7 times higher than that of NVM, MioDB can achieve much higher write throughput against NoveLSM. On the other hand, due to the zipfian distribution, most reads are hit in the large MemTable of NoveLSM and the top levels of MioDB, respectively. Thus, the read throughput of NoveLSM and MioDB are similar. Taking both reads and writes into account, MioDB achieves higher total throughput than NoveLSM. These results demonstrate the significant advantage of MioDB for write-dominant workloads.

For read-dominant workloads (i.e., B, C, and D), MioDB also achieves considerable performance improvement compared with other KV stores. Because there are very few PMTables in each level of MioDB and most queries are actually performed on the huge PMTable (a large skip list), the read performance of MioDB is significantly improved by up to 5.1× compared with NoveLSM and MatrixKV. When the write traffic is high, most queries hit in $L_{n-1}$ because most PMTables are accumulated there. MioDB can quickly locate the PMTable which contains the requested key through bloom filters, and searches the key in a smaller skip list rather than a big skip list. Thus, MioDB even achieves 1.2× higher throughput than NoveLSM-NoSST for the read-only workload C.

For scan-dominant workload E, NoveLSM-NoSST (a single large skip list) shows the best performance among all KV stores. This implies that the persistent skip list is also efficient for scans in KV stores. In fact, the workload E is tested immediately after the dataset is loaded into KV stores. During this period, MioDB is still busy with compacting small PMTables in the elastic buffer. Since the benefit of bloom filters for multiple small PMTables declines for

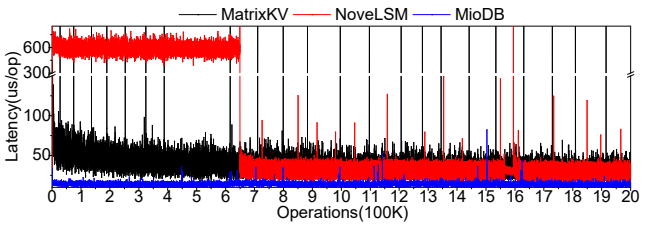


**Figure 8: Latency of workload A in YCSB (4 KB values)**

range queries, the scan performance of MioDB is lower than that of NoveLSM-NoSST. However, when intensive PMTable compactions finish, MioDB also maintains a large sorted skip list in the data repository. The performance of MioDB would approach that of NoveLSM-NoSST for scan operations.

When the value size decreases from 4 KB to 1 KB, MioDB shows even higher performance improvement than NoveLSM and MatrixKV. Since there are more KV pairs of 1 KB values in MioDB than the case of 4 KB values, more frequent data compactions are conducted for smaller KV pairs. Because the compaction in MioDB is much faster than other KV stores, MioDB achieves higher performance improvement for smaller KV pairs. These results demonstrate that MioDB is even more efficient for small KV pairs.

**Tail Latency** is critical for latency-sensitive applications on LSM-tree based KV stores. We use YCSB workload A to evaluate the tail latencies of requesting 1 KB and 4 KB KV pairs. Table 2 shows the average, $90^{th}$, $99^{th}$, and $99.9^{th}$ percentile latencies in in MioDB, MatrixKV, and NoveLSM. When the value size is 4 KB, the $99.9^{th}$ percentile latency of MioDB is 21.7× and 17.1× lower than MatrixKV and NoveLSM, respectively. When the value size is smaller, MioDB still shows much lower latency than NoveLSM and MatrixKV. These results demonstrate that MatrixKV can significantly improve the quality of service by reducing write stalls.

Figure 8 shows that MioDB significantly reduces latency spikes compared with MatrixKV and NoveLSM for workload A. During the load phase of YCSB, NoveLSM uses up all MemTables in both DRAM and NVM buffers. When workload A starts, NoveLSM has to flush the MemTables to SSTables, and thus the read/write latency is extremely high due to periodical write stalls. When MemTables have been flushed from the DRAM buffer completely, NoveLSM can use the DRAM MemTables to stage write requests, and thus the average latency declines significantly. MatrixKV also shows higher latency at the beginning because $L_0$-to-$L_1$ compactions result in massive cumulative stalls. Both NoveLSM and MatrixKV show many latency spikes during periodical write stalls. In contrast, as MioDB eliminates almost all write stalls, both average and tail latencies are very low.

## 5.3 Sensitivity Studies

The number of levels in the elastic buffer, the size of dataset, and the MemTable size have a substantial impact on the performance of MioDB. In the following, we use db_bench to evaluate how different parameters affect the performance of KV stores.

**The Number of Levels.** Figure 9 shows how the random write latency and throughput of MioDB vary with the number of levels in the elastic buffer. We note that NoveLSM does not support parallel

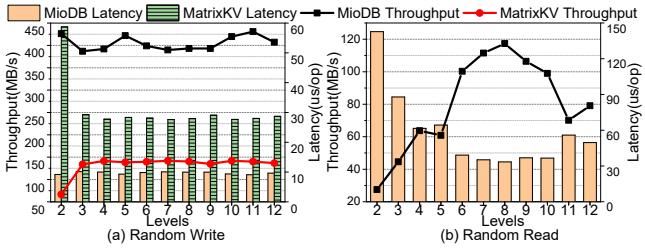


**Figure 9: The performance varies with the number of levels (i.e., compaction threads) in the elastic buffer of MioDB**

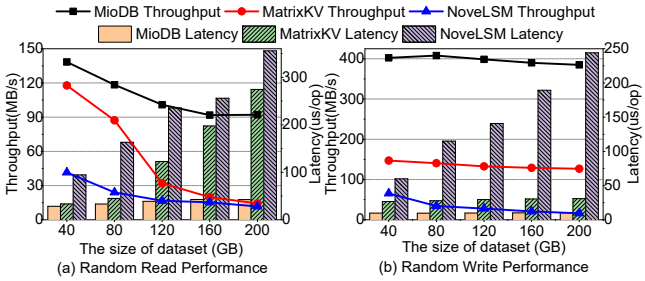


**Figure 10: The random read/write performance varies with the size of dataset in MioDB, MatrixKV, and NoveLSM**

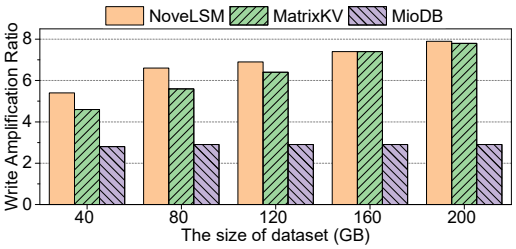


**Figure 11: The WA ratio varies with the size of dataset**

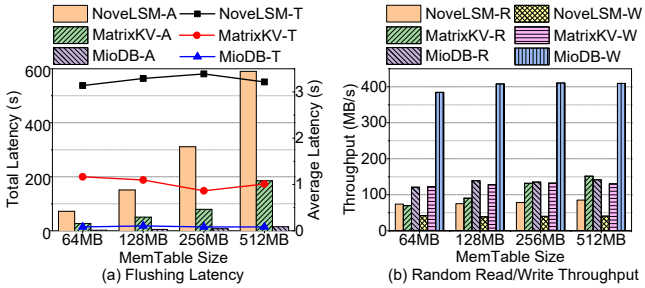


**Figure 12: The latency and throughput of MemTable flushing vary with the size of MemTable in the DRAM. A/T denotes average/total latency, and R/W denotes read/write bandwidth.**

compaction, and thus the number of levels in the LSM-tree does not affect the write performance of NoveLSM. In contrast, MioDB uses one thread to compact PMTables in each level. Thus, the number of levels is equal to the number of compaction threads. MatrixKV also supports parallel compaction for all levels. However, the parallelism of compaction is often lower due to the data dependence between two adjacent levels. The random write latency and throughput of MioDB remain stable when the number of compaction threads increase. The reason is that the elastic buffer in MioDB can receive burst writes in $L_0$, and PMTable compactions do not cause any write stalls during the data flushing. As a result, the write performance is mainly determined by the speed of data flushing in MioDB. MatrixKV needs 4 compaction threads to achieve its best write performance, which is much lower than MioDB no matter how many compaction threads are used. This implies MioDB spends less CPU time in data compaction than MatrixKV because MioDB does not move data during the zero-copy compaction.

The random read performance is also significantly improved by using more levels in MioDB, as shown in Figure 9(b). This verifies our designs that deeper levels and parallel compaction can flush data to the data repository quickly, and eventually improve the read performance of MioDB. We do not show the results of MatrixKV in Figure 9(b) because the number of levels is determined by the size of dataset (not configurable). MioDB achieves the best performance when the number of levels increases to 8. Because the size of a PMTable in the 8th level becomes large enough to exploit the fast intra-PMTable querying. When the number of levels continues to increase, the size of a single PMTable in the 9th level grows to 16 GB. As the false positive rate of bloom filters becomes high, the read performance of MioDB declines. Based on this observation, we set the number of levels in MioDB to 8.

**The Size of Datasets.** Figure 10 shows how the size of dataset affects the random read/write performance of MioDB, MatrixKV and NoveLSM. When the size of dataset increases from 40 GB to 200 GB, the random read/write performance in NoveLSM and MatrixKV declines significantly due to increased write stalls and write amplification. In contrast, MioDB only shows a slight slowdown of write performance because of the elastic buffer and zero-copy compaction. Moreover, since there is not data deserialization cost in MioDB, and queries are performed fast within large skip lists, its read throughput only drops by 33.5% when the capacity of KV store increases by 5 times.

**Write Amplification (WA)**. Figure 11 shows the WA ratios of three KV stores when we write five datasets with different sizes into the KV stores. Because MioDB exploits zero-copy compactions for $L_0$ to $L_{n-1}$, and only causes WA during the lazy-copy compaction, its WA ratio is only 2.9× (the theoretical upper bound is 3). When the dataset increases to 200 GB, MioDB significantly reduces write amplification by up to 5× and 4.9× compared with NoveLSM and MatrixKV, respectively.

**The Size of MemTables**. Figure 12 shows how the size of the MemTable in the DRAM affects the latency and throughput of MemTable flushing when we use db_bench to write 80 GB datasets into three KV stores. As shown in Figure 12(a), the average flushing latency of MioDB for each MemTable is 37.6× and 11.9× shorter than that of NoveLSM and MatrixKV, respectively. The reason is that the one-piece flushing in MioDB can completely eliminate the cost of data serialization in NoveLSM and MatrixKV. Moreover, the data copying in MioDB is also much more efficient because MioDB only uses one *memcpy* to copy the MemTable in a batch, while NoveLSM and MatrixKV have to copy all KV pairs to NVM one by one. Specifically, the latency of flushing a 64 MB MemTable in MioDB is only 11 milliseconds on average. This implies that the

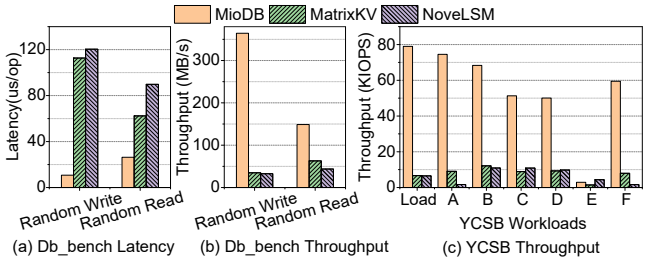


**Figure 13: The performance of MioDB, MatrixKV and NoveLSM when DRAM/NVM/SSD are used**

**Table 3: Tail latencies of of YCSB workload A when DRAM/NVM/SSD are used in NoveLSM, MatrixKV, and MioDB**

| KV Size | KV stores | Latency ($\mu$s) | | | |
|---|---|---|---|---|---|
| | | avg. | 90% | 99% | 99.9% |
| 4 KB | NoveLSM | 291.2 | 626.2 | 713.9 | 971.8 |
| | MatrixKV | 99.5 | 137.7 | 157.1 | 1979.5 |
| | **MioDB** | **14.7** | **16.0** | **20.1** | **39.6** |
| 1 KB | NoveLSM | 189.8 | 376.8 | 4217.3 | 5696.2 |
| | MatrixKV | 32.9 | 41.0 | 61.9 | 389.3 |
| | **MioDB** | **15.7** | **19.1** | **26.7** | **103.5** |

background pointer swizzling has a little impact on the effective capacity of DRAM used for MemTables. Although the latency for flushing a larger MemTable is higher, less MemTables are flushed in total for a given dataset. Thus, all three KV stores show slight fluctuations of the total flushing time regardless of the MemTable size. Figure 12(b) shows that the random write throughput also varies slightly with the MemTable size for each KV store. On the other hand, the random read throughput of all KV stores is improved slightly because more read operations hit in the DRAM buffer when it becomes larger. Overall, the MemTable size only affects the frequency of flushing, but has very limited impact on the random read/write throughput.

## 5.4 SSD-Supported MioDB Extension

We also evaluate the performance of SSD-supported MioDB when the value size is 4 KB. Figure 13(a) and (b) show that MioDB can significantly improve the random read/write performance of db_bench compared with MatrixKV and NoveLSM. Particularly, MioDB improves the random write throughput by 10.5× and 11.2×, respectively. Figure 13(c) also shows significant performance improvement for YCSB workloads. For write-dominant *load* workloads, the throughput of MioDB is even 11.8× and 12.1× higher than that of MatrixKV and NoveLSM. The reason is that the elastic buffer in MioDB can effectively handle I/O bursts to minimize most write stalls, and eliminate data movement via zero-copy compaction before data is written into the SSD. For read-dominant workloads (i.e., B, C, and D), MioDB can also improve the throughput by up to 5.7× and 6.3× because most KVs can be retrieved from the elastic buffer. Table 3 also shows that MioDB can significantly reduce the $99.9^{th}$ percentile latency of YCSB workload A by up to 49.9× and 24.5×, respectively.

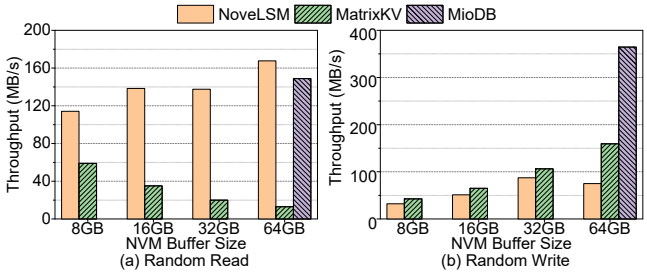


**Figure 14: The throughput of db_bench varies with the size of NVM buffer in MioDB, MatrixKV and NoveLSM**

NoveLSM and MatrixKV use fix-sized 8 GB NVM as an extension of the DRAM buffer, while MioDB uses an elastic NVM buffer to handle write bursts. Thus, MioDB can use more NVMs when it is heavily loaded. However, when the load becomes light, MioDB actually maintains only one PMTable in each level, and thus the total NVM consumption is 16 GB. When YCSB workload A is testing on the 80 GB dataset, the maximum and average NVM usage in MioDB are 39.1 GB and 19.5 GB, respectively. These experimental results demonstrate that our multi-level elastic buffer in MioDB can also achieve high performance in a DRAM-NVM-SSD architecture with reasonable NVM consumption.

To further explore the impact of the NVM buffer size on the performance of KV stores deployed in a DRAM-NVM-SSD storage hierarchy, we increase the size of NVM MemTables in NoveLSM and the matrix container in MatrixKV exponentially. Since the size of the elastic buffer in MioDB can change elastically, we configure the maximum capacity that the NVM buffer can use as 64 GB. Figure 14 shows the random read/write throughput varies with the size of the NVM buffer in MioDB, MatrixKV and NoveLSM. For NoveLSM, using a large skip list in NVM can reduce the frequency of MemTable flushing and improves the read throughput because more KV pairs can be directly read from a larger buffer. The random write throughput of NoveLSM also increases moderately with a larger NVM buffer. However, because more lookups are required to locate the position of a given KV pair for each read/write when the buffer size becomes larger, the cost of synchronous reads/writes would offset the benefit of a larger NVM buffer. As shown in the Figure 14(b), when the NVM buffer size increases from 32 GB to 64 GB, the random write throughput of NoveLSM even declines. For MatrixKV, although a large matrix container in NVM can improve the throughput of random writes, it increases the cost of reads because the single column compaction lowers the performance of indexing in the matrix container. As a result, the random read throughput of MatrixKV even decreases with the growth of the buffer size.

When all three KV stores can use 64 GB NVM buffers, the random write and read throughput of MioDB is 2.3× and 11.4× higher than that of MatrixKV, respectively. MioDB also improves the random write throughput by 4.9× compared with NoveLSM. Because MioDB can also exploit the good feature of large skip lists for read operations, the random read throughput of MioDB approximate to that of NoveLSM. These results demonstrate that the significant performance improvement of MioDB is not attributed to simply using a large buffer, but stems from our graceful design of the multi-level PMTable compaction.

## 6 RELATED WORK

We present the most related work in the following categories.

**KV store indexing for hybrid memories.** A few earlier studies have exploited byte-addressability and persistence features of NVMs to design KV store systems. Write atomic B$^+$-tree [5], NV-Tree [42], FPTree [32], and FAST&FAIR [17] are proposed to optimize B$^+$-tree indexing for NVMs. Level hashing [48], CCEH [30], HMCached [19], and STLT [44] all focus on hash indexing optimizations for NVMs. HiKV [40] is a hybrid-indexed KV store that exploits in-NVM hash tables and an in-DRAM B$^+$-tree to accelerate single-key and range queries, respectively. ChameleonDB [47] uses LSM-trees to admit writes, and an in-DRAM hash table to accelerate read operations. FlatStore [6] also proposes a persistent log structure in NVM for write operations, and exploits hash and B$^+$-tree indexes in DRAM for fast indexing. PACTree [22] is a persistent hybrid index structure comprising a trie index for internal nodes and B$^+$-tree-like leaf nodes. These proposals all store KV pairs in NVMs directly, and maintain hybrid indexes in DRAM to improve the read performance. However, since NVMs have much higher write latency and lower bandwidth than DRAM, it is meaningful to rethink the principle of log-structured merging for improving write throughput of NVM-based KV stores.

**LSM-tree based KV stores designed for fast SSDs.** There have been a number of studies on redesigning LSM-trees for SSD devices. PebblesDB [36] presents fragmented LSM-trees to reduce data involved in each compaction, and thus mitigates write amplification. WiscKey [26] splits keys and values, and only uses keys for compaction to avoid rewriting values. However, it complexes garbage collection and range querying. SILK [3] proposes an I/O scheduler to reduce the tail latency of LSM-based KV stores. Dostoevsky [8] makes a trade-off between storage space and performance of LSM-tree based KV stores by adaptive removing superfluous merging. KVell [23] aims to fully utilize SSD bandwidth to maximize read/write performance of KV stores. RLSM [37] simplifies the logical layout of storage and keeps data as unsorted to improve the write performance. Jungle [1] transplants copy-on-write B+-tree into LSM-tree to reduce update cost without sacrificing lookup performance. UniKV [46] unifies hash indexing and the LSM-tree in a single system to simultaneously improve the performance of reads/writes/scans. Although some of these schemes designed for SSDs are also meaningful for NVM devices, they may not be applicable to hybrid memory systems directly.

**LSM-tree based KV stores designed for NVMs.** SLM-DB [20] designs a single-level LSM-tree in an NVM-SSD architecture, and uses a B+-tree to accelerate read operations. However, data compactions in SLM-DB are very costly because SLM-DB must guarantee the order of B+-tree indexes. SLM-DB always compacts SSTables in a candidate list by choosing SSTables with a large overlapping range of keys. When the candidate list becomes large, the selection can be very costly. Meanwhile, since the operations on B+-tree must be serialized, it is impossible for SLM-DB to perform data flushing and compaction in parallel, and thus often suffers non-trivial write stalls. NoveLSM [21] exploits a mutable and persistent large MemTable in NVMs to extend the capacity of DRAM buffer. The write stalls still exist when MemTables are flushed to block-based SSTables. NVMRocks [24] aims at the awareness of NVM in RocksDB. It also uses persistent MemTable in NVMs, and thus faces write stall problems like NoveLSM. To solve the problem of write stalls of LSM-trees, MatrixKV [43] designs a matrix container to receive MemTables in NVMs, and performs fine-grained data compaction to reduce the cost of NVM-to-SSD data compaction. These studies all rely on block-based SSTables and traditional LSM-trees to store and manage KV pairs, and still suffer from costly data serialization/deserialization and periodical write stalls. In contrast, MioDB fully utilizes the byte-addressability feature of NVMs to manage data in skip lists with multiple levels. Our multi-level buffer architecture enables many optimizations to mitigate the cost of data compaction, such as one-piece flushing, zero-copy compaction, and parallel compaction. It allows costly skip-list merging operations to be performed asynchronously and efficiently without causing write amplification. Thus, our multi-level elastic buffer in MioDB is fundamentally different from NoveLSM, and significantly reduces write stalls and write amplification for LSM-tree based KV stores.

## 7 CONCLUSION

We present MioDB, a novel LSM-tree based *key-value* (KV) store system designed to fully exploit the advantages of byte-addressable NVMs. We advocate byte-addressable and persistent skip-lists to replace the on-disk data structure of LSM-tree. We propose one-piece flushing to minimize the cost of data serialization from DRAM to NVM. We exploit an elastic NVM buffer with multiple levels and zero-copy compaction to eliminate write stalls and reduce write amplification. Our experimental results demonstrate that MioDB achieves 17.1× and 21.7× lower $99.9^{th}$ percentile latencies, 8.3× and 2.5× higher random write throughput, and up to 5× and 4.9× lower write amplification compared with the state-of-the-art NoveLSM and MatrixKV, respectively.

## DATA-AVAILABILITY STATEMENT

The source code, benchmarks, datasets, and experimental data related to this publication are available in the repository at Github [28] and Zenodo [29].

## ACKNOWLEDGMENTS

We appreciate anonymous reviewers and our shepherd for their constructive comments. This work is supported jointly by National Key Research and Development Program of China under grant No.2022YFB4500303, and National Natural Science Foundation of China (NSFC) under grants No.62072198, 61732010, 61825202, 61929103. This work is also sponsored by Huawei Technologies Co., Ltd (No. YBN2021035018A7).

## REFERENCES

[1] Jung-Sang Ahn, Mohiuddin Abdul Qader, Woon-Hak Kang, Hieu Nguyen, Guogen Zhang, and Sami Ben-Romdhane. 2019. Jungle: Towards Dynamically Adjustable Key-value Store by Combining LSM-tree and Copy-on-write B+-tree. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*.

[2] Apache HBase. 2022. http://hbase.apache.org/.

[3] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. 2019. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*. 753–766.

[4] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008.

Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 1–26.
[5] Shimin Chen and Qin Jin. 2015. Persistent B+-trees in Non-volatile Main Memory. *Proceedings of the VLDB Endowment* 8, 7 (2015), 786–797.
[6] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. 2020. FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory. In *Proceedings of the 2020 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 1077–1091.
[7] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*. 143–154.
[8] Niv Dayan and Stratos Idreos. 2018. Dostoevsky: Better Space-time Trade-offs for LSM-tree based Key-value Stores via Adaptive Removal of Superfluous Merging. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD)*. 505–520.
[9] Zhuohui Duan, Haikun Liu, Xiaofei Liao, Hai Jin, Wenbin Jiang, and Yu Zhang. 2019. HiNUMA: NUMA-aware Data Placement and Migration in Hybrid Memory Systems. In *Proceedings of the 2019 IEEE 37th International Conference on Computer Design (ICCD)*. IEEE, 367–375.
[10] Zhuohui Duan, Haikun Liu, Haodi Lu, Xiaofei Liao, Hai Jin, Yu Zhang, and Bingsheng He. 2021. Gengar: An RDMA-based Distributed Hybrid Memory Pool. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*. 92–103.
[11] Zhuohui Duan, Haodi Lu, Haikun Liu, Xiaofei Liao, Hai Jin, Yu Zhang, and Song Wu. 2021. Hardware-supported Remote Persistence for Distributed Persistent Memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 1–14.
[12] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. 2016. Data Tiering in Heterogeneous Memory Systems. In *Proceedings of the 9th ACM European Conference on Computer Systems (EuroSys)*. 15:1–15:16.
[13] Facebook RocksDB. 2022. http://rocksdb.org/.
[14] FIO. 2022. https://github.com/axboe/fio.
[15] Guy Golan-Gueta, Edward Bortnikov, Eshcar Hillel, and Idit Keidar. 2015. Scaling Concurrent Log-structured Data Stores. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys)*. 1–14.
[16] Google LevelDB. 2022. https://github.com/google/leveldb.
[17] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)*. 187–200.
[18] Intel Optane DIMMs. 2022. https://www.tomshardware.com/news/intel-optane-dimm-pricing-performance,39007.html.
[19] Hai Jin, Zhiwei Li, Haikun Liu, Xiaofei Liao, and Yu Zhang. 2020. Hotspot-Aware Hybrid Memory Management for In-Memory Key-Value Stores. *IEEE Trans. Parallel Distributed Syst.* 31, 4 (2020), 779–792.
[20] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H Noh, and Young-ri Choi. 2019. SLM-DB: Single-level Key-value Store with Persistent Memory. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST)*. 191–205.
[21] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*. 993–1005.
[22] Wook-Hee Kim, R Madhava Krishnan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. 2021. PACTree: A High Performance Persistent Range Index Using PAC Guidelines. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*. 424–439.
[23] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2019. KVell: the Design and Implementation of a Fast Persistent Key-value Store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 447–461.
[24] Li J, Pavlo A, Dong S. 2022. Nvmrocks: Rocksdb on Non-volatile Memory Systems. http://istcbigdata.org/index.php/nvmrocks-rocksdb-on-non-volatilememory-systems/..
[25] Ruicheng Liu, Peiquan Jin, Xiaoliang Wang, Zhou Zhang, Shouhong Wan, and Bei Hua. 2019. NVLevel: A High Performance Key-Value Store for Non-Volatile Memory. In *Proceedings of the 2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. 1020–1027.
[26] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2017. Wiskey: Separating Keys from Values in SSD-conscious Storage. *ACM Transactions on Storage (TOS)* 13, 1 (2017), 1–28.
[27] MatrixKV. 2020. https://github.com/PDS-Lab/MatrixKV.
[28] mioDB. 2022. https://github.com/CGCL-codes/mioDB.
[29] mioDB artifact. 2022. https://doi.org/10.5281/zenodo.7423535.
[30] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H Noh, and Beomseok Nam. 2019. Write-Optimized Dynamic Hashing for Persistent Memory. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST)*. 31–44.
[31] NoveLSM. 2019. https://github.com/SudarsunKannan/lsm_nvm.
[32] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-tree for Storage Class Memory. In *Proceedings of the 2016 International Conference on Management of Data*. 371–386.
[33] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-structured Merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385.
[34] William Pugh. 1990. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Commun. ACM* 33, 6 (1990), 668–676.
[35] Moinuddin K Qureshi, Vijayalakshmi Srinivasan, and Jude A Rivers. 2009. Scalable High Performance Main Memory System Using Phase-change Memory Technology. *ACM SIGARCH Computer Architecture News* 37, 3 (2009), 24–33.
[36] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. Pebblesdb: Building Key-value Stores Using Fragmented Log-structured Merge Trees. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*. 497–514.
[37] Nae Young Song, Heon Young Yeom, and Hyuck Han. 2018. Efficient Key-value Stores with Ranged Log-structured Merge Trees. In *Proceedings of the 2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. 652–659.
[38] Xiaoyuan Wang, Haikun Liu, Xiaofei Liao, Ji Chen, Hai Jin, Yu Zhang, Long Zheng, Bingsheng He, and Song Jiang. 2019. Supporting Superpages and Lightweight Page Migration in Hybrid Memory Systems. *ACM Trans. Archit. Code Optim.* 16, 2 (2019), 11:1–11:26.
[39] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. 2015. LSM-trie: An LSM-tree-based Ultra-large Key-value Store for Small Data Items. In *Proceedings of the 2015 USENIX Annual Technical Conference (ATC)*. 71–82.
[40] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. 2017. HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 349–362.
[41] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST)*. 169–182.
[42] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*. 167–181.
[43] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. 2020. MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*. 17–31.
[44] Chencheng Ye, Yuanchao Xu, Xipeng Shen, Xiaofei Liao, Hai Jin, and Yan Solihin. 2021. Hardware-based Address-Centric Acceleration of Key-Value Store. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 736–748.
[45] Chencheng Ye, Yuanchao Xu, Xipeng Shen, Xiaofei Liao, Hai Jin, and Yan Solihin. 2021. Supporting Legacy Libraries on Non-volatile Memory: a User-Transparent Approach. In *Proceedings of the ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 443–455.
[46] Qiang Zhang, Yongkun Li, Patrick PC Lee, Yinlong Xu, Qiu Cui, and Liu Tang. 2020. UniKV: Toward High-Performance and Scalable KV Storage in Mixed Workloads via Unified Indexing. In *Proceedings of the 2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 313–324.
[47] Wenhui Zhang, Xingsheng Zhao, Song Jiang, and Hong Jiang. 2021. ChameleonDB: A Key-value Store for Optane Persistent Memory. In *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys)*. 194–209.
[48] Pengfei Zuo, Yu Hua, and Jie Wu. 2018. Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 461–476.

Received 2022-07-07; accepted 2022-09-22