2022 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)



# Improving Write Performance for LSM-tree-based Key-Value Stores with NV-Cache

Xuzhen Jiang[†], Miao Cai[*†‡], Baoliu Ye[‡*†]
[*]Key Laboratory of Water Big Data Technology of Ministry of Water Resources, Hohai University
[†]School of Computer and Information, Hohai University
[‡]State Key Laboratory for Novel Software Technology, Nanjing University

***Abstract*—Log Structured Merge tree (LSM-tree) is widely used in modern key-value stores due to its superior write performance. However, current LSM-tree-based key-value stores suffer from two performance issues: (1) write stall caused by speed gap between fast DRAM and slow storage devices; (2) write amplification incurred by existing LSM-tree compaction.**

**This paper proposes NV-Cache, a write-optimized non-volatile cache for LSM-tree-based key-value stores. We design a split LSM-tree over a hybrid storage hierarchy consisting of fast Non-Volatile Memory (NVM) and slow solid state drive (SSD). It promotes frequently-accessed, high LSM-tree levels onto fast NVM and offloads cold, large LSM-tree levels into the slow storage device. Further, we propose a range of techniques, including unsorted LSM-tree node layout, lightweight MemTable flush, write-optimal list compaction, and LRU-based NV-Cache eviction, to extensively optimize the write path. NV-Cache closes the performance gap between fast DRAM and slow storage devices to prevent the write stall and reduce write amplification. We implement NV-Cache based on Google LevelDB and conduct a series of experiments using db_bench and YCSB workloads. Experiment results suggest that NV-Cache effectively prevents write stall, achieves 2.3× write reduction, and performs 54% better than LevelDB, NoveLSM, and MatrixKV.**

***Index Terms*—Key-Value Store, LSM-Tree, Non-Volatile Memory**

## I. Introduction

LSM-tree-based key-value stores (KVS), such as LevelDB [1], RocksDB [2], and Cassandra [3], are popularly used in today's data center and big data applications like social networks [4], e-commerce platforms [5], and search engines [6], etc. We use popular Google LevelDB [1] to demonstrate LSM-tree-based key-value store. LSM-tree exhibits a multi-layer structure that provides attractive write performance. It adopts an out-of-place data write strategy. In LevelDB, a data write is first issued to a volatile in-DRAM buffer called MemTable. Lately, data are flushed to the persistent SSTables. Furthermore, newer data stored in the top LSM-tree level are moved to the lower levels and merged with older data versions. This procedure is called LSM-tree data compaction. Ideally, a LSM-tree achieves much better write performance than other indexes like $B^+$ tree. Unfortunately, two performance problems degrade its write performance.

First, current LSM-tree-based key-value stores are deployed in slow storage devices like HDDs or SSDs [7]. They keep MemTables in fast DRAM to buffer new data writes. As LSM-tree $L_0$ level has a size restriction (e.g., 40MB in LevelDB), compaction moves data down to the next layer to accommodate new data flushed from the MemTables. However, the performance gap between DRAM and SSDs or HDDs leads to unbalanced data write speed in DRAM MemTable writes, DRAM MemTable flush to SSD, and LSM-tree $L_0$-$L_1$ compaction in SSD, causing pending requests to be easily stuck at MemTable write.

Second, LSM-tree compaction loads multiple SSTables into the DRAM, performs data sorting, merges key-value data stored in these SSTable files, and finally flushes data to new SSTables in the lower tree level. Independent LSM-tree levels have redundant data storage for a key-value pair. Repeated data writes in conventional block-based devices across different tree levels cause significantly high data write and I/O amplification.

Recent non-volatile memories (NVMs) like Intel Optane persistent memory promises byte-addressability, data durability, ultra-low latency, and large memory bandwidth [8]. These attractive device characteristics provide great opportunities to address the aforementioned challenges. Towards this end, this paper proposes NV-Cache, a non-volatile cache to improve write performance for LSM-tree-based key-value stores. The core of NV-Cache is a novel split log-structured merged tree design, which promotes frequently-accessed, high LSM-tree levels onto high-speed, byte-addressable NVM and offloads large, lower levels into relatively slow, block-based storage devices (SSD in our paper).

Furthermore, we propose a range of techniques to extensively optimize the write path in NV-Cache. In particular, we design a high-performance cached LSM-tree for high-speed NVM. This cached LSM-tree has a persistent skiplist based data organization with a novel storage layout called Non-Volatile Table (NVTable). NVTable features an unsorted, linked-list-based data layout that supports write-ahead-log-free, lightweight MemTable flush. This MemTable flush design unifies the write-ahead log and persistent data writes in an NVM linked list, which removes the data persist off the MemTable flush. To reduce the write amplification in data compaction, we propose a list compaction algorithm to avoid data movement between layers. List compaction leverages NVM byte-addressability to organize KV pairs in an NVTable as a linked list. Data merging between high-level and lower-level NVTables only requires updating eight-byte list pointers without copying large, actual data. Besides, hot data separation in NV-Cache guarantees most data compaction happens in fast NVM. Finally, we integrate an LRU mechanism into NV-Cache to minimize data eviction from NVM to SSD, thereby

978-1-6654-6497-0/22/$31.00 ©2022 IEEE
DOI 10.1109/ISPA-BDCloud-SocialCom-SustainCom57177.2022.00057

reducing slow, write-heavy compaction in SSD-based storage.

We implement NV-Cache in Google LevelDB and compare with LevelDB, NVM-optimized NoveLSM [9] and MatrixKV [7] with a range of microbenchmarks and YCSB workloads. Experiment results suggest that NV-Cache improves write throughput by up to 74% and reduces write tail latency by up to 83% compared to state-of-the-art research works.

Overall, this paper makes the following contributions:

- We exploit fast, byte-addressable NVM to design a write-optimized NV-Cache to address both write stall and write amplification in conventional LSM-tree-based key-value stores in slow storage devices.
- We propose a novel split LSM-tree structure over hybrid NVM-SSD storage. It improves LSM-tree-based key-value stores write performance by proposing a range of techniques, including lightweight MemTable flush, NVM-friendly skiplist, list compaction algorithm, and LRU-based data eviction mechanism.
- We conduct extensive experiments to demonstrate NV-Cache performance and effectiveness in write stall prevention, write amplification reduction, and write performance improvement for skewed data flow.

The rest of this paper is organized as follows. Section §II describes the background and motivation. Section §III elaborates on NV-Cache design and implementation. Section §IV shows the experimental results. We discuss related work in Section §V and conclude this paper in Section §VI.

## II. Background And Motivation

### A. LSM-tree-based Key-Value Store

The key-value storage system gains great attention in many real-world applications and becomes an essential building block of them [6], [10]. Key-value stores are useful for handling real-time random data access [11], managing metadata [12], and serving data cache, and their adoption is rapidly accelerated with the increasing amount of data volume in the big data era.

Log Structured Merge Tree is a data structure that performs out-of-place data updates to transform random I/O into sequential I/O. LSM-tree is superior in data write performance and is widely used in modern key-value storage systems [13]. All key-value requests, including insert, update, and remove, are implemented by appending a log entry to the LSM-tree. We use LevelDB as an example to further demonstrate LSM tree design in detail.

In LevelDB, each newly inserted key-value pair is first persisted in the write-ahead log. Then, it inserts KV data into the volatile MemTable, which serves as a temporary buffer in DRAM. When the in-DRAM MemTable size grows beyond the threshold (i.e., 4 MB by default), it is switched into the ImmuTable. A background worker thread flushes the ImmuTable to the disk and generates a new SSTable in the LSM-tree $L_0$ level. LSM-tree manages a large number of SSTable files from $L_0$ to $L_N$ (N is 6 by default). Every LSM-tree level has a size limitation. The size amplification factor between $L_{N-1}$ and $L_N$ is ten in LevelDB. To prevent exceeding the size limitation, LevelDB schedules periodic background compaction. The compaction picks a SSTable from a level whose size exceeds the threshold and merges data with SSTables in the next level.

### B. Write Issues in LevelDB

The current LSM-tree-based key-value store has two common write issues. We use LevelDB as an example to analyze them in detail.

**Write stall.** A new key-value pair is written to the MemTable first. However, when MemTable size reaches the threshold, threads must wait until the background thread finishes flushing data from DRAM to SSD. Commodity NAND flash-based SSD is orders of magnitude slower than DRAM in terms of latency and bandwidth. It almost takes around $7\mu s$ to flush a MemTable to SSD. For write-intensive applications, the data write bursts easily fill the MemTable, leading to no available MemTable space for new data and making subsequent write requests stall. Moreover, current LSM-tree compaction causes heavy write amplification. It further results in extra I/O traffic and exacerbates the write stall problem.

**Write amplification.** LSM-tree compaction leads to severe write amplification. When LevelDB compacts an SSTable $t_1$ in $L_{n-1}$ to another SSTable $t_2$ in the lower layer $L_n$, it loads key-value pairs of $t_1$ and $t_2$ into the DRAM, sorts and deduplicates data between $t_1$ and $t_2$, writes the compaction results into new SSTables. As a result, only a small portion of data is deduplicated during compaction. The remaining data is read and written redundantly from $L_{n-1}$ to $L_n$. This redundant data read and write amplifies level by level. Prior work [14] reports that actual SSD writes are ten times higher than writes issued by the user, mainly due to amplified compaction. More seriously, another work [15] finds that as the size of each LSM-tree level increases, the key range of an SSTable in one level often overlaps more SSTables in lower levels. Consequently, when thread updates a key-value pair in a SSTable in the upper level, this SSTable compaction involves multiple SSTables stored in the lower level, resulting in more severe write amplification.

### C. NVM versus SSD

Non-volatile memory technologies, such as Phase-Change Memory [16], ReRAM [17], and 3D XPoint [18], significantly advance current storage devices as well as offer great opportunities for storage systems. NVMs provide comparable read and write latency as DRAM [19]. Similar to SSDs, NVMs support data persistence. Besides, NVMs have three main differences from flash-based SSDs.

First, NVM exhibits superior performance than SSD, i.e., up to $1000\times$ lower I/O latency, $10\times$ higher R/W bandwidth [8]. Second, NVM has higher write endurance [20]. Flash memory suffers from significant wear issues. Its internal flash page only supports $10^5$ program/erase. Compared to SSD, NVM has nearly $1000\times$ higher endurance. Third, SSD provides a fixed-size page-based read/write interface. On the other side, NVM is byte-addressable via the memory controller. All these

promising hardware features provide opportunities to address the write issues in LSM-tree-based KVS.

## III. Design

### A. NV-Cache Overview

Figure 1 demonstrates the split log-structured merged tree. The split LSM-tree consists of two parts. Upper layers, which are frequently accessed, are uploaded onto the fast NVM, whereas lower levels reside on the slow SSD. Such a split LSM-tree structure guarantees most of the data writes, such as volatile MemTable flush and persistent LSM-tree compaction, are handled in the fast NVM, to improve the write performance for key-value stores.

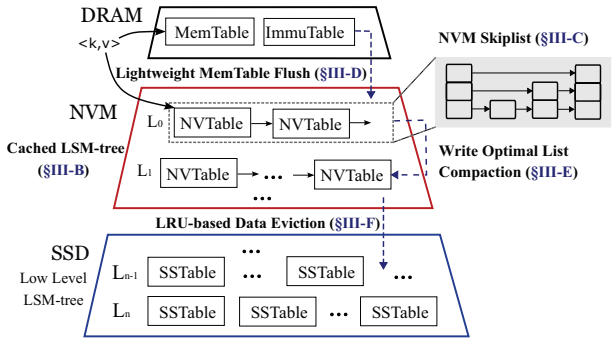


Figure 1: NV-Cache Overview.

Furthermore, we design a range of techniques on ultra-fast, byte-addressable NVM to extensively optimize the write path. Specifically, these techniques include:

- We exploit NVM byte-addressability to design a novel data storage layout called NVTable for cached LSM-tree. Different from the previous array-based ordered data storage format of LevelDB SSTable, NVTable manages key-value pairs in an unsorted, linked list to support fast data writing. It also introduces an offset array in NVTable to improve read performance §III-B.
- We propose a persistent skiplist to manage NVTables which supports efficient lock-free NVTable lookup. We also propose a safe memory reclaim mechanism to ensure safe memory access to NVTable with multi-version skiplist §III-C.
- We propose a write-ahead-log-free, lightweight MemTable flush design. It unifies the write-ahead log and persistent data writes in the NVTable. A synchronous MemTable flush only needs to build the offset array without any data persists §III-D.
- We propose a write-optimal list compaction algorithm. List compaction merges multiple NVTables by simply connecting two linked lists and deduplicating key-value data. It greatly reduces the redundant data writes incurred by the conventional compaction algorithm §III-E.
- We propose a LRU-based NV-Cache data eviction mechanism. We build a LRU-list over NVTables stored in the cached LSM-tree which picks the least frequently updated NVTable to evict. This cache eviction policy minimizes future compaction happened in the slow SSD §III-F.

### B. NV-Cache Data Organization

The cached LSM-tree uses NVTable for key-value data storage and management. SSTable in LevelDB is an ordered linear array of key-value pairs. SSTable adjustments like key-value pair inserts cause costly data movements to maintain the array order.

In contrast, NVTable exploits NVM byte-addressability to organize key-value pairs in an unsorted doubly linked list called *KV list*. A NVTable insertion is just a simple, sequential KV list append without extra data movements as the SSTable. Although NVTable provides fast data write, however, unsorted data layout degrades data read performance. A read operation should traverse the KV list and visit each key-value pair of each list node. To address this shortcoming, NVTable introduces an offset array design. Key-value pair order in the NVTable is the offset array index, and its memory offset is the offset array element. With the offset array, threads can quickly find the requested key-value pair in the NVTable by performing a binary search.

### C. Skiplist-based NVTable Index

Multiple NVTables at each LSM-tree level are organized in a skiplist optimized for NVM. Similar to the traditional LSM-tree, NVTables at every level form a hierarchical tree structure. Multiple NVTables within the same level have a non-overlapping key range except for the $L_0$ level. $L_0$ level contains NVTables which are directly flushed from the DRAM without any further compaction or merging. Their key ranges may be overlapped.

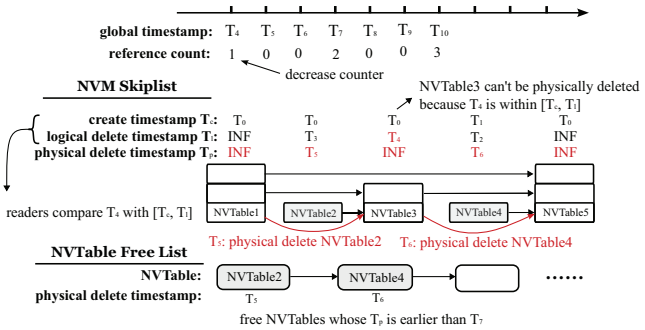


Figure 2: NVM Skiplist and SMR Mechanism.

NV-Cache uses a persistent skiplist to index NVTables in each LSM-tree level except the $L_0$ level. The index keys are the maximum keys of each NVTable. In contrast, NVTables in the $L_0$ level are organized as an ordered doubly linked list, sorted by their sequence numbers. It is because searching the NVTables in the $L_0$ level must obey a time order. Sequence numbers are monotonously increasing timestamps that are generated in NVTable creation.

NV-Cache provides lock-free concurrency control for reading and scanning threads. As shown in Figure 2, each NVTable has three timestamps: create timestamp $T_c$, logical delete timestamp $T_l$, and physical delete timestamp $T_p$. The create timestamp is the time when the NVTable is inserted into the skiplist during compaction. Moreover, when a thread accesses

a NVTable, it creates a global timestamp and an associated reference counter for this NVTable read. Then, it compares the current timestamp $T_4$ with the create timestamp $T_c$ and logical delete timestamp $T_l$. If $T_c \leq T_4 \leq T_l$, it indicates this NVTable is valid. The thread increases the reference counter and reads the NVTable.

For a skiplist, there are multiple readers and a compaction thread. The skiplist lookup is lock-free. Suppose the key-value item is located in the LSM-tree bottom layer. The thread first performs lookup on the MemTable. Then, it searches the NVTables at the $L_0$ level. It first searches the most recent NVTables using the offset array. Afterward, it traverses every skiplist starting from the $L_1$ level. It tries to find the first NVTable whose key is greater than or equal to the request key. If there exists one in the current search level, it checks whether this NVTable is valid by comparing the timestamp. It then performs a binary search with the offset array in NVTable. If there is an item whose key matches the requested key, it returns this item. Otherwise, such a KV item does not exist.

In NV-Cache, writers never modify a key-value pair except for the NVTable which has a corresponding MemTable in DRAM. During compaction, the background thread generates a new NVTable at a time. These old version NVTables can only be freed when no readers refer to them. As a result, no sophisticated techniques are required to ensure coordination between readers and writers.

**Safe memory reclamation.** Removing an NVTable takes two steps to achieve safe memory access to NVTables. In the first step, the thread first marks the NVTable as logically deleted and sets the associated timestamp. When there are still readers referring to this NVTable (e.g., $NVTable3$ in Figure 2), it can not be physically deleted.

Otherwise, the skiplist node of this NVTable is physically removed in the second step. It also sets the physical delete timestamp for this NVTable and appends it to the free list. When a thread finishes the NVTable reading, it decreases the reference counter for the current timestamp. If the counter reaches zero, it scans the free list and reclaims these NVTables whose physical delete timestamp is less than the current timestamp. For example, $NVTable2$ and $NVTable4$ can be freed since their $T_p$ is less than $T_7$.

### D. Lightweight MemTable Flush

Lightweight MemTable flush reduces the flush latency by removing the key-value data persist off the execution path. It is achieved by our unified write-ahead log and NVTable design. As illustrated in Figure 3, the fast MemTable flush consists of three steps. First, NV-Cache creates an associated NVTable for the DRAM MemTable. Before the thread inserts a new key-value pair into the MemTable, it first creates a persistent key-value pair and appends this item to the KV list in the associated NVTable in step ① instead of creating a new write-ahead log. The crash consistency of this NVTable item insert is achieved by using PMDK transaction [21]. Removing the write-ahead log also improves recovery performance since NV-Cache does not need to handle write-ahead logs.

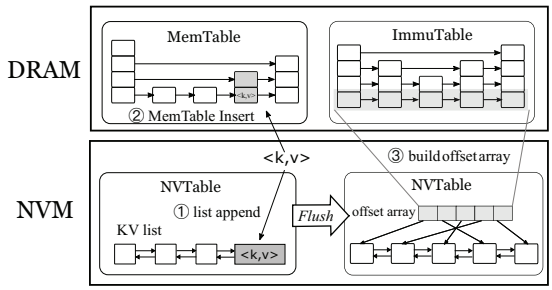


Figure 3: Lightweight MemTable Flush.

Second, the thread inserts a new item into the MemTable in DRAM in step ②. When the MemTable becomes full, it is changed to an ImmuTable. NV-Cache creates a new empty MemTable and NVTable in DRAM and NVM, respectively. In step ③, a background thread flushes the ImmuTable to the $L_0$ level in the LSM-tree. Fortunately, key-value data in the ImmuTable is already persisted in the NVTable. In this step, we only scan the bottom level of the ImmuTable skiplist and sort key-value pairs in the NVTable to build the offset array. After the thread finishes building the offset array for NVTable, a completed NVTable is generated, and this ImmuTable can be freed.

### E. Write-optimal List Compaction

In conventional LSM-tree compaction, it picks one or multiple victim SSTables, loads them into the DRAM, and writes a new SSTable into the SSD after merging, sorting, and removing redundant key-value data. The write amplification issue raises as only a small portion of redundant key-value data is removed. The remaining data writes are unnecessary. After compaction, these victim SSTables can be freed. To tackle this issue, NV-Cache designs write-optimal list compaction to reduce write amplification.

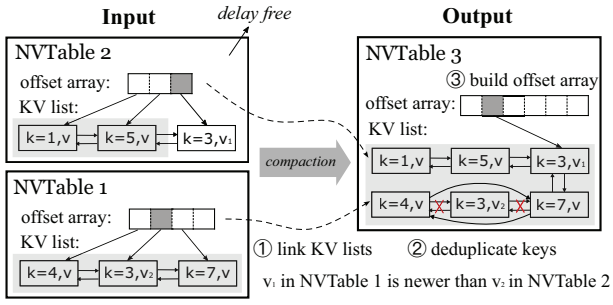


Figure 4: Illustrative Examples of List Compaction.

Suppose the size of a LSM-tree level $L_k$ exceeds the predefined threshold, list compaction picks a victim NVTable randomly. Then it finds those NVTables in the lower $L_{k+1}$ level whose key ranges overlap with the victim NVTable. Together, these NVTables are input files of list compaction, as shown in Figure 4.

In step ①, our algorithm links KV lists of input NVTables to get a new, longer KV list for a new NVTable. Then, list compaction performs key-value data deduplication on the new

KV list in step ②. If duplicate keys are detected, we only add the list node with the newer data version to the new KV list. The list node with the older data version remains unchanged in its original NVTable. Because there could be other threads referring to this old data, we delay freeing this NVTable based on our safe memory reclamation mechanism. Finally, it rebuilds the offset array for the new NVTable and logically delete these input NVTables in step ③.

List compaction adjusts the previous and next pointers of a list node to move a key-value pair instead of copying the real data. As write amplification is mainly caused by a large number of data movements, list compaction significantly reduces such overheads by simply modifying four eight-byte pointers for each list node. Suppose the input NVTable number is $N$. In the best case, the total write amount of list compaction is $32 \times (N-1)$ which is irrespective of the input key-value pair number.

### F. LRU-based NV-Cache Eviction

NVM has better performance and higher endurance than SSD. Therefore, keeping data written in NVM is especially important for ensuring NV-Cache performance. However, the cached LSM-tree has a size limitation, data will be evicted from the NVM to the SSD. To minimize the data compaction on the SSD, we propose a LRU-based NV-Cache eviction policy. Our insight into cache eviction policy is that the least frequently updated NVTable contains the coldest data. As a result, evicting such NVTables to the SSD can ensure hot data are likely to be kept in the cached LSM-tree, thereby reducing LSM-tree compaction in SSD.

Specifically, we build a global LRU list for NVTables stored across LSM-tree levels. The newly created NVTable will be inserted into the LRU list head because it is the hottest. In list compaction, we pick the NVTable which is in the LRU tail node as the victim since it is the coldest. If the victim NVTable resides in the lowest LSM-tree level, it will be evicted from the NVM cache and compacted to the LSM-tree stored in the SSD. The compaction procedure is similar to a regular LevelDB compaction except it uses an NVTable as the input. Otherwise, it will be compacted to the next level of the cached LSM-tree.

### G. NV-Cache Recovery

Key-value data in an NVTable is modified in two situations: when creating new NVTables during compaction or inserting a key-value pair in the NVTable during a MemTable insertion. In these two situations, crash consistency of key-value pair insertion and skiplist node insertion is achieved using PMDK transactions. However, a skiplist has several upper levels. To reduce the crash consistency performance overheads, we do not use the PMDK transaction to guarantee the crash safety of the skiplist upper layer modification. During recovery, NV-Cache first uses the skiplist bottom layer to reconstruct the upper layers from a top-down order for the cached LSM-tree.

## IV. Evaluation

### A. Experiment Setup

We implement NV-Cache based on Google LevelDB [1]. We use Intel PMDK [21] to manage mapped NVM memory space. The implementation effort of NV-Cache is approximately 5000 lines of C++ code. Experiments are conducted on a two-socket Intel machine. This server has two Intel Xeon Gold 5220R processors. Each processor has twenty-four physical cores running at 2.2 GHz. This server is equipped with 1.5TB (12×128GB) Intel Optane DC persistent memory and 192GB (12×16GB) DDR4 DRAM. Besides, it also has a 1TB Crucial BX500 SATA SSD.

We compare NV-Cache with Google LevelDB [1], NoveLSM [9], and MatrixKV [7]. NoveLSM is developed based on LevelDB and employs NVM to store large MemTable to avoid data serialization and write-ahead log overheads. MatrixKV is implemented based on RocksDB. It stores LSM-tree $L_0$ level in NVM and uses fast matrix compaction. We use db_bench in RocksDB as the microbenchmark. In addition, we also use the YCSB workload to evaluate NV-Cache performance in realistic scenarios. We use a fixed-size 16-byte key and variable-sized value in the experiment.

### B. Microbenchmarks

*1) Write without Preload:* We analyzed key-value store load performance without any data preloading. This experiment loads 5 GB of data into the KVS. Then we measure the average latency and throughput of inserting a key-value pair with variable value sizes ranging from 256 to 2048 bytes.

Fig. 5 shows the experimental results. As the value grows, the throughput of NV-Cache improves from 31MB/s to 84MB/s, which is 1.4 and 1.7× higher than LevelDB, respectively. Compared to NoveLSM and MatrixKV, NV-Cache achieves 16% and 54% higher throughput, respectively.

To fully understand the impact of value size on latency, we measured the tail latency of each operation using a 1KB value. As shown in Figure 7, the 90th percentile latency of NV-Cache, NoveLSM, and MatrixKV are comparable. However, their 99th percentile latency is two orders of magnitude lower than LevelDB.

For 99.5th percentile latency, NV-Cache tail latency is one order of magnitude lower than NoveLSM and MatrixKV. The completion percentage for each job demonstrates a significant increment in tail latency. It shows that the write stall causes a delay of up to $10^3$ microseconds in the worst case. For NV-Cache, only 0.3% of key-value insertion trigger write stall compared to 9.4% in LevelDB, leading to a $10^3$ microsecond execution latency.

*2) Write with Preload:* We evaluate key-value store *put* performance with preloading data. In this evaluation, we first load 5 GB of data into the database. Then we measure the average latency of inserting 5 GB of data into the database. The result is shown in Figure 8. The throughput of NV-Cache is 1.7× and 1.2× higher than LevelDB and MatrixKV but similar to NoveLSM when using 256-byte values. The performance gap between NV-Cache and LevelDB increases when the value size

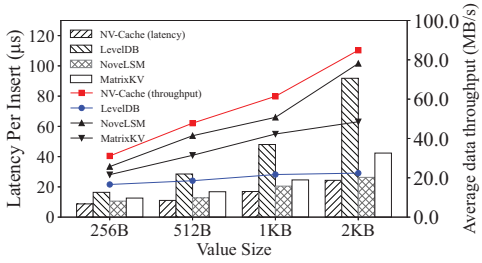


Figure 5: Random load

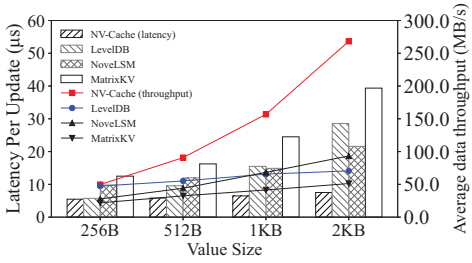


Figure 6: Skewed insert

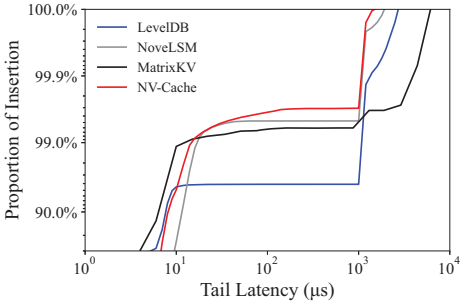


Figure 7: Tail latency of random load

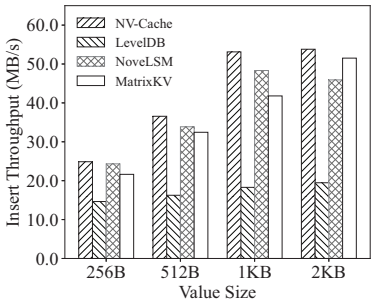


Figure 8: Random insert

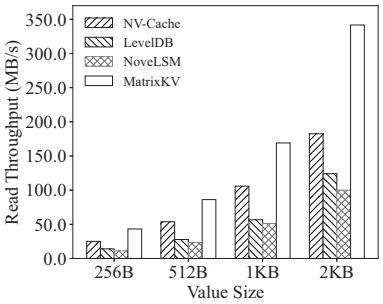


Figure 9: Random read

grows. The throughput of NV-Cache with 2KB values is 2.8× and 1.2× higher than LevelDB and NoveLSM but similar to MatrixKV. It is because the dataset generated by db_bench is uniformly distributed. Therefore, NV-Cache has a low cache hit rate which affects its performance.

Besides, we also evaluate the key-value store performance with a skewed dataset. Our evaluation begins with preloading the database with 5 GB of data. Then we use a skewed workflow to update up to 5 GB of data. In our dataset, the hottest 1% of keys accounted for 99% of all update requests.

As depicted in Figure 6, the performance of NV-Cache is comparable to that of LevelDB when 256-byte values are used, but it is about 2× higher than NoveLSM and MatrixKV. The throughput of LevelDB, NoveLSM, and MatrixKV improves slowly when the value size increases. However, the throughput of NV-Cache increases dramatically, which is 3.8×, 2.9×, and 5.2× higher than LevelDB, NoveLSM, and MatrixKV.

The list compaction algorithm is designed to compact NVTables and clear up redundant data with minimal overhead. It ensures that the compaction overhead is only related to the number of key-value pairs rather than the actual data size. As a result, NV-Cache performance improves as the value size increases.

*3) Read Performance:* We also evaluated the database read performance. In the following evaluation, we insert 5 GB data into the database and then look up the same amount of data randomly. We perform read on the database twice and use the result of the second test to avoid the impact of incomplete compaction in the background.

The results of this experiment are shown in Figure 9. The read throughput of NV-Cache is 46% lower than MatrixKV but 1.82× and 1.47× higher than NoveLSM and LevelDB. When handling the 100% read workload, the DRAM cache of RocksDB is optimized for point queries, which makes it the fastest. NV-Cache optimizes write performance by allowing reading queries to take full advantage of the locality of newly inserted data which is not designed for pure read workloads. Compared with NoveLSM and LevelDB, the read performance of NV-Cache is acceptable.

### C. YCSB Benchmarks

To evaluate the NV-Cache performance in real-world cloud applications, we use YCSB [22], a benchmark suit that simulates the workflows of cloud servers to test the performance of key-value stores.

The YCSB experiment procedure consists of two phases. First, an amount of data is loaded into the database to simulate a warm working environment, and then the testing procedure is carried out with the preloaded data. Each workload preloads identical data but a unique combination of reading, writing, inserting, and updating requests in the second phase. We use five workloads in the YCSB benchmark suit. The proportion of different operations in each workload is marked in the captions from Figure 10 to Figure 15.

During the loading phase, we load 5 GB data into the database with 5 million key-value pairs using a value size of 1000 bytes and a key size of 23 bytes. During the running phase, four threads submit 5 million operation requests to the database concurrently.

First, for YCSB preload, LevelDB needs to merge $L_0$ data with overlapping key ranges with multiple $L_1$ SSTables in SSD. NV-Cache, NoveLSM, and MatrixKV put $L_0$ data in NVM and pre-organize more data before compacting them to SSD. So the data loading performance is much higher than that of LevelDB. NV-Cache achieves 3.1× the data load throughput of LevelDB and improves the load performance by roughly 13% compared to MatrixKV and NoveLSM.

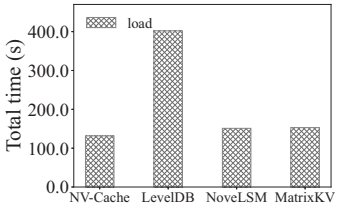


Figure 10: YCSB load

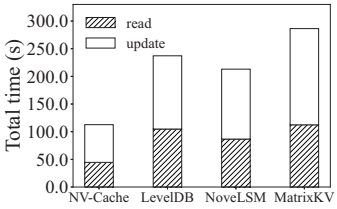


Figure 11: Workload A

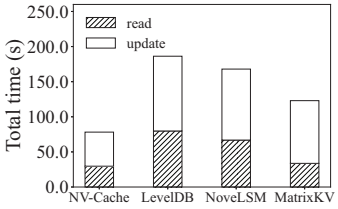


Figure 12: Workload B

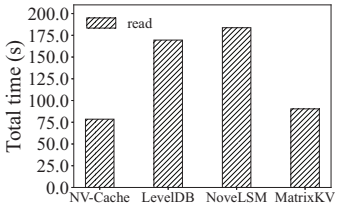


Figure 13: Workload C

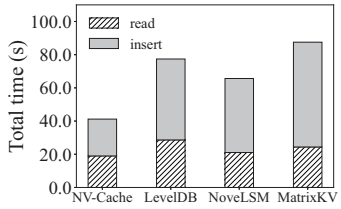


Figure 14: Workload D

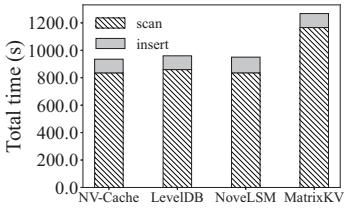


Figure 15: Workload E

For workload A, which has a uniform mix of reading and writing requirements, the throughput of NV-Cache is approximately $2.1\times$, $1.9\times$, $2.6\times$ higher than that of LevelDB, NoveLSM, and MatrixKV, respectively. NV-Cache accelerates data updates and retains the newly updated data in NVM via the list compaction algorithm, enhancing the localization of data reads and writes and resulting in improved performance in workflows with mixed reads and writes.

For read-oriented workloads, In workloads B and C, NV-Cache data read performance is comparable to MatrixKV, but it outperforms LevelDB and NoveLSM by around $2\times$. NV-Cache takes almost $0.5\times$ higher than the other three databases to handle the update queries in workload B. In workload D, NV-Cache offers $4.7\times$ and $3\times$ higher throughput than MatrixKV and NoveLSM. At the same time, read performance is $49\%$, $11\%$, and $28\%$ higher than LevelDB, NoveLSM, and MatrixKV.

For workload E, NV-Cache skiplist index for NVTables within NVM supports both point and range queries. However, because scanning the whole database cannot avoid visiting every level in SSDs, we cannot expect a large performance boost from fast retrieval in NVM. MatrixKV is based on RocksDB, which is not optimized for the scan. RocksDB's SSTable design favors point queries rather than range queries, resulting in relatively low scan performance.

## V. Related work

### A. Improve LSM-tree write performance

LSM-trie [23] and PebblesDB [24] reduce the number of SSTables in compaction. They control the upper limit of write amplification. NV-Cache achieves compaction with extremely low write amplification using the byte-addressable NVM. Wisckey [25] employs a key-value separation policy so that it doesn't have to rewrite the value during compaction. As a trade-off, Wisckey has to schedule extra background work to remove redundant data because its compaction can't free the value that has been deleted or updated.

NoveLSM [9] designs a persistent MemTable in NVM. However, NoveLSM only stores the MemTable in the NVM, which cannot reduce the total overhead of compaction. SLM-DB [26] uses a single-level structure to avoid level-by-level compaction and employs a $B^+$ tree to index all the key-value pairs in the single-level LSM-tree. SLM-DB still needs to schedule background work to clean up redundant data on a single level and rewrite the actual key-value pair data when merging SSTables.

MatrixKV [7] stores the LSM-tree $L_0$ level in NVM. When compacting $L_0$ SSTables, it selects data with the same range from overlapped $L_0$ SSTables and merges them with a number of $L_1$ SSTables with the matrix compaction algorithm. MatrixKV reduces write stalls caused by the inefficient compaction from $L_0$ to $L_1$. MatrixKV does not distinguish between hot and cold data. ListDB [27] constructs each SSTable in NVM as a NUMA-friendly NVM skiplist. It merges multiple SSTables without any data rewriting by changing the pointer in skiplists rather than the actual data. For compaction in NVM, it will cause larger write amplification to dynamically insert key-value pairs to the skiplists while NV-Cache only has to maintain a doubly linked list and an offset array for each SSTable.

### B. NVM Key-Value Store

NV-tree [28] and HiKV [29] optimize the $B^+$ tree and hash index for NVM. NV-tree [28] proposes a $B^+$ tree in that the leaf nodes are stored in NVM while the upper layer nodes are stored in DRAM to reduce the overhead of ensuring data crash consistency. It also employs unordered entries in the leaf nodes of the $B^+$ tree to reduce adjusting overhead. HiKV [29] builds hybrid indexes in DRAM and NVM. It uses $B^+$ tree in DRAM to index hash tables in NVM. The hash table in NVM is updated synchronously, and the $B^+$ tree in DRAM is updated asynchronously, which guarantees crash consistency while hiding the delay of adjusting the $B^+$ tree.

Bullet [30] and FlatStore [31] improve the performance of the index in NVM by optimizing the NVM flush. Bullet [30]

maintains two hash tables in NVM and DRAM and leverages NVM to persist the index. It employs a cross-referencing log to move the process of writing from DRAM to NVM out of the critical path. FlatStore [31] performs NVM writes at 256-byte granularity by writing them to NVM in batches.

## VI. Conclusion

In this paper, we propose NV-Cache, a write-optimized non-volatile cache to improve the write performance of LSM-tree-based key-value stores. We design a split LSM-tree structure on NVM-SSD storage architecture with a range of techniques to extensively optimize the write path. Experiment results show that NV-Cache significantly improves data write performance by up to $4\times$, reduces $2.3\times$ write amplification, and effectively prevents the write stall.

## Acknowledgments

We thank the reviewers for their valuable feedback. This paper is supported by Fundamental Research Funds for the Central Universities (No. B220202073), CCF-Huawei Innovation Research Plan (No. CCF2021-admin-270-202101), Natural Science Foundation of Jiangsu Province (No. BK20220973), Future Network Scientific Research Fund Project (No. FNSRFP-2021-ZD-7), China Postdoctoral Science Foundation (No. 2022M711014), Jiangsu Planned Projects for Postdoctoral Research Funds (No. 2021K635C). Miao Cai is the corresponding author.

## References

[1] Leveldb. [Online]. Available: https://github.com/google/leveldb

[2] Y. Matsunobu, S. Dong, and H. Lee, "Myrocks: Lsm-tree database storage engine serving facebook's social graph," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 3217–3230, 2020.

[3] Cassandra. [Online]. Available: https://cassandra.apache.org/

[4] Z. Cao, S. Dong, S. Vemuri, and D. H. C. Du, "Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook," in *USENIX Conference on File and Storage Technologies*, 2020, pp. 209–223.

[5] L. Yang, H. Wu, T. Zhang, X. Cheng, F. Li, L. Zou, Y. Wang, R. Chen, J. Wang, and G. Huang, "Leaper: A learned prefetcher for cache invalidation in lsm-tree based storage engines," *Proc. VLDB Endow.*, vol. 13, no. 11, pp. 1976–1989, 2020.

[6] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A Distributed Storage System for Structured Data," *ACM Transactions on Computer Systems*, vol. 26, no. 2, pp. 1–26, 2008.

[7] T. Yao, Y. Zhang, J. Wan, Q. Cui, L. Tang, H. Jiang, C. Xie, and X. He, "MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM," in *USENIX Annual Technical Conference*, 2020, pp. 17–31.

[8] T. E. Anderson, M. Canini, J. Kim, D. Kostic, Y. Kwon, S. Peter, W. Reda, H. N. Schuh, and E. Witchel, "Assise: Performance and Availability via Client-local NVM in a Distributed File System," in *USENIX Symposium on Operating Systems Design and Implementation*, 2020, pp. 1011–1027.

[9] S. Kannan, N. Bhat, A. Gavrilovska, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Redesigning LSMs for Nonvolatile Memory with NoveLSM," in *USENIX Annual Technical Conference*, 2018, pp. 993–1005.

[10] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's Highly Available Key-value Store," in *ACM Symposium on Operating Systems Principles*, 2007, pp. 205–220.

[11] D. Basin, E. Bortnikov, A. Braginsky, G. Golan-Gueta, E. Hillel, I. Keidar, and M. Sulamy, "KiWi: A Key-Value Map for Scalable Real-Time Analytics," in *ACM Symposium on Principles & Practice of Parallel Programming*, 2017, pp. 357–369.

[12] O. Balmau, D. Didona, R. Guerraoui, W. Zwaenepoel, H. Yuan, A. Arora, K. Gupta, and P. Konka, "TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores," in *USENIX Annual Technical Conference*, 2017, pp. 363–375.

[13] C. Luo and M. J. Carey, "LSM-based Storage Techniques: a Survey," *VLDB Journal*, vol. 29, no. 1, pp. 393–418, 2020.

[14] Y. Li, Z. Liu, P. P. C. Lee, J. Wu, Y. Xu, Y. Wu, L. Tang, Q. Liu, and Q. Cui, "Differentiated Key-Value Storage Management for Balanced I/O Performance," in *USENIX Annual Technical Conference*, 2021, pp. 673–687.

[15] K. Huang, Z. Jia, Z. Shen, Z. Shao, and F. Chen, "Less is More: De-amplifying I/Os for Key-value Stores with a Log-assisted LSM-tree," in *International Conference on Data Engineering*, 2021, pp. 612–623.

[16] H.-S. P. Wong, S. Raoux, S. Kim, J. Liang, J. Reifenberg, B. Rajendran, M. Asheghi, and K. Goodson, "Phase Change Memory," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2201–2227, 2010.

[17] H. Akinaga and H. Shima, "Resistive Random Access Memory (ReRAM) Based on Metal Oxides," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2237–2251, 2010.

[18] F. T. Hady, A. P. Foong, B. Veal, and D. Williams, "Platform Storage Performance With 3D XPoint Technology," *Proceedings of the IEEE*, vol. 105, no. 9, pp. 1822–1833, 2017.

[19] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An Empirical Guide to the Behavior and Use of Scalable Persistent Memory," *Usenix magazine*, vol. 45, no. 3, 2020.

[20] Intel optane technology delivers new levels of endurance. [Online]. Available: https://www.intel.co.id/content/www/id/id/products/docs/memory-storage/optane-technology/delivering-new-levels-of-endurance-article-brief.html

[21] Persistent memory development kit. [Online]. Available: https://github.com/pmem/pmdk

[22] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *ACM Symposium on Cloud Computing*, 2010, pp. 143–154.

[23] 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 *USENIX Annual Technical Conference*, 2015, pp. 71–82.

[24] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham, "PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees," in *ACM Symposium on Operating Systems Principles*, 2017, pp. 497–514.

[25] 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 Transactions on Storage*, vol. 13, no. 1, pp. 1–28, 2017.

[26] O. Kaiyrakhmet, S. Lee, B. Nam, S. H. Noh, and Y. Choi, "SLM-DB: Single-Level Key-Value Store with Persistent Memory," in *USENIX Conference on File and Storage Technologies*, 2019, pp. 191–205.

[27] W. Kim, C. Park, D. Kim, H. Park, Y. ri Choi, A. Sussman, and B. Nam, "ListDB: Union of Write-Ahead Logs and Persistent SkipLists for Incremental Checkpointing on Persistent Memory," in *USENIX Symposium on Operating Systems Design and Implementation*, 2022, pp. 161–177.

[28] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He, "NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems," in *USENIX Conference on File and Storage Technologies*, 2015, pp. 167–181.

[29] F. Xia, D. Jiang, J. Xiong, and N. Sun, "HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems," in *USENIX Annual Technical Conference*, 2017, pp. 349–362.

[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 *USENIX Annual Technical Conference*, 2018, pp. 967–979.

[31] 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 *Internal Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 1077–1091.