WAL-assisted Tiering: Painlessly Improving Your Favorite Log-Structured KV Store Instead of Building a New One

Xubin Chen, Jingpeng Hao, Yifan Qiao, and Tong Zhang
Rensselaer Polytechnic Institute
chenx22@rpi.edu

ABSTRACT

This paper presents a simple design approach that can be easily integrated into existing mature log-structured key-value (KV) stores (e.g., RocksDB) to mitigate the impact of background compaction. Reducing compaction-induced performance degradation has been widely studied, and most prior work focused on developing innovative data structures and algorithms to directly reduce the compaction-induced write amplification. Nevertheless, it is non-trivial or even practically infeasible for existing mature KV stores to adopt these new data structures and algorithms. Meanwhile, in the presence of well-established ecosystem and community around existing ones, it is a challenge to build and grow a new log-structured KV store with meaningful real-world adoption. Therefore, this work focuses on mitigating the impact of compaction while keeping the data structures and algorithms in existing KV stores completely intact. Instead of directly reducing the write amplification, this work applies the simple memory/storage tiering concept to mitigate the impact of compaction at the cost of larger write-ahead log (WAL) and host memory capacity usage. This paper presents design approaches to effectively reduce the WAL size and memory cost. We integrated this solution into RocksDB by only adding about 1,200 lines of code, without touching its core data structure and algorithm. Using 100GB and 1TB datasets as test vehicles, we carried out experiments with db_bench and YCSB workloads, and the results show that the modified RocksDB can improve the ops/s by up to 100.7% and meanwhile reduce the 99-percentile tail latency by up to 82%.

CCS CONCEPTS

• **Information systems** → *DBMS engine architectures*; *Storage class memory*; • **Hardware** → *Memory and dense storage*; • **Computer systems organization** → *Reliability*.

KEYWORDS

Log Structured Merge Tree, Key Value Store, Block-protected DRAM, Error Correction Code (ECC), Crash Recovery

ACM Reference Format:
Xubin Chen, Jingpeng Hao, Yifan Qiao, and Tong Zhang. 2020. WAL-assisted Tiering: Painlessly Improving Your Favorite Log-Structured KV Store Instead of Building a New One. In *The International Symposium on Memory Systems (MEMSYS 2020), September 28-October 1, 2020, Washington, DC, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3422575.3422802

MEMSYS 2020, September 28-October 1, 2020, Washington, DC, USA

ACM ISBN 978-1-4503-8899-3/20/09…$15.00
https://doi.org/10.1145/3422575.3422802

1 INTRODUCTION

With unique advantages on handling write-intensive data, key-value (KV) store built upon log-structured merge (LSM) tree [25] has been heavily studied over recent years [23]. In addition to proprietary implementations (e.g., Ressi in Google's Spanner [3]), several open-source log-structured KV stores (e.g., RocksDB [30] and Cassandra [1]) have become increasingly mature with an ever-growing feature list and are being widely deployed. Compaction is an essential (and arguably the most important) operation in log-structured KV stores. Compaction-induced write amplification causes two issues: (1) interference with foreground operations (e.g., *GET* and *INSERT*), leading to KV store performance degradation; (2) impact on the solid-state drive (SSD) lifetime, especially SSDs built with low-cost 3D QLC NAND flash memory. Aiming to mitigate these two issues, most prior work [4, 11, 15, 22, 28, 29, 32, 34, 36] focused on directly reducing the write amplification through innovations on the KV store data structures and algorithms. Given the well-established ecosystem and community around existing ones such as RocksDB, integrating these new design techniques into those KV stores is much more preferred and practically valuable than building new log-structured KV stores from the ground. Nevertheless, it can be difficult or even impractical for existing KV stores to change their core implementations in order to incorporate these new data structures and algorithms.

This paper advocates an alternative approach to mitigate the impact of write amplification, which can be easily integrated into existing log-structured KV stores. In contrast to prior work, we do not aim at reducing the write amplification at all, and hence obviate any changes to the core data structures and algorithms of existing KV stores. The basic idea is motivated by a simple observation: In log-structured KV stores with leveled compaction, different levels occupy exponentially different storage capacity, but account for similar write traffic volumes. Let $\widetilde{C_i}$ denote the runtime size of the level-i (or L_i) and α denote the *level multiplier* (typical value is around 10), KV stores aim to keep $\widetilde{C_{i+1}} \approx \alpha \cdot \widetilde{C_i}$ through compaction. As a result, the bottom 1~2 levels completely dominate the storage capacity of the entire KV store. Intuitively, if we move a few top levels (e.g., $L_0 \sim L_2$) from SSD into host DRAM and meanwhile *expand* the write-ahead log (WAL) to ensure the crash recovery of those in-memory levels, we can substantially reduce the write traffic over SSD by occupying a relatively small amount of host DRAM. This can be considered as *WAL-assisted tiering* of KV stores across memory and SSDs. Although this simple approach does not reduce the CPU overhead of compaction, it can reduce the IO traffic

over SSD during compaction, leading to less compaction-induced KV store performance variation and longer SSD lifetime.

WAL-assisted tiering can be easily integrated into existing KV stores without impacting their core data structures and algorithms. Nevertheless, its practical realization comes with overhead in terms of larger WAL on SSD and higher host memory usage. In current practice, since WAL is only responsible for protecting a small in-memory write buffer (e.g., the *memtable* in RocksDB), its size is simply upper-bounded by the in-memory buffer size. However, in the case of WAL-assisted tiering, the WAL size can be significantly larger than the in-memory tier size. This is because the order in which KV pairs are appended into WAL can be completely different from the order in which KV pairs migrate from the in-memory tier to the on-SSD tier through compaction. To address this issue, this paper presents a method that can tightly control the WAL size. Its basic idea is to trim (or prune) the KV pairs, which no longer reside in the in-memory tier, from the sealed WAL segments. As a KV store continues its background compaction operations, more and more KV pairs in sealed WAL segments will be either dropped or migrated from the in-memory tier to the on-SSD tier. Those KV pairs can be safely trimmed from the sealed WAL segment without impacting the KV store durability. To ensure its practical feasibility, we developed techniques that can identify those KV pairs without noticeably interfering the normal KV store operations. In addition, very intuitively we can use background data lossless compression to further reduce the storage space occupied by WAL at modest CPU usage. The above two strategies (i.e., KV pair trim and WAL compression) are completely orthogonal to each other and can be combined together to minimize the on-SSD WAL size.

The second problem is the high memory usage. By increasing the number of in-memory levels, the performance improvement is more significant at the penalty of higher memory cost. The upper limit of the in-memory tier size is around 10% of the dataset size (given typical value of 10 for level multiplier α) by moving all levels but last level from SSD to DRAM. In addition to adjusting the number of in-memory levels to trade-off performance improvement vs. cost, we further present a strategy to reduce the memory cost. Although the long-waited non-volatile memory (NVM) technologies (e.g., 3DXP and ReRAM) may reduce the memory cost overhead (and even obviate the use of WAL), there are still substantial uncertainties about NVM technologies, e.g., real-world adoption in the foreseeable future, technology scalability and cost, and achievable bandwidth and cycling endurance. Therefore, this work does not conveniently assume the availability of high-bandwidth NVM modules on the host, and only considers the DRAM-SSD tiering. Hynix recently demonstrated the practical feasibility of reducing DRAM cost by giving up its true byte-accessibility [20]. The basic idea is to reduce DRAM manufacturing cost by sacrificing the raw memory reliability, and meanwhile rely on stronger ECC (error correction code) with long codeword length (e.g., hundreds of bytes) to ensure the data integrity. We call such DRAM as *block-protected* DRAM. In fact, to reduce the memory cost, each Intel 3DXP-based Optane DC Persistent Memory DIMM internally protects 256-byte user data per ECC codeword [16]. It is plausible that, when using block-protected DRAM other than today's byte-accessible DRAM as the baseline, the proclaimed bit cost advantage of 3DXP may largely diminish. With coarse-grained access pattern (e.g., 4KB or 8KB per access) for read, flush and compaction operations, the in-memory tier of the proposed WAL-assisted KV store tiering can naturally fit to the low-cost block-protected DRAM. Therefore, we propose to reduce the memory cost overhead by deploying a DRAM-only heterogeneous memory system that consists of convenient byte-accessible DRAM and low-cost block-protected DRAM. We accordingly studied its impact on the KV store performance by emulating the longer access latency of block-protected DRAM.

As a case study, we modified RocksDB to support the proposed WAL-assisted tiering by adding only about 1,200 lines of code and meanwhile keeping all the existing core algorithms and data structures intact. We ran a variety of workloads on RocksDB with small (100GB) and large (1TB) datasets, and the results well demonstrate the effectiveness of the proposed WAL-assisted tiering approach. For example, when running a write-only workload on a 1TB dataset, by allocating only 14GB block-protected DRAM for in-memory tier (with three in-memory levels), we can improve the RocksDB ops/s by 49.8%, and meanwhile keep the WAL size below 22GB and reduce the SSD write traffic volume by 53.2%. In summary, this paper makes the following main contributions:

(1) It for the first time carries out a thorough study on applying the simple WAL-assisted tiering concept to lessen the impact of write amplification without changing the core algorithms and data structures of existing log-structured KV stores;
(2) It presents techniques that can tightly control the WAL size and reduce the host memory cost in order to facilitate the practical implementation of WAL-assisted KV store tiering;
(3) It demonstrates the simplicity and practical feasibility of WAL-assisted tiering in the context of RocksDB, and the experimental results well confirm its effectiveness.
(4) It demonstrates the effectiveness of the WAL-assisted tiering on today's byte-addressable DRAM and the envisioned heterogeneous DRAM with emulated block-protected DRAM.

2 BACKGROUND AND MOTIVATION

Log-structured KV store is fundamentally subject to a trade-off among write amplification, read amplification, and storage space amplification [2]. One could explore such a trade-off space by employing different compaction strategies and/or adjusting the configuration of the chosen compaction strategy. As the default option in RocksDB, the *leveled* compaction strategy trades higher write amplification for lower storage space amplification (hence less storage cost). Nevertheless, a higher write amplification directly results in a higher CPU and IO overhead during compaction, which can noticeably degrade the KV store performance. Therefore, most prior research (e.g., see [4, 28, 34, 36]) focused on reducing the write amplification for log-structured KV stores with leveled compaction. In this work, we also focus on the use of leveled compaction.

KV stores with leveled compaction organize all the KV pairs into multiple levels, denoted as $L_0, L_1, \cdots, L_n$. Each level further organizes KV pairs into multiple files. All files in each level, except L_0, have non-overlapping key ranges. Let C_i denote the target size of the level L_i, we have $C_{i+1} = \alpha \cdot C_i$ for $i \geq 1$, where α is the level amplifier (e.g., around 10). KV stores aim to keep the runtime size of each level reasonably close to the target size, which is realized through background compaction. As a result, the per-level storage

capacity exponentially increases with the level, and the bottom 1~2 levels dominate the entire data storage usage. Meanwhile, as the compaction process pushes down KV pairs across the LSM tree, all the levels tend to experience similar write amplification and hence similar data read/write IO traffic.

For example, we created a 1TB KV store using RocksDB as follows: There are total 1 billion random KV pairs with each KV pair size of 1KB. There are total 6 levels with the size of 214MB, 74MB, 885MB, 8.4GB, 83.7GB, 583.7GB, respectively. We measured the per-level data write volume by running a write-only workload on this 1TB dataset. The results show that the ratio of data write volume among all the 6 levels is $1 : 0.9 : 2.9 : 3.8 : 3.9 : 2.6$. The above results clearly demonstrate that, although the per-level storage capacity increases exponentially (starting from level L_1), the per-level data write volume is comparable (especially among the bottom 4 levels).

3 PROPOSED DESIGN SOLUTION

3.1 Basic Idea

The discussion and results presented above in Section 2 directly motivate the basic idea underlying this work: WAL-assisted data tiering across host memory and SSD. Given a log-structured KV store with total $n+1$ levels (i.e., $L_0 \sim L_n$), we keep the top $m+1$ levels $L_0 \sim L_m$ (where $m < n$) in the host memory, and meanwhile expand the role of WAL to ensure the durability of the $(m+1)$-level in-memory tier. Fig. 1 further illustrates and compares the current practice (where all the levels reside in SSD) and WAL-assisted tiering. As illustrated in Fig. 1(b), WAL-assisted tiering leverages the host memory bandwidth to absorb a large portion of compaction-induced data read/write traffic, leaving much less IO traffic over SSD. Therefore, this simple design concept can reduce the KV store performance variation caused by the interference between the IOs of foreground operations and background compaction. Moreover, by reducing the amount of data being physically written to SSD, we can deploy KV stores on SSDs built with low-cost 3D QLC NAND flash memory that suffers from poor program/erase cycling endurance. Contrary to most prior work, this approach does not aim to reduce the write amplification at all, hence does not demand any changes to the core data structures and algorithms of existing KV stores. As a result, it can be much more easily integrated into existing matured log-structured KV stores.

Apparently, in return for its easy integration into existing KV stores, this approach incurs higher host memory usage and larger WAL size. Recall that C_i denotes the target size of the level L_i, and let w_i denote the average data write volume endured by the level L_i. By moving the top $m+1$ levels (out of total $n+1$ levels) into the in-memory tier, we consume on average $\sum_{i=0}^{m} C_i$ amount of host memory to reduce the SSD write IO traffic by

$$\gamma(m) = \frac{\sum_{i=0}^{m} w_i}{\sum_{j=0}^{n} w_j}. \quad (1)$$

As discussed above, the value of C_i exponentially increases with i, while the value of w_i tends not to largely vary across different levels. Therefore, by keeping a few top levels in memory, we can substantially reduce the SSD IO traffic at the cost of modest amount of host memory, which can be clearly observed from the example above in Section 2.

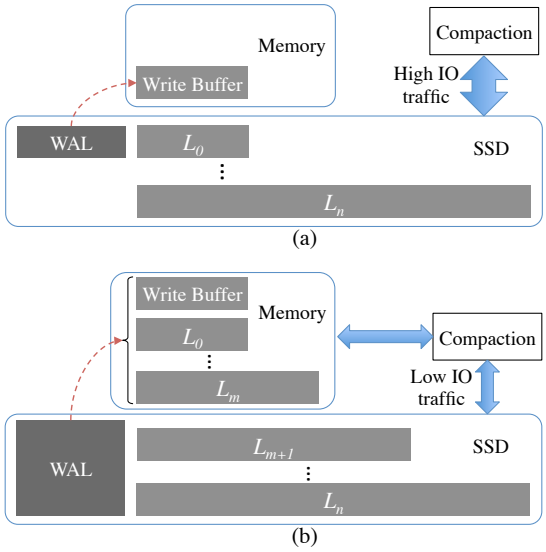

Figure 1: Illustration of (a) current practice, and (b) WAL-assisted tiering.

Moreover, WAL-assisted tiering could result in a very large on-SSD WAL. The average WAL size is lower bounded by $\sum_{i=0}^{m} C_i$, which however could be a very loose lower bound. WAL consists of multiple sealed segments (or files) and one open segment on SSD. KV stores append KV pairs into the open segment, and once its size reaches a pre-defined threshold (e.g., 64MB), the open segment will be sealed and a new open segment will be created. KV stores assign each inserted/updated KV pair a unique *sequence number* (SN) that monotonically increments (i.e., KV pairs logged in WAL are ordered with SN). Let N_{min} denote the SN of the oldest KV pair in the in-memory tier. Clearly, we cannot delete one WAL segment unless its newest KV pair is older than any in-memory KV pairs (i.e., the biggest SN within the segment is smaller than N_{min}). If the value of N_{min} remains unchanged over a long time, it will cause long-standing accumulation of sealed WAL segments, leading to a total WAL size much larger than $\sum_{i=0}^{m} C_i$.

The above discussions suggest that the practical feasibility of WAL-assisted tiering depends on (1) how tightly we can control the WAL size, and (2) how much we can reduce the memory cost overhead. The rest of this section presents strategies for addressing these two issues.

3.2 Controlling the WAL Size

Intuitively, we could reduce the WAL size by applying the *oldest-first* compaction scheduling to increase the value of N_{min} (i.e., the SN of the oldest in-memory KV pair). In order to keep the LSM tree in shape (i.e., make the size of each level close to its target value), compaction scheduling uses the total size of each level to determine the level from which a file will be chosen for compaction. Let C_i and $\widetilde{C}_i$ denote the target and runtime size of the level L_i. The relative level size deviation, defined as $\frac{\widetilde{C}_i - C_i}{C_i}$, can be used as a criterion for selecting the level. Let $\mathbb{D}$ denote the set consisting of all the files in the selected level. Compaction scheduling may use

different criteria to choose a file $d_i \in \mathbb{D}$ for compaction. Let $N(d_i)$ denote the SN of the oldest KV pair in the file d_i. If the compaction process chooses the file d_i so that

$$N(d_i) \leq N(d_j),\ \forall d_j \in \mathbb{D} \setminus \{d_i\}, \tag{2}$$

it is called oldest-first compaction scheduling, which evidently can increase the value of N_{min}. Unfortunately, oldest-first compaction scheduling may suffer from a higher write amplification overhead, compared with other compaction scheduling algorithms (e.g., the default compaction scheduling in RocksDB that chooses the file based on the key overlapping ranges). Moreover, different workloads may favor different compaction scheduling algorithms. Therefore, KV stores typically support several different compaction scheduling algorithms, from which users can choose based on the runtime workload characteristics. Enforcing the oldest-first compaction scheduling may not be acceptable in practice.

This work focuses on solutions that are independent from compaction scheduling algorithms. Hence, users can freely choose the compaction scheduling algorithm solely based on the runtime workload characteristics. The remainder of this subsection presents the basic idea and implementation of our proposed solution.

3.2.1 Basic Idea. To reduce the WAL size independently from compaction scheduling, one option is to apply *background data compression* to sealed WAL segments. In current practice, databases typically do not compress WAL because of its small size. Compressing WAL becomes much more desirable in the context of WAL-assisted tiering. Other than on-the-fly compressing the records being appended to WAL, we should compress each sealed WAL segment entirely in the background in order to (1) obviate latency penalty on the foreground KV store write commitment, and (2) improve the WAL compression ratio because of the strong dependency of compression ratio on the compression chunk size. After one WAL segment has become full and subsequently been sealed, it can be compressed in the background to generate a compressed WAL segment. Although background WAL segment compression could noticeably reduce the on-SSD WAL size (e.g., by 2×), itself cannot fundamentally solve the problem, and WAL may still grow to a very large size, even almost unboundedly.

To more tightly control the WAL size, we propose a method motivated by the following observation: Let $\mathbb{K}_i$ denote the set consisting of all the KV pairs in one sealed WAL segment, and let $\mathbb{M}_i \subset \mathbb{K}_i$ denote the set consisting of all the KV pairs that still reside in the in-memory tier. When the segment is just sealed, we have that $|\mathbb{M}_i| \approx |\mathbb{K}_i|$ (i.e., almost all the KV pairs remain in memory). Over the time, as the compaction process continues to migrate KV pairs from the in-memory tier to the on-SSD tier, $|\mathbb{M}_i|$ gradually reduces. However, as long as $\mathbb{M}_i \neq \emptyset$, we cannot delete this sealed WAL segment from SSD. Suppose there are total g sealed WAL segments that cannot be deleted, we define the WAL amplification factor as

$$\lambda = \frac{\sum_1^g |\mathbb{K}_i|}{\sum_1^g |\mathbb{M}_i|}, \tag{3}$$

where $\sum_1^g |\mathbb{K}_i|$ is the total size of sealed WAL segments, and $\sum_1^g |\mathbb{M}_i|$ is the total size of all the KV pairs residing in the in-memory tier. In essence, the WAL amplification factor $\lambda \geq 1$ represents the ratio between the on-SSD WAL size and in-memory tier size. Regardless of which compaction algorithm is being used, there could be a large number of sealed WAL segments for which $\mathbb{M}_i \neq$ and $|\mathbb{M}_i| << |\mathbb{K}_i|$, leading to a large WAL amplification factor λ.

The above observation suggests that, in order to tightly control the WAL size, we have to avoid the occurrence of $|\mathbb{M}_i| << |\mathbb{K}_i|$ on too many WAL segments. To achieve this objective, we propose a method, called *background KV trim*. Given one WAL segment, we can re-write this segment by *trimming* all the KV pairs that do not belong to the $\mathbb{M}_i$ (i.e., prune all the KV pairs that have already been either dropped or migrated into the on-SSD tier). After a segment has been re-written to SSD through background KV trim, its original larger-size version can be safely deleted from SSD. Once the WAL amplification factor λ grows beyond a pre-defined threshold $\lambda_t > 1$, we invoke the background KV trim process by starting from the segment that has the smallest $|\mathbb{M}_i| : |\mathbb{K}_i|$ ratio. This process continues until λ drops below another threshold $\lambda_s < \lambda_t$. This method ensures that the WAL size stays upper bounded by around $\lambda_t\times$ of the in-memory tier size. For its practical implementation, one major issue is how to keep track of the set $\mathbb{M}_i$ for each WAL segment. One option is to periodically scan the entire in-memory tier to construct all the $\mathbb{M}_i$'s, which however could interfere with normal foreground operations and hence impact the KV store performance. Next we will present a strategy that can more effectively implement the proposed background KV trim without interfering the normal foreground operations.

3.2.2 Implementation Strategy. Within each sealed WAL segment, let $SN^{(s)}$ denote the SN of the first (i.e., oldest) KV pair and w denote the total number of KV pairs (i.e., SN of all the KV pairs consecutively increments from $SN^{(s)}$ to $SN^{(s)}+w-1$). Accordingly, for each sealed WAL segment, we maintain an in-memory length-w *KV status* bitmap, in which each bit represents whether the corresponding KV pair still resides in the memory tier. In particular, for the k-th bit in one segment status bitmap, if its value is zero, it means that the k-th KV pair (with the SN of $SN^{(s)}+k-1$) has been either dropped or migrated to the on-SSD tier through compaction; otherwise (i.e., its value is one), the k-th KV pair still resides in memory. We can directly construct the set $\mathbb{M}_i$ of each sealed WAL segment using its KV status bitmap. For each segment, the value of w and $SN^{(s)}$ remain unchanged as the segment undergoes multiple rounds of background KV trim. Fig. 2 further illustrates this proposed implementation strategy.

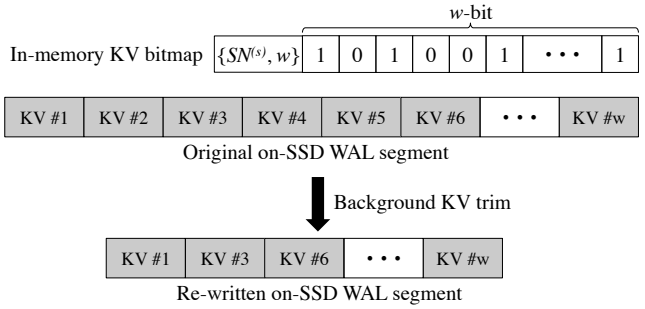

Figure 2: Illustration of using in-memory segment KV status bitmap to realize background KV trim.

To ensure the practical feasibility, we must update all the segment KV status bitmaps with minimal interference with foreground KV store operations. To achieve this objective, we add the following simple operation into the KV store compaction process: For each compaction operation, if at least one input file is an in-memory file, then the compaction process will generate an SN array containing the SN of all the KV pairs that are either dropped during the compaction or written into an on-SSD output file. If none of the input files is in-memory, the compaction process does not need to generate any SN array and operates exactly the same as normal. During the runtime, the KV store keeps collecting the SN arrays generated by the compaction process. Once the total size of all the collected SN arrays reaches a pre-specified threshold, the KV store invokes a background process to update all the segment status bitmaps. Let SN_{max} and SN_{min} denote the maximum and minimum SN among all the WAL segments. For each SN h in each SN array, we have

- If $h > SN_{max}$, then the corresponding KV pair still resides in an open WAL segment, and we keep this SN for the next bitmap update process.
- If $h < SN_{min}$, the corresponding KV pair has already migrated into the on-SSD tier, hence we can simply ignore this SN.
- If $SN_{max} \geq h \geq SN_{min}$, we find the segment status bitmap for which we have $SN^{(s)} \leq h \leq SN^{(s)} + w - 1$, and set the $(h - SN^{(s)} + 1)$-th bit in this bitmap as zero. Since all the KV status bitmaps are ordered in terms of SN, given any SN h, we can easily find the bitmap through a simple binary search.

Once the KV status bitmap of one segment becomes an all-zero vector, we can safely delete the corresponding WAL segment and drop the KV status bitmap from the memory. After each round of background KV status bitmap update, we accordingly re-calculate the WAL amplification factor of each segment and the overall average amplification factor, based on which we will determine whether another round of background KV trim should be invoked.

3.3 Crash Recovery

Since we keep the top $m + 1$ levels and several other data structures (e.g., segment KV status bitmaps) in host DRAM, we have to accordingly enhance the crash recovery procedure. In the case of system crash, we assume that all the in-memory data are lost, and hence have to reconstruct the entire $(m + 1)$-level in-memory tier. To achieve this objective, we scan all the WAL segments and insert all the logged KV pairs back into the KV store. Meanwhile, as we scan each WAL segment, we reconstruct the corresponding segment KV status bitmap. We note that each WAL segment always keeps the SN of its first KV pair (i.e., $SN^{(s)}$) and the total number of SNs (i.e., w) unchanged throughout its lifetime. Hence, regardless how many times one WAL segment has been processed with background KV trim, we can always reconstruct the segment KV status bitmap by scanning through each WAL segment. Crash recovery process initializes the set of SN arrays as empty and gradually adds SN arrays as the KV store carries out compaction operations.

Compared with conventional practice, our proposed design approach inevitably incurs a longer crash recovery latency because of the larger amount of in-memory data to be reconstructed. Nevertheless, because the in-memory tier accounts for a very small percentage of the total data volume and very high DRAM operational bandwidth, the crash recovery latency tends to be insignificant, which will be quantitatively demonstrated later in Section 4.

3.4 Reducing the Cost of In-memory Tier

The long-waited and widely-researched NVM technologies (e.g., 3DXP and ReRAM) may appear as the best option to reduce the cost of the in-memory tier. Moreover, due to their non-volatility, one may argue that NVM-based in-memory tier can obviate the use of WAL. Nevertheless, in spite of the recent launch of Intel Optane DC Persistent Memory, it is not clear yet how quickly and widely NVM-based memory modules will be deployed in production environment in the foreseeable future. Although the first generation Optane DC Persistent Memory shows reasonable metrics in terms of performance (i.e., ~10× longer latency and ~10× lower bandwidth than DRAM) and cycling endurance (i.e., up to a few million), it is not clear (at least in the open literature) how those important metrics will change with the 3DXP technology scaling.

In this work, we advocate an alternative **DRAM-only** approach to reduce the in-memory tier cost overhead. It is built upon a simple principle: Regardless of the specific memory technology, one may reduce the memory cost by relaxing the raw reliability, and meanwhile restore the data integrity via low-cost system-level fault tolerance, in particular error correction code (ECC). The error correction strength of ECC improves as its codeword length increases. Evidently, this principle involves a trade-off between the cost and native data access granularity. In fact, Optane DC Persistent Memory leverages this principle as well: Optane DC Persistent Memory DIMM internally uses a strong ECC that protects 256-byte user data in each ECC codeword [16]. As a result, regardless of the size of requests from host CPU, Optane DC Persistent Memory DIMM internally always operates with the unit of 256 bytes.

To further illustrate the trade-off, let us consider the following example: Suppose we use BCH code as ECC, and let r denote the ratio between the size of ECC redundancy and user data in each codeword. In today's commercial ECC DRAM DIMM, the ratio r is 1:8 and each codeword protects only 8-byte user data. Fig. 3 shows the BCH code decoding failure rate vs. raw bit error rate under different codeword length and ratio r. With the target decoding failure rate of 10^{-20}, under the same redundancy ratio of 1:8, increasing ECC codeword length from 8-byte (as in today's ECC DIMM) to 256-byte (as in Optane Persistent Memory) can improve the error correction strength by **eight orders of magnitude** (i.e., the tolerable bit error rate increases from 10^{-12} to 10^{-4}). Even if we reduce the ECC redundancy ratio from 1:8 to 1:16 (i.e., reduce the ECC coding redundancy by 50%) in the case of 256-byte user data per codeword, we can still achieve an improvement of seven orders of magnitude.

The above results suggest that the proclaimed bit cost advantage of 3DXP over DRAM in the open literature is not based on a fair comparison. We should compare the bit cost between DRAM and 3DXP (or any other NVM) under the condition of using ECC with the same (or at least comparable) error correction strength. In fact, since DRAM has a shorter access latency than NVM, we may even

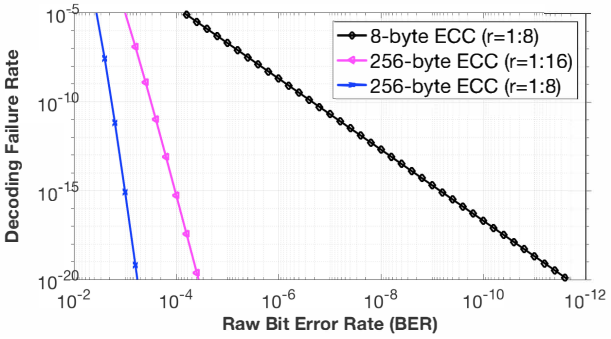

Figure 3: BCH decoding failure rate vs. raw bit error rate.

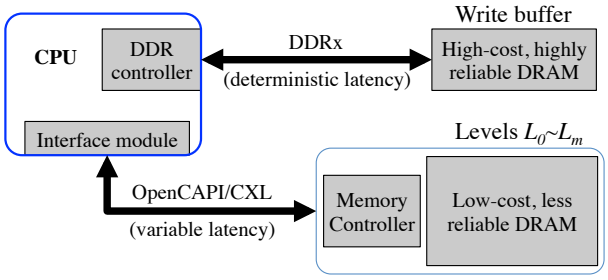

Figure 4: Illustration of storing the in-memory tier on a low-cost DRAM-only heterogeneous memory system.

use stronger ECC (with longer codeword length) for DRAM while still maintaining the same (or even shorter) data access latency than NVM. The potential of trading the raw DRAM reliability for lower bit cost has been recently demonstrated by Hynix [20], a major DRAM manufacturer, where each ECC codeword protects 64-byte user data in a DRAM DIMM. We use the term *block-protected DRAM* for such low-cost DRAM to distinguish from today's byte-accessible DRAM. Because of the complicated ECC coding, low-cost block-protected memory modules cannot meet the strict and deterministic latency specs of existing DDRx standards (e.g., DDR3 and DDR4). Fortunately, driven by the trend towards heterogeneous computer architecture [14], computing industry has been actively developing media-agnostic latency-oblivious CPU-memory interfaces (e.g., OpenCAPI [24], CXL [8], and Gen-Z [13]). This natively makes block-protected DRAM a practical viable option to reduce memory cost in future computing systems. Under this framework, memory control is primarily handled by dedicated memory controllers on each individual memory module (e.g., DIMM or PCIe card), other than by CPUs as in current practice.

Based on the above discussions, we envision a DRAM-only heterogeneous memory architecture consisting of convenient but expensive byte-accessible DRAM and low-cost block-protected DRAM. With very different native data access granularity (i.e., a few bytes vs. hundreds or thousands of bytes), these two types of DRAM have very different random data access latency/throughput, while achieving similar sequential data access latency/throughput as long as the throughput of ECC engines can match the DRAM bandwidth. As illustrated in Fig. 4, all the $m+1$ levels of the in-memory tier $L_0 \sim L_m$ are stored in the block-protected DRAM, and the OS page cache, RocksDB block cache and write buffer (e.g., memtable in RocksDB) stays in the conventional byte-accessible DRAM. With a much larger capacity than the write buffer, the levels $L_0 \sim L_m$ only involve block-oriented (e.g., 4KB or 8KB) data access, which can naturally match to the low-cost block-protected DRAM. Because block-protected DRAM can have a much lower bit cost than today's byte-accessible DRAM, this approach can largely reduce the memory cost overhead of the WAL-assisted KV store tiering. Moreover, by relying on DRAM other than NVM, this approach can perfectly leverage the existing DRAM manufacturing infrastructure, and is not subject to the cycling endurance issue and any fundamental technology risk/uncertainty. As a viable alternative to widely studied hybrid-DRAM/NVM memory architecture, our envisioned DRAM-only heterogeneous memory architecture can play a key role in building a cost-effective infrastructure for future in-memory computing applications.

4 EVALUATION

As a case study, we accordingly modified RocksDB 6.0 to support the proposed WAL-assisted tiering. We use a single thread from the RocksDB high-priority thread pool to implement all the operations in support of background KV trim (e.g., management of in-memory KV status bitmaps and SN arrays, and KV trim on WAL segments). The modified RocksDB allows users to configure the number of LSM tree levels stored in block-protected DRAM and the number of levels stored on SSD. The block-protected DRAM for in-memory tier is emulated using RAMDisk mounted with *tmpfs* on Linux. We set the capacity of RAMDisk to be slightly larger than the target size of the in-memory tier (i.e., $\sum_{i=0}^{m} C_i$). During the runtime, the in-memory tier size can dynamically vary and even temporarily grow beyond the RAMDisk capacity. To address this issue, the modified RocksDB can spill-over files from the in-memory tier to the SSD. To simplify the implementation, we still treat those spilled-over files as non-persisted (without updating KV status bitmap), and hence process those files in the same way as the other in-memory files.

4.1 Experimental Setup

We ran all the experiments on a server with dual-socket Intel Xeon E5-2630 2.2GHz CPUs (10 cores per socket), 128GB DDR4 DRAM, and a 2TB Intel DC P4600 NVMe SSD. The SSD performance specs are 3.2GB/s (1.5GB/s) sequential read (write) throughput, 610k (196k) random read (write) IOPS under whole-drive preconditioning, and 85μs (15μs) 4KB read (write) latency with IO queue depth of 1. Regarding the RocksDB configuration, the maximum number of compaction and flush threads are set to 12 and 4, respectively, both the *bytes-per-sync* and *WAL-bytes-per-sync* are set to 1MB, and all the other parameters are left as their default settings. We set the size of each file and memtable as 16MB, and the target size of L_1 as 80MB. We set the size of each key as 16B and the size of each value as 1KB. All the experiments were carried out on two datasets: (1) 100GB dataset that is populated by randomly inserting 100M KV pairs and spans over 5 levels (i.e., $L_0 \sim L_4$), and (2) 1TB dataset that is populated by randomly inserting 1,000M KV pairs and spans over 6 levels ((i.e., $L_0 \sim L_5$). In this paper, we use *RocksDB-Orig* and *RocksDB-Tier* to denote the original

RocksDB and our modified RocksDB that supports WAL-assisted tiering. *RocksDB-Tier* sets the background KV trim parameters so that the WAL size is below 2.5× of the in-memory tier size.

4.2 Micro-benchmarks

We first studied the performance (both ops/s and latency) and SSD IO traffic using the db_bench micro-benchmark tool in the RocksDB package. In particular, we ran three different workloads: (1) write-only workload that randomly inserts/updates 100M KV pairs, (2) read-only workload that randomly reads 100M KV pairs, and (3) mixed workload consisting of 233M random KV reads and 100M random KV writes, corresponding to 70%-read and 30%-write. The number of client threads is 32.

4.2.1 Results on 100GB dataset. When running experiments on the 100GB dataset (with total of 5 levels), we locked the majority of the 128GB host DRAM in order to leave 8GB (or 18GB) DRAM available for *RocksDB-Orig* and *RocksDB-Tier*. In the context of *RocksDB-Tier*, we allocate 4GB (or 14GB) DRAM for RAMDisk given the total available 8GB (or 18GB) DRAM. With 4GB RAMDisk, the in-memory tier covers the top 3 levels (i.e., $L_0 \sim L_2$); with 14GB RAMDisk, the in-memory tier covers the top 4 levels (i.e., $L_0 \sim L_3$).

A. Ops/s performance: Fig. 5 compares the ops/s performance between *RocksDB-Orig* and *RocksDB-Tier* on the 100GB dataset under the three different workloads. For each workload, all the ops/s values are normalized against the ops/s of the *RocksDB-Orig* (8GB DRAM). Compared with *RocksDB-Orig* (8GB DRAM), by moving the top 3 levels (out of total 5 levels) into memory, *RocksDB-Tier* (8GB DRAM) can improve the ops/s by 19.9% and 27.7% for the write-only and mixed workloads, respectively. Compared with *RocksDB-Orig* (18GB DRAM), by moving the top 4 levels (out of total 5 levels) into memory, *RocksDB-Tier* (18GB DRAM) can improve the ops/s by 95.5% and 48.0% for the write-only and mixed workloads, respectively. The results can be easily justified by the fact that, by leveraging the very high host DRAM bandwidth to absorb a large percentage of compaction-induced data traffic, *RocksDB-Tier* incurs much less SSD IO traffic than *RocksDB-Orig* during compaction. The results show that, under the read-only workload, *RocksDB-Orig* and *RocksDB-Tier* have similar ops/s performance, because of the very small compaction activities in the read-only workload.

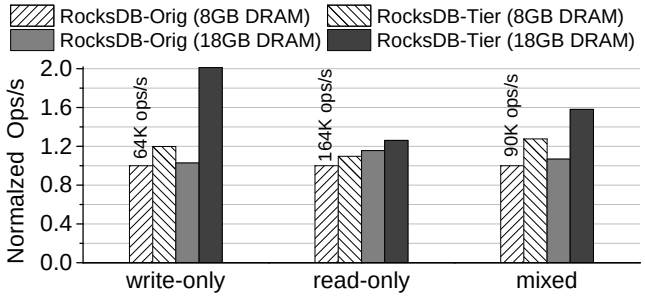

Figure 5: Normalized average ops/s performance when running different workloads on the 100GB dataset.

B. Latency performance: In addition to the ops/s performance, we also measured and compared the latency profile under different workloads. Fig. 6 shows the write latency profile under the write-only workload, where each latency value is normalized against the latency of *RocksDB-Orig* (8GB DRAM) under the same latency type. The results show that the WAL-assisted tiering can improve the ops/s performance and reduce the latency simultaneously. When operating with 8GB DRAM, compared with *RocksDB-Orig*, *RocksDB-Tier* can reduce the average write latency by 16.6% while achieving 19.9% higher ops/s performance. When operating with 18GB DRAM, compared with *RocksDB-Orig*, *RocksDB-Tier* can reduce the average write latency by 48.9% while achieving 95.5% higher ops/s performance. In addition to average write latency, WAL-assisted tiering also largely reduces the tail latency. When operating with 8GB DRAM, compared with *RocksDB-Orig*, *RocksDB-Tier* can reduce the 99-percentile and 99.9-percentile latency by 12.4% and 36.4%, respectively. When operating with 18GB DRAM, compared with *RocksDB-Orig*, *RocksDB-Tier* can reduce the 99-percentile and 99.9-percentile latency by 51.3% and 77.5%, respectively. The results suggest that reducing IO traffic on SSD can very effectively reduce both the average latency and tail latency.

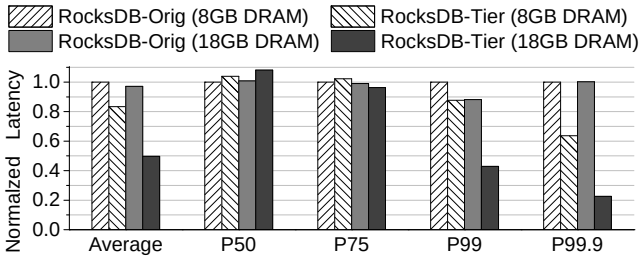

Figure 6: Normalized latency profile when running the write-only workload on the 100GB dataset.

Fig. 7(a) and (b) show the normalized read and write latency profile, respectively, when running the mixed workload on the 100GB dataset. When operating with 8GB and 18GB DRAM, compared with *RocksDB-Orig*, *RocksDB-Tier* can reduce the average read latency by 23.3% and 35.7%, respectively, and can reduce the average write latency by 23% and 9.7%, respectively. We note that *RocksDB-Tier* accomplishes the latency reduction while achieving 27.7% (8GB DRAM) and 48.0% (18GB DRAM) higher ops/s performance at the same time. As shown in Fig. 7(b), compared with *RocksDB-Orig*, *RocksDB-Tier* has noticeably longer 50-percentile, 75-percentile, and 99-percentile tail write latency under the mixed workload. We conjecture that it is likely caused by the higher ops/s (hence heavier CPU usage) of *RocksDB-Tier*.

C. SSD IO traffic volume: As discussed above, the performance benefit of WAL-assisted tiering is enabled by the SSD IO traffic volume reduction. Fig. 8 (a) and (b) further show the normalized SSD read and write traffic volume when running the three different workloads. The results clearly show the significant SSD IO traffic volume reduction enabled by WAL-assisted tiering. When operating with 8GB DRAM, compared with *RocksDB-Orig*, *RocksDB-Tier* can reduce

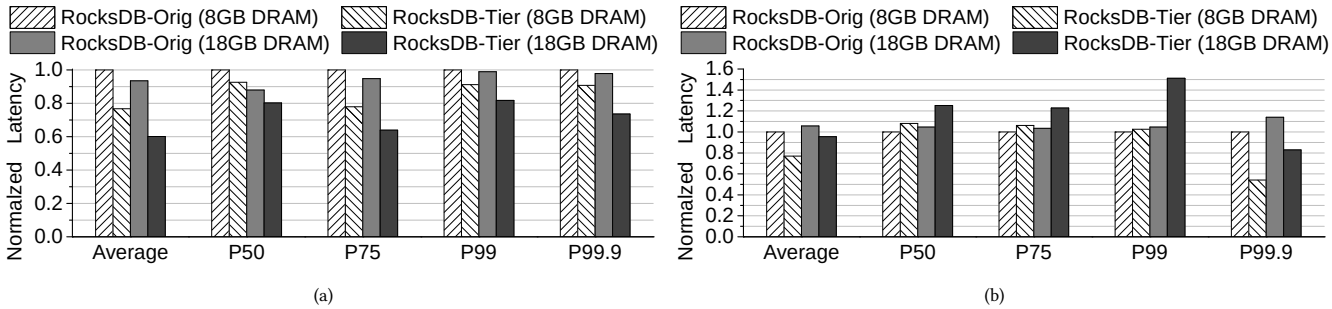

Figure 7: Normalized (a) read latency and (b) write latency profile when running the mixed workload on the 100GB dataset.

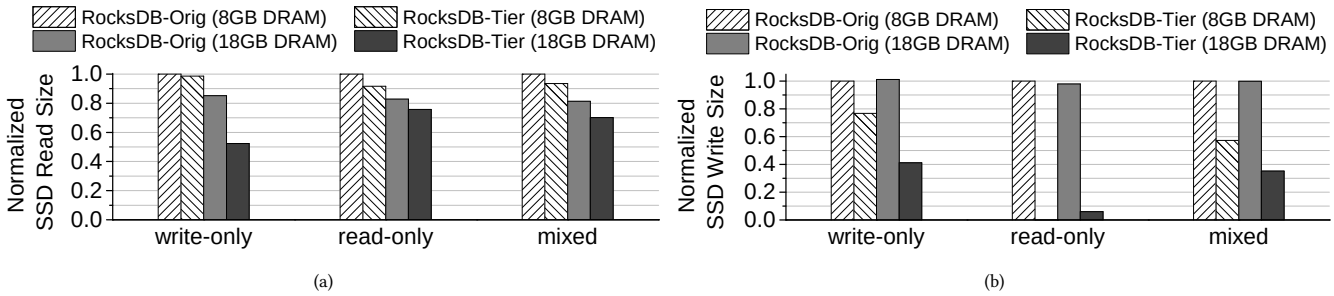

Figure 8: Normalized SSD (a) read volume and (b) write volume when running three different workloads on the 100GB dataset.

the SSD read traffic volume by 1.4% and 6.4% under the write-only and mixed workloads, respectively. The SSD read traffic volume reduction can improve to 38.5% and 13.7% when operating with 18GB DRAM (i.e., the in-memory tier contains the top 4 out of total 5 levels). When operating with 8GB DRAM, compared with *RocksDB-Orig*, *RocksDB-Tier* can reduce the SSD write traffic volume by 23.2% and 42.7% for write-only and mixed workloads, respectively. The SSD write traffic volume reduction can improve to 59.4% and 64.8% when operating with 18GB DRAM.

D. On-SSD WAL size: As pointed out above, we applied the background KV trim to keep the on-SSD WAL size below 2.5× of the in-memory tier size. For the purpose of comparison, we carried out experiments on *RocksDB-Tier* with the background KV trim disabled. Table. 1 compares the peak on-SSD WAL size when running the write-only workload under different DRAM capacity. The results clearly show the effectiveness of the proposed background KV trim on reducing the WAL size. When operating with 18GB DRAM (14GB in-memory tier), the peak on-SSD WAL size can be 102GB if we turn off the background KV trim, which is even larger than the KV store dataset size. Using our proposed background KV trim, we can reduce the peak on-SSD WAL size by 5× from 102GB to only 19.6GB.

4.2.2 Results on 1TB dataset. We carried out experiments on the 1TB dataset (with total 6 levels) as well, where we made 18GB (or 120GB) host DRAM available for both *RocksDB-Orig* and *RocksDB-Tier*. In the context of *RocksDB-Tier*, we allocate 14GB (or 100GB) for

Table 1: Peak on-SSD WAL size of 100GB dataset

DRAM Capacity	8GB		18GB	
In-memory Tier	4GB		14GB	
KV trim	ON	OFF	ON	OFF
Peak WAL size	9.8GB	29.6GB	19.6GB	102GB

RAMDisk given the total available 18GB (or 120GB) DRAM. With 14GB RAMDisk, the in-memory tier covers the top 4 levels (i.e., $L_0 \sim L_3$); with 100GB RAMDisk, the in-memory tier covers the top 5 levels (i.e., $L_0 \sim L_4$).

A. Ops/s and latency performance: Fig. 9 shows the normalized ops/s performance when running the three workloads on the 1TB dataset.

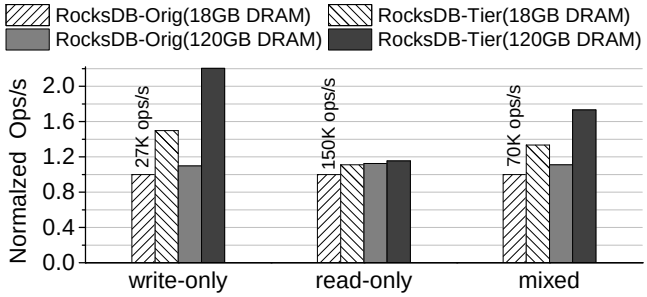

Figure 9: Normalized average ops/s performance when running different workloads on the 1TB dataset.

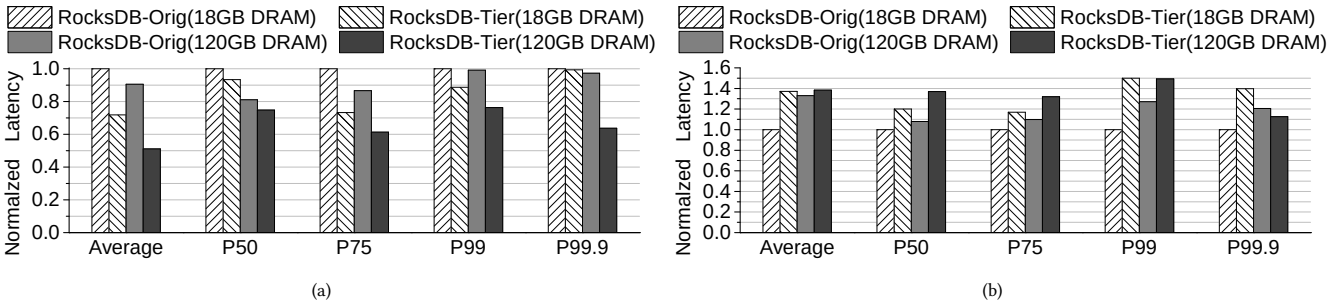

Figure 11: Normalized (a) read latency and (b) write latency profile when running the mixed workload on the 1TB dataset.

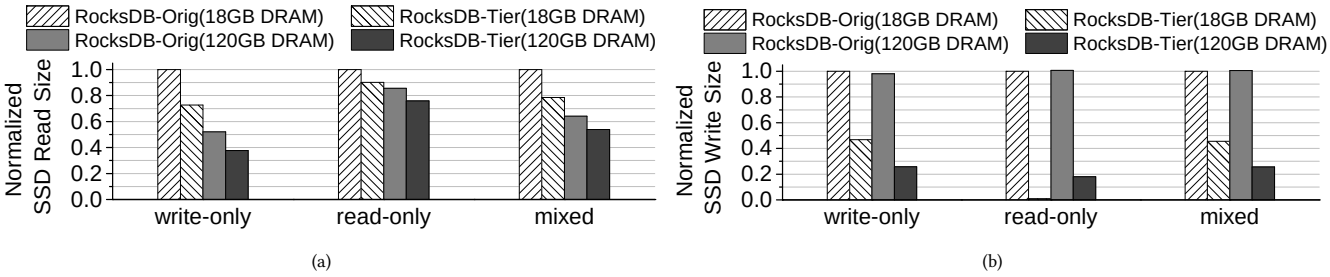

Figure 12: Normalized SSD (a) read volume and (b) write volume when running three different workloads on the 1TB dataset.

Similar to the results on 100GB dataset, *RocksDB-Tier* can consistently achieve significant ops/s performance gain over *RocksDB-Orig*. For example, when operating with 18GB DRAM, *RocksDB-Tier* can improve the ops/s performance by 49.8% and 33.4% for write-only and mixed workloads, respectively.

Fig. 10 shows the normalized write latency profile when running the write-only workload on the 1TB dataset. The results again reveal the effectiveness of the WAL-assisted tiering on reducing the latency while maintaining a higher ops/s performance. When operating with 18GB DRAM, *RocksDB-Tier* can reduce the average write latency by 33.2%, and reduce the 99-percentile and 99.9-percentile tail latency by 75.1% and 20%, respectively.

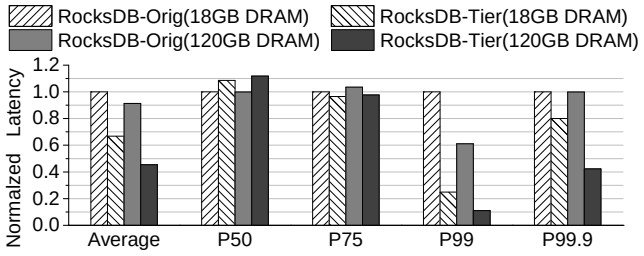

Figure 10: Normalized write latency profile when running the write-only workload on the 1TB dataset.

Fig. 11(a) and (b) show the normalized read and write latency profile when running the mixed workload on the 1TB dataset. The results show that WAL-assisted tiering can noticeably reduce the read latency while maintaining higher ops/s performance. For example, when operating with 120GB DRAM, *RocksDB-Tier* can reduce the average read latency by 43.6%, and reduce the 99-percentile and 99.9-percentile read tail latency by 23% and 34.5%, respectively. Nevertheless, as shown in Fig. 11(b), the write latency of *RocksDB-Tier* becomes worse than that of *RocksDB-Orig*. For example, when operating with 120GB DRAM, the average and 99-percentile write latency of *RocksDB-Tier* is 4.1% and 17.5% longer than that of *RocksDB-Orig*. Again, we conjecture that it is caused by the higher ops/s (hence heavier CPU usage) of *RocksDB-Tier*.

B. SSD IO traffic volume: Fig. 12(a) and (b) show the normalized SSD read and write traffic volume when running the three different workloads on the 1TB dataset. Again, the results clearly show that the proposed design approach can significantly reduce the SSD IO traffic volume, leading to the ops/s and latency performance benefits. For example, when operating with 18GB DRAM, compared with *RocksDB-Orig*, *RocksDB-Tier* can reduce the SSD write traffic volume of the write-only and mixed workloads by 53.2% and 54.4%, respectively, which can directly translate into about 2× longer SSD lifetime.

C. On-SSD WAL size: For the purpose of comparison, we also carried out experiments on *RocksDB-Tier* with the background KV trim disabled. Table. 2 compares the peak on-SSD WAL size under different DRAM capacity with KV trim ON or OFF. Without using the proposed background KV trim, the peak on-SSD WAL size can be very close to the size of KV store dataset itself. By turning on the background KV trim, we can reduce the peak on-SSD WAL size by about 4×.

Table 2: Peak on-SSD WAL size of 1TB dataset

DRAM Capacity	18GB		120GB	
In-memory Tier	14GB		100GB	
KV trim	ON	OFF	ON	OFF
Peak WAL size	22GB	86GB	213GB	950GB

4.3 YCSB benchmarks

To evaluate over a wider range of workload characteristics, we further ran the industry standard YCSB macro-benchmarks [5, 9] on *RocksDB-Tier* and *RocksDB-Orig*. Table. 3 lists the six core YCSB macro-benchmark workloads. All the experiments are carried out on a 100GB dataset that is populated by randomly inserting 100M unique KV pairs. Same as the above db_bench experiments, we set the size of each key and value as 16 bytes and 1KB bytes, and use 32 client threads. Out of the total 128GB host DRAM, we lock 110GB and leave the remaining 18GB available for *RocksDB-Tier* and *RocksDB-Orig*. In the context of *RocksDB-Tier*, we allocate 14GB DRAM for RAMDisk that hosts the top 4 levels out of the total 5 levels. Each YCSB workload runs with 100M random keys. In addition to the uniform key distribution, we also used the zipf distribution with three different zipf constants: 0.5, 0.75 and 0.99 (default).

Table 3: YCSB core workloads

Workload	Description
YCSB A	50% reads, 50% updates
YCSB B	95% reads, 5% updates
YCSB C	100% reads
YCSB D	95% reads, 5% inserts
YCSB E	95% scans, 5% inserts
YCSB F	50% reads, 50% read-modify-writes

Fig. 13 shows ops/s performance improvement of *RocksDB-Tier* over *RocksDB-Orig*, under different YCSB workloads and different key distribution models. The results show that the gain of *RocksDB-Tier* improves as the workload write intensity increases. The workloads YCSB A (50% updates) and YCSB F (50% read-modify-writes) can much more benefit from *RocksDB-Tier* than the other workloads. With uniform key distribution, *RocksDB-Tier* improves the ops/s by

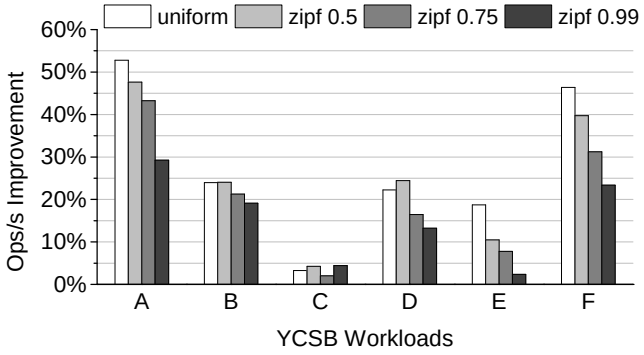

Figure 13: The ops/s performance improvement of *RocksDB-Tier* over *RocksDB-Orig*, under different YCSB workloads and different key distribution models.

52.8% and 46.4% for YCSB A and YCSB F, respectively. In comparison, the ops/s gain on the other workloads is below 25%, and YCSB C (with 100% read) hardly benefits from *RocksDB-Tier*. This can be explained as follows: The objective of WAL-assisted tiering is to benefit KV store performance by reducing the compaction-induced SSD write traffic. Larger workload write intensity will trigger heavier compaction and hence higher compaction-induced SSD write traffic, for which WAL-assisted tiering can be more beneficial. Moreover, because *read-modify-writes* trigger more reads than *updates*, the gain of *RocksDB-Tier* is slightly higher on YCSB A than YCSB F. Fig. 13 shows that the gain of WAL-assisted tiering is also dependent on the workload key distribution. *RocksDB-Tier* brings the highest ops/s performance gain under uniform key distribution, and its performance gain decreases as the key distribution becomes more skewed (note that zipf distribution is more skewed as its constant increases). This can be explained as follows: When the key distribution becomes more skewed, there is a higher probability that a new KV pair displaces its older version in a top level (e.g., level L_2 or L_3). This will help to slowdown the growth of the size of those top levels, leading to less compaction activities and hence less write traffic. Accordingly, the benefit of WAL-assisted tiering becomes less significant.

To further study the impact of workload stress, we measured the ops/s performance of YCSB A and YCSB F under a variety of RocksDB client number, as shown in Fig. 14. All the workloads have uniform key distribution. As we increase the client number from 32 to 96 (recall that the server has total 20 cores), the total ops/s value always improves. Meanwhile, the gain of *RocksDB-Tier* over *RocksDB-Orig* enlarges as the client number increases. For the workload YCSB A, compared with *RocksDB-Orig*, *RocksDB-Tier* improves the ops/s by 44%, 52.8%, 63.6 and 61% under 16, 32, 64 and 96 clients, respectively. For the workload YCSB F, compared with *RocksDB-Orig*, *RocksDB-Tier* improves the ops/s by 41.9%, 46.4%, 50.9%, and 55.3% under 16, 32, 64 and 96 clients, respectively. The results suggest that the proposed WAL-assisted tiering could improve the KV store performance scalability.

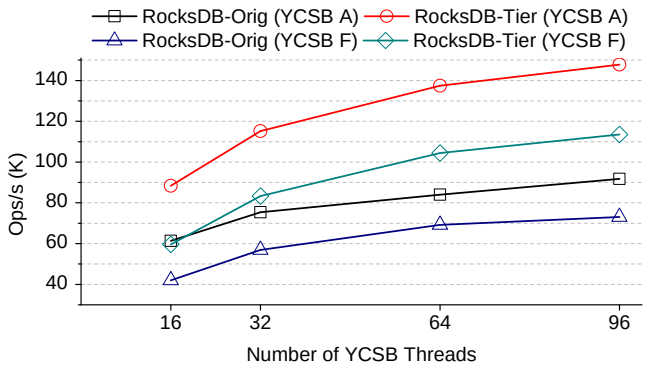

Figure 14: The ops/s performance of YCSB A and YCSB F under different number of YCSB clients.

Finally, Fig. 15 shows SSD write data volume reduction of *RocksDB-Tier* over *RocksDB-Orig*, under all the YCSB workloads (except the read-only workload YCSB C). Although the absolute write data volume strongly depends on the workload characteristics (e.g., write

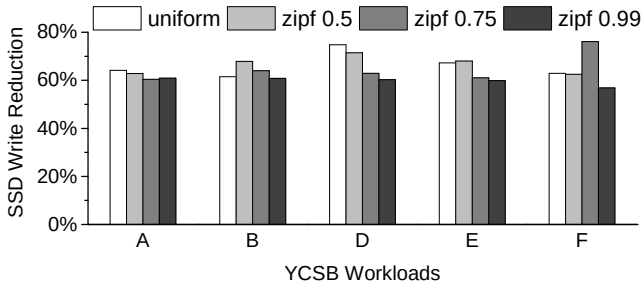

Figure 15: SSD write data volume reduction of *RocksDB-Tier* over *RocksDB-Orig*.

intensity and key distribution), the results show that the *RocksDB-Tier* over *RocksDB-Orig* write data volume reduction remains consistently around 60%. This further affirms the effectiveness of using WAL-assisted tiering to lengthen the SSD lifetime.

4.4 Impact of Longer DRAM Access Latency

The above experiment results well demonstrate the effectiveness of the proposed WAL-assisted tiering on today's byte-addressable DRAM. Since the files in in-memory tier are stored in RAMDisk by default and accessed in the granularity of block through tmpfs file system API, all the above experiment results also readily apply to the envisioned block-protected DRAM, assuming that the block-protected DRAM has the same 4KB page access latency as today's DDR DRAM DIMM. Since the ECC decoding hardware engine in the block-protected DRAM controller can match the throughput of the media-agnostic latency-oblivious interfaces (e.g., CXL), it is reasonable to expect that block-protected DRAM and today's DDR DRAM DIMM may achieve similar sequential data access (e.g., 4KB) latency. Because the in-memory tier is accessed in the unit of blocks and each block is at least a few KBs, in the above experiments, we directly use the DDR DRAM DIMMs in our server to emulate the envisioned block-protected DRAM. Of course, it is still necessary to study the effect if block-protected DRAM has substantially longer sequential data access latency than DDR DRAM DIMM. For this purpose, we introduce a sequential data access slowdown factor $\beta \geq 1$. In particular, we modified the RAMDisk source code so that the latency of each 4KB data access is amplified by β times.

Accordingly, we carried out further experiments to study the impact of the in-memory tier data access slowdown factor β. We ran the experiments on the 100GB dataset with 18GB available host DRAM, and used the same configurations and workloads as in Section 4.2.1. *RocksDB-Orig* runs with 18GB byte-addressable DRAM. *RocksDB-Orig* runs with 4GB byte-addressable DRAM and 14GB emulated block-protected DRAM. Fig. 16 shows the measured ops/s performance of *RocksDB-Tier* under three values of β (i.e., 1, 2, and 5).The slowdown factor is only applied to the 14GB emulated block-protected DRAM. For the purpose of comparison, it also shows the ops/s performance of the original RocksDB (i.e., *RocksDB-Orig*). The results show that the in-memory data access latency indeed could have a noticeable impact on the performance of the write-only workload. Nevertheless, even with the slowdown factor of 5, *RocksDB-Tier* can still achieve 51% higher ops/s than *RocksDB-Orig*. The results show that the performance of read-only and mixed workloads is almost independent from the in-memory tier data access latency slowdown factor. This is because the read quests are mostly served by SSD access, which has nothing to do with the in-memory tier data access latency. The above results further strengthen the potential of applying the envisioned block-protected DRAM to reduce the cost overhead of the WAL-assisted tiering design approach.

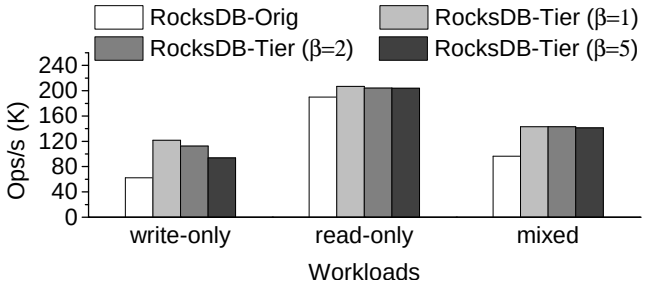

Figure 16: Measured ops/s performance on the 100GB dataset under different slowdown factor β.

4.5 Crash Recovery Time

WAL-assisted tiering is subject to a longer crash recovery time, simply because a much larger amount of in-memory data must be reconstructed from the WAL. Moreover, as discussed above, in addition to the proposed background KV trim method, one could apply background lossless data compression to further reduce the WAL storage footprint on SSD. For instance, by setting the KV pair data compressibility as 0.5 in RocksDB db_bench, our measurement shows that the WAL file compression ratio is about 0.550, 0.438, and 0.443 when using the LZ4, ZSTD, and Zlib compression libraries, respectively.

During the crash recovery, WAL segments are first decompressed and then replayed to reconstruct the in-memory tier. To reduce the recovery time overhead caused by WAL segment decompression, we can use multiple threads to decompress multiple WAL segments concurrently, and then replay these WAL segments one-by-one sequentially. In this study, we use four threads for concurrent WAL segment decompression. Using the 100GB dataset as a test vehicle, Table. 4 compares the crash recovery time under different configurations. Because LZ4 decompression can achieve very high throughput (e.g., above 1GB/s per CPU core), applying LZ4 to compress WAL segments does not cause noticeable crash recovery time overhead. The results show that, compared with no WAL compression, LZ4-based WAL compression can even slightly reduce the crash recovery time. This is because of the reduction on the SSD read traffic during crash recovery. Because ZSTD and Zlib have (much) lower decompression throughput than LZ4, they noticeably increase the crash recovery time. The results show that, compared with no compression, ZSTD and Zlib compression increase the crash recovery time by about 6.4% and 33.4%, respectively.

For the original RocksDB, since we only need to reconstruct the small in-memory memtable (e.g., a few tens of MBs), the crash

Table 4: Crash Recovery Time of 100GB dataset

DRAM Capacity	8GB	18GB
WAL Size	4GB	13GB
No compression	10.9 Seconds	36.2 Seconds
LZ4 Compression	10.9 Seconds	35.0 Seconds
ZSTD compression	11.6 Seconds	38.3 Seconds
Zlib Compression	13.7 Seconds	48.3 Seconds

recovery time is much shorter (e.g., less than 1s). Although the use of WAL-assisted tiering increases the crash recovery time by 1 or 2 orders of magnitude, the absolute crash recovery time is still not prohibitively long (i.e., tens or hundreds of seconds). Therefore, for typical use cases, we expect that the crash recovery time overhead of WAL-assisted tiering can be justified by its performance and SSD lifetime benefits.

5 RELATED WORK

Being very intensively studied over the recent years, log-structured KV store is fundamentally subject to trade-offs among write amplification, read amplification, and space amplification [2, 10]. Aiming to reduce the space amplification (hence reduce the data storage cost) at the penalty of higher write amplification, the *leveled* compaction strategy tends to have worse KV store performance. As a result, prior work mainly focused on improving the performance of log-structured KV store with leveled compaction.

Accordion [6] shares the same basic principle as this work: Keeping a larger amount of data in memory can help to reduce the write amplification experienced by on-disk data. In particular, Accordion [6] partitions the memory component into an active mutable segment and a pipeline of flat immutable segments. By keeping all the in-memory data flat in one level, it avoids the WAL size problem, but meanwhile cannot seamlessly scale up the in-memory data volume. In comparison, WAL-assisted tiering can naturally utilize tens or hundreds GBs of host memory.

Most prior work aimed at reducing the write amplification by more fundamentally modifying the KV store architecture. PebblesDB [28] reduces the write amplification by developing a fragmented LSM tree structure. Dostoevsky [11] reduces the write amplification by developing a lazy leveling scheme, and also presents a generalization of the entire LSM tree design space. TRIAD [4] reduces the write amplification by separating hot and cold keys, deferring the compaction and using the commit log as L_0 files. SkipTree [36] reduces the write amplification by allowing certain KV pairs to skip the level-by-level compaction. VT-tree [31] reduces the write amplification by using stitching operation to avoid unnecessary data copies for sequential data. LSM-trie [32] reduces the write amplification by integrating the exponential growth pattern in the LSM tree with a linear growth pattern and using tries to organize the data. SlimDB [29] reduces the write amplification by modifying the KV store structure specifically for semi-sorted data. X-Engine [15] reduces the write amplification by identifying and recycling the data blocks whose key ranges do not overlap with any other data blocks during the compaction. Write amplification can also be reduced by separating the storage of key and value, where all the values are managed outside of LSM tree with much lower write amplification. Prior work [7, 19, 22, 26, 27, 34] have well demonstrated its effectiveness, especially when the value size is relatively large.

Although directly reducing the write amplification can most fundamentally mitigate the performance impact, it is difficult for existing mature KV stores to adopt these design solutions due to the significant changes on the core data structure and algorithm. Moreover, due to the inherent trade-offs of log-structured KV store, these design solutions typically come with penalty in terms of read amplification and/or space amplification. In contrast to these prior work, the proposed WAL-assisted DRAM/SSD tiering approach in this paper does not aim at reducing the write amplification at all and hence obviates any changes on the core data structures and algorithms of existing KV stores.

Prior work also studied the design of log-structured KV store for new storage technologies, including NVM [12, 17, 18, 21, 33] and shingled magnetic recording (SMR) drive [34, 35]. The underlying theme is to customize the data structure and operations of log-structured KV store in order to better embrace the unique characteristics of these storage technologies. In contrast to these prior work, this paper presents a solution that can significantly improve the performance of existing log-structured KV stores using today's mature DRAM and SSD technologies. We further advocate an alternative DRAM-only strategy to reduce the memory cost by using block-protected DRAM.

6 CONCLUSIONS

This paper presents a WAL-assisted DRAM/SSD tiering approach to improve the performance of log-structured KV stores. In contrast to most prior efforts, this approach does not aim at reducing the write amplification at all and hence obviates any changes on the core data structures and algorithms of existing KV stores. In spite of the simple design concept, its practical implementation is subject to the issues of WAL size and host memory usage. We present a background KV trim method to control and reduce the on-SSD WAL size, and propose a DRAM-only low-cost heterogeneous memory architecture to mitigate the memory usage overhead. To demonstrate its practical feasibility, we integrated the developed techniques into RocksDB by only adding about 1,200 lines of code and without touching its core data structures and algorithms. Its effectiveness has been well demonstrated through extensive experiments over a variety of workloads.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their constructive comments. This work was supported in part by the U.S. National Science Foundation under Grants CCF-1629218, CCF-1629201, and CNS-1814890.

REFERENCES

[1] Apache Cassandra. http://cassandra.apache.org/.
[2] Athanassoulis, M., Kester, M. S., Maas, L. M., Stoica, R., Idreos, S., Ailamaki, A., and Callaghan, M. Designing Access Methods: The RUM Conjecture. In *Proceedings of the International Conference on Extending Database Technology (EDBT)* (2016), pp. 461–466.
[3] Bacon, D. F., Bales, N., Bruno, N., Cooper, B. F., Dickinson, A., Fikes, A., Fraser, C., Gubarev, A., Joshi, M., Kogan, E., Lloyd, A., Melnik, S., Rao, R., Shue, D., Taylor, C., van der Holst, M., and Woodford, D. Spanner: Becoming a SQL System. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2017), pp. 331–343.

[4] Balmau, O., Didona, D., Guerraoui, R., Zwaenepoel, W., Yuan, H., Arora, A., Gupta, K., and Konka, P. TRIAD: Creating synergies between memory, disk and log in log structured key-value stores. In *Proceedings of USENIX Annual Technical Conference (ATC)* (2017), pp. 363–375.

[5] Balmau, O., Dinu, F., Zwaenepoel, W., Gupta, K., Chandhiramoorthi, R., and Didona, D. SILK: Preventing latency spikes in log-structured merge key-value stores. In *Proceedings of USENIX Annual Technical Conference (ATC)* (2019), pp. 753–766.

[6] Bortnikov, E., Braginsky, A., Hillel, E., Keidar, I., and Sheffi, G. Accordion: Better memory organization for LSM key-value stores. *Proceedings of the VLDB Endowment 11*, 12 (2018), 1863–1875.

[7] Chan, H. H., Liang, C.-J. M., Li, Y., He, W., Lee, P. P., Zhu, L., Dong, Y., Xu, Y., Xu, Y., Jiang, J., et al. HashKV: Enabling efficient updates in KV storage via hashing. In *Proceedings of USENIX Annual Technical Conference (ATC)* (2018), pp. 1007–1019.

[8] Compute Express Link (CXL). https://www.computeexpresslink.org.

[9] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking cloud serving systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)* (2010), ACM, pp. 143–154.

[10] Dayan, N., Athanassoulis, M., and Idreos, S. Monkey: Optimal navigable key-value store. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2017), ACM, pp. 79–94.

[11] Dayan, N., and Idreos, S. Dostoevsky: Better space-time trade-offs for LSM-tree based key-value stores via adaptive removal of superfluous merging. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2018), ACM, pp. 505–520.

[12] Eisenman, A., Gardner, D., AbdelRahman, I., Axboe, J., Dong, S., Hazelwood, K., Petersen, C., Cidon, A., and Katti, S. Reducing DRAM footprint with NVM in Facebook. In *Proceedings of the Thirteenth European Conference on Computer Systems (EuroSys)* (2018), ACM, p. 42.

[13] Gen-Z. . https://genzconsortium.org.

[14] Hennessy, J. L., and Patterson, D. A. A new golden age for computer architecture. *Commun. ACM 62*, 2 (2019), 48–60.

[15] Huang, G., Cheng, X., Wang, J., Wang, Y., He, D., Zhang, T., Li, F., Wang, S., Cao, W., and Li, Q. X-Engine: An optimized storage engine for large-scale E-commerce transaction processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2019), ACM, pp. 651–665.

[16] Intel Optane DC Persistent Memory Architecture Overview. https://techfieldday.com/video/intel-optane-dc-persistent-memory-architecture-overview/.

[17] Kaiyrakhmet, O., Lee, S., Nam, B., Noh, S. H., and Choi, Y.-r. SLM-DB: single-level key-value store with persistent memory. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)* (2019), pp. 191–205.

[18] Kannan, S., Bhat, N., Gavrilovska, A., Arpaci-Dusseau, A., and Arpaci-Dusseau, R. Redesigning LSMs for nonvolatile memory with NoveLSM. In *Proceedings of USENIX Annual Technical Conference (ATC)* (2018), pp. 993–1005.

[19] Lai, C., Jiang, S., Yang, L., Lin, S., Sun, G., Hou, Z., Cui, C., and Cong, J. Atlas: Baidu's key-value storage system for cloud data. In *Proceedings of the Symposium on Mass Storage Systems and Technologies (MSST)* (2015), IEEE, pp. 1–14.

[20] Lee, S., Jeon, B., Kang, K., Ka, D., Kim, N., Kim, Y., Hong, Y., Kang, M., Min, J., Lee, M., Jeong, C., Kim, K., Lee, D., Shin, J., Han, Y., Shim, Y., Kim, Y., Kim, Y., Kim, H., Yun, J., Kim, B., Han, S., Lee, C., Song, J., Song, H., Park, I., Kim, Y., Chun, J., and Oh, J. 512GB 1.1V Managed DRAM Solution with 16GB ODP and Media Controller. In *IEEE International Solid- State Circuits Conference (ISSCC)* (Feb 2019), pp. 384–386.

[21] Lersch, L., Oukid, I., Lehner, W., and Schreter, I. An analysis of LSM caching in NVRAM. In *Proceedings of the 13th International Workshop on Data Management on New Hardware* (2017), pp. 1–5.

[22] Lu, L., Pillai, T. S., Gopalakrishnan, H., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. WiscKey: Separating keys from values in SSD-conscious storage. *ACM Transactions on Storage (TOS) 13*, 1 (2017), 5.

[23] Luo, C., and Carey, M. J. LSM-based Storage Techniques: A Survey. *arXiv preprint arXiv:1812.07527* (2018).

[24] Open Coherent Accelerator Processor Interface (OpenCAPI). https://opencapi.org.

[25] O'Neil, P., Cheng, E., Gawlick, D., and O'Neil, E. The log-structured merge-tree (LSM-tree). *Acta Informatica 33*, 4 (1996), 351–385.

[26] Papagiannis, A., Saloustros, G., González-Férez, P., and Bilas, A. Tucana: Design and implementation of a fast and efficient scale-up key-value store. In *Proceedings of USENIX Annual Technical Conference (ATC)* (2016), pp. 537–550.

[27] Papagiannis, A., Saloustros, G., González-Férez, P., and Bilas, A. An efficient memory-mapped key-value store for flash storage. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)* (2018), ACM, pp. 490–502.

[28] Raju, P., Kadekodi, R., Chidambaram, V., and Abraham, I. PebblesDB: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)* (2017), pp. 497–514.

[29] Ren, K., Zheng, Q., Arulraj, J., and Gibson, G. SlimDB: A space-efficient key-value storage engine for semi-sorted data. *Proceedings of the VLDB Endowment 10*, 13 (2017), 2037–2048.

[30] RocksDB. https://github.com/facebook/rocksdb.

[31] Shetty, P. J., Spillane, R. P., Malpani, R. R., Andrews, B., Seyster, J., and Zadok, E. Building workload-independent storage with VT-trees. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)* (2013), pp. 17–30.

[32] Wu, X., Xu, Y., Shao, Z., and Jiang, S. LSM-trie: An LSM-tree-based ultra-large key-value store for small data items. In *Proceedings of USENIX Annual Technical Conference (ATC)* (2015), pp. 71–82.

[33] Xia, F., Jiang, D., Xiong, J., and Sun, N. HiKV: a hybrid index key-value store for DRAM-NVM memory systems. In *Proceedings of USENIX Annual Technical Conference (ATC)* (2017), pp. 349–362.

[34] Yao, T., Wan, J., Huang, P., He, X., Gui, Q., Wu, F., and Xie, C. A light-weight compaction tree to reduce I/O amplification toward efficient key-value stores. In *Proceedings of the International Conference on Massive Storage Systems and Technology (MSST)* (2017).

[35] Yao, T., Wan, J., Huang, P., Zhang, Y., Liu, Z., Xie, C., and He, X. GearDB: a GC-free key-value store on HM-SMR drives with gear compaction. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)* (2019), pp. 159–171.

[36] Yue, Y., He, B., Li, Y., and Wang, W. Building an efficient put-intensive key-value store with skip-tree. *IEEE Transactions on Parallel and Distributed Systems 28*, 4 (2016), 961–973.