Redesigning High-Performance LSM-based Key-Value Stores with Persistent CPU Caches

Yijie Zhong*, Zhirong Shen*✉, Zixiang Yu*, Jiwu Shu*†,
*Xiamen University, †Tsinghua University
yijiezhong@stu.xmu.edu.cn, shenzr@xmu.edu.cn, yuzixiang23@foxmail.com, jwshu@xmu.edu.cn

***Abstract*—By providing non-volatility with DRAM-comparable performance, the emerging persistent memory (PMem) is propelling new key-value (KV) store designs. The recently released Intel Optane DC PMem now shifts the persistent boundary from memory up to CPU caches, which further eliminates the needs of cacheline flush instructions used in extensive KV stores. However, we uncover via testbed experiments that this change can even degrade the performance of existing KV stores once directly deploying them atop the new generation of the Optane PMem, stemming mainly from the mismatch of access granularities and heavy software designs.**

In this paper, we present CacheKV, the first KV store built atop persistent CPU caches. CacheKV allocates per-core sub-MemTable in CPU caches with a lazy index update mechanism, so as to fast absorb incoming writes. It then proposes a copy-based flush mechanism to convert small-sized cacheline evictions into large-sized flushes to suppress the write amplification. CacheKV finally accelerates read operations via periodically compacting the sub-skiplists. Extensive testbed experiments show that CacheKV improves the write throughput by 19.5× on average in the write-dominated environment without compromising the read performance, when compared to the state-of-the-art KV stores for the PMem.

I. Introduction

KV stores have been emerging as a backbone of modern storage infrastructures to support a wide spectrum of data-intensive applications, ranging from web indexing [1], [2], social networking [3]–[5], graph databases [6]–[8], stream processing [9], [10], down to data caching [11]–[13]. Extensive studies [2], [3], [7], [14]–[20] popularly adopt *log-structured merge tree* (LSM-tree) [21] to build KV stores on top of block storage devices (e.g., HDD [2], [3], [18], [22] and SSD [14], [23]). At a high level, the LSM-tree comprises several salient designs: (i) it transforms random small writes into large sequential writes, so as to fully utilize the storage bandwidth; and (ii) it organizes the KV pairs resided on storage devices into multiple levels with gradually increased capacity, so as to balance the I/O cost of updates and that of lookups.

While most empirical KV stores are designed based on block-level storage, the emerging PMem [1] (e.g., phase-change memory (PCM) [24]–[27], Resistive Memory (ReRAM) [28], [29], and 3D-XPoint [30]) has exhibited great potential to revolutionize the designs for LSM-tree-based KV stores (Figure 1), since it offers DRAM-comparable access performance and persistent storage with a much larger capacity. The first commercial product of the PMem, *Intel Optane DC persistent memory module* (called "Optane PMem" for brevity), has been available on the marketplace [31], making the co-designs of KV stores with the PMem promising and practical. A large number of studies [15], [32]–[39] have explored the utilization of the PMem in KV stores, most of which fall in the following branches: (i) replacing DRAM with the PMem to timely persist incoming KV pairs without data logging (e.g., SLM-DB [33]); (ii) designing KV stores for hybrid DRAM-PMem memory systems to provide fast, reliable, and durable data storage (e.g., HiKV [34], Bullet [35], Viper [36] and NoveLSM [32]); (iii) laying the PMem between DRAM and SSDs for throttling the storage I/O in compactions (e.g., MatrixKV [15]); and (iv) aggregating small writes to saturate the write-combining buffer embedded in the Optane PMem for accelerating the write performance (e.g., FlatStore [38] and ChameleonDB [37]).

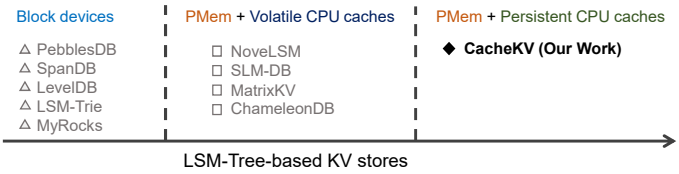


Figure 1: How CacheKV compares to representative KV stores.

Our observation is that there still remains a huge gap between *the assumptions made for the PMem* and *the real characteristics exhibited in the Optane PMem*. Such a gap, if not properly considered in KV store designs, will limit or even counteract the performance gains made in previous studies [32]–[35]. First, previous studies [32], [34], [35], [40], [41] commonly leverage the PMem as a fully byte-addressable memory device, while the Optane PMem actually has a write unit of 256 B by default (the size of the access granularity of the storage media in the Optane PMem), making it more like a block storage device instead. The write unit creates performance variations between small writes (smaller than 256 B) and large writes (256 B or larger); for example, the 64-B writes (aligned with the 256 B unit sizes) only achieve one-fourth of the write throughput of the 256-B writes [37]. Second, existing KV stores [32], [33], [37], [38] usually employ the PMem

✉Corresponding author: Zhirong Shen (shenzr@xmu.edu.cn). This work was supported by the National Key R&D Program of China (2021YFF0704001), Natural Science Foundation of China (62072381 and 61832011), Science and Technology Projects of Innovation Laboratory for Sciences and Technologies of Energy Materials of Fujian Province (IKKEM) HRTP-[2022]-1, and CCF-Huawei Innovation Research Plan (CCF2021-admin-270-202102).

[1]We use PMem to denote the persistent memory technology (e.g., PCM) and employ Optane PMem to refer to the Optane DC PMem product.

based on the premise that it provides persistent storage and leaves CPU caches volatile, while the newest Optane PMem has already shifted the persistent boundary from the memory up to CPU caches [42]. Apparently, this change eliminates the usage of expensive data flush instructions (e.g., `clflush` and `clwb`) and favors the overall access performance, yet we uncover that the advantages of the persistent CPU caches cannot be fully leveraged by existing KV store designs, stemming mainly from the mismatch of the access granularities and the heavy software designs (see Section II-C for more details). Hence, *how to design a high-performance KV store that fully leverages both characteristics of the Optane PMem is crucial yet unfortunately largely overlooked.*

We fill in the blank by presenting CacheKV, which incorporates the two significant characteristics of the Optane PMem into the KV store designs. To leverage the persistent nature of CPU caches, CacheKV divides the single MemTable in conventional LSM-tree to multiple small-sized sub-MemTables and manages the sub-MemTable pool in CPU caches, which dynamically assigns a sub-MemTable to each CPU core for serving concurrent writes. It defers the index updates and flushes data from the CPU caches to PMem in the unit of sub-MemTable (with several MBs), thereby mitigating metadata updates and saturating the write-combining buffer in the Optane PMem. CacheKV finally accelerates read operations by periodically compacting sub-skiplists. To the best of our knowledge, CacheKV is the first work that brings the persistent CPU caches into the KV store designs (Figure 1). The contributions of this paper are summarized as below.

- We unveil via real-world benchmarks that the persistent CPU caches cannot naturally expedite the performance of existing KV stores and capture the root causes (Section II-C).
- We design CacheKV, the first KV store that is built atop the Optane PMem with the persistent CPU caches. CacheKV carefully aggregates incoming writes in the CPU caches and defers the updates to the index structures without compromising consistency. It then periodically flushes the packed KV pairs to the PMem to improve the write throughput, and compacts the sub-skiplists to accelerate read operations (Section III).
- We implement a CacheKV prototype atop LevelDB [2] by re-designing its storage hierarchy, process flow for KV requests, and metadata synchronization (Section III).
- We conduct extensive testbed experiments, showing that CacheKV improves the write throughput by 19.5× on average without compromising the read performance, when compared to state-of-the-art KV designs with the PMem and their variants leveraging persistent CPU caches (Section IV).

We have released the source code of the CacheKV prototype: https://github.com/shenzr/CacheKV-code.

II. Background and Observations

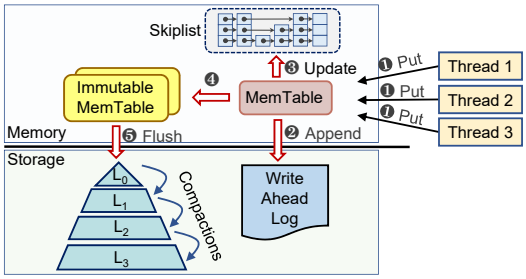


Figure 2: Operation flows of the LSM-tree-based KV stores to serve incoming writes, where there are four levels (i.e., $n = 3$).

A. LSM-Tree-Based KV Stores

A large body of studies [2], [3], [7], [14]–[20], [43] are popularly employing *log-structured merge tree* (LSM-tree) [21] to build KV stores on traditional block storage devices (e.g., HDDs and SSDs), whose main idea is to (i) aggregate random small-sized writes in memory to generate sequential writes and (ii) keep most of the KV pairs fully sorted in storage, so as to achieve high-speed writes and fast reads. Figure 2 summarizes the organization of the LSM-tree-based KV stores, which comprises a *memory component* and a *storage component*.

Memory component: It comprises a MemTable, an Immutable MemTable, and the index structure (skiplist as an example in Figure 2) to index the tables. When a new KV pair reaches (Step ❶ in Figure 2), an LSM-tree-based KV store first persists it to an on-disk *write-ahead log* (WAL) [44] for crash consistency (Step ❷). It then appends the KV pair to a *MemTable* and updates the index structure of the MemTable (Step ❸), which is responsible for quickly indexing the unsorted KV pairs in the MemTable. To regulate concurrent writes issued to the commonly shared MemTable, the KV store often employs synchronization mechanisms (e.g., mutex locks) to enable exclusion [2], [3], [32], [33]. When the MemTable becomes full (i.e., reaching a pre-configured size), it will be transformed into an *Immutable MemTable* (Step ❹), while at the same time another new MemTable is created to serve the subsequent writes. The generated Immutable MemTable will then be sealed and flushed to the storage component (Step ❺) in the form of *Sorted String Tables* (SSTables).

Storage component: The LSM-tree-based KV stores further organize the data in storage into $n + 1$ hierarchical levels (Figure 2). Usually, a level has a storage capacity that is multiple times (e.g., 10 times in LevelDB [2]) of the lower level. On the other hand, among the $n+1$ levels, KV pairs in the lowest level L_0 are partially sorted (for fast persistence), while those in other n higher levels (i.e., $\{L_1, L_2, \cdots, L_n\}$) are all fully sorted by keys (for fast KV lookups).

B. Persistent Memory

The Optane PMem [31] is the first commercial PMem device implemented based on the 3D-XPoint technology. Figure 3 presents the architecture overview of the Optane PMem. By connecting to processors directly through *integrated memory controller* (iMC), the Optane PMem can persist

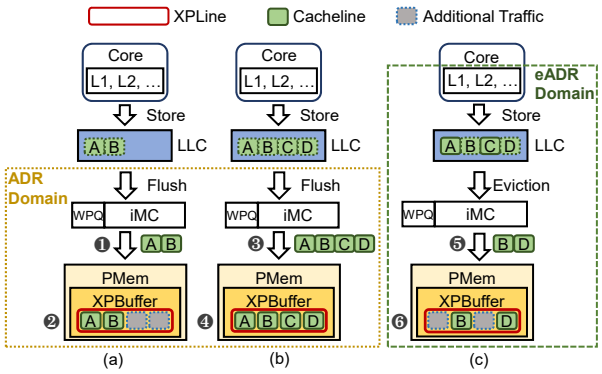


Figure 3: Architecture overview of the ADR-enable and eADR-enabled Optane PMems. (a) The small writes in ADR-enabled Optane PMem will introduce the write amplification. (b) The large writes in ADR-enabled Optane PMem can suppress the write amplification. (c) The large writes in eADR-enabled Optane PMem will reawaken the write amplification caused by the cacheline eviction algorithms.

data with high write throughput (e.g., 2.3 GB/s with a single DIMM [45]), low read latency (e.g., merely $2\times \sim 3\times$ higher than DRAM [45]), and large storage capacity (e.g., up to 512 GB per DIMM [31]). Hence, the Optane PMem is often considered to complement (or even substitute) DRAM in memory hierarchy [32], [46] and co-exist with block storage devices [15], [33] for building highly efficient KV stores. Moreover, the Optane PMem exhibits two intrinsic features which may revolutionize the KV store designs.

Feature 1 (Mismatch of access granularities between the CPU caches and the Optane PMem media): The Optane PMem embeds an internal write-combining buffer (called "XPBuffer") to stage the cachelines (usually in the granularity of 64 B) coming from CPU caches and has a constant media access granularity of 256 B (called "XPLine"), making the Optane PMem much more like a block device, rather than a fully byte-addressable memory device commonly assumed in prior studies [32], [34], [35], [40], [41]. The mismatch between the cacheline access granularity (from the CPU caches) and the XPLine access granularity (to the Optane PMem media) induces *write amplification* for the writes that operate smaller than 256 B of data, which triggers additional read-modify-write operations to encapsulate an XPLine and write the resulting 256 B data to the physical Optane PMem media. Figure 3(a) depicts the process of a small write operation in the Optane PMem, which flushes the cachelines A and B (with 64 B each) from the CPU caches to the Optane PMem (Step ❶). In this case, the Optane PMem has to complement an XPLine using additional data (marked in dashed lines, Step ❷), thereby amplifying the write traffic. Figure 3(b) shows that the large writes (whose sizes are larger than the XPLine) using the flush instructions can fill an XPLine (Steps ❸ and ❹) to eliminate the write amplification.

Feature 2 (Persistent boundary is shifted from memory up to CPU caches): The Optane PMem supports two kinds of *persistent domains* (i.e., data paths where stores are safe against power-failures), namely *Asynchronous DRAM Refresh* (ADR) and *enhanced Asynchronous DRAM Refresh* (eADR) [47]. Figure 3 compares the differences between the ADR-enabled and eADR-enabled Optane PMems. The ADR-enabled Optane PMem ensures that even in the presence of power failures, the writes currently in the *write pending queue* (WPQ) of the iMC can still be persisted onto the PMem. As the ADR persistent domain only includes the PMem and the WPQ, it needs to proactively flush data out of the CPU caches using dedicated cacheline flush instructions (e.g., `clflush` and `clwb`) and memory barrier (Figure 3(a) and (b)), ensuring that the data can be persisted in the right order.

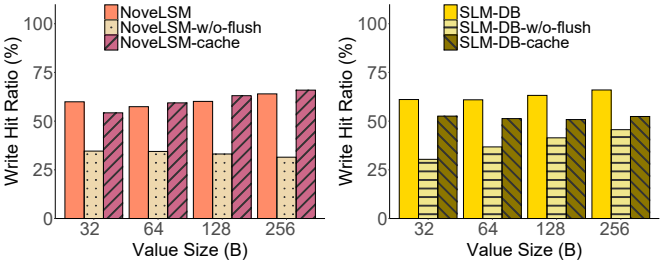
The eADR-enabled Optane PMem further shifts the persistence boundary up to CPU caches [2]. This feature ensures that data can be directly persisted in the CPU caches without needing additional expensive flush instructions [48], thereby favoring the overall performance. It exhibits that the performance doubles without flush instructions [47]. However, removing flush instruction might reproduce the write amplification problem, as the cacheline eviction is fully controlled by the cache eviction algorithms (e.g., Least Recently Used (LRU)). Figure 3(c) shows an example of a large write operation in the eADR-enabled Optane PMem, implying that even for the large writes, the eviction algorithm will be likely to evict a small number of cachelines (e.g., B and D) to reproduce the write amplification problem (Steps ❺ and ❻). In this paper, we mainly consider the eADR-enabled Optane PMem and design KV stores based on the persistent *last-level cache* (LLC) in non-inclusive cache hierarchy [49], as it can supply enough cache space (12 MB-60 MB [42]). We also demonstrate that our design is insensitive to different occupied cache space, indicating that it can achieve good performance even given limited cache space (e.g., 3 MB, Section IV-C). Finally, our work can also be extended to other PMem products once they have persistent CPU caches and exhibit the same mismatch problem in access granularity.

C. Analysis and Observations

To examine if existing KV stores can benefit from the persistent CPU caches, we conduct preliminary experiments with two state-of-the-art KV stores that are both designed for the PMem: (i) NoveLSM [32], which places a MemTable in PMem to leverage its persistent nature and avoid logging overhead with in-place durability; and (ii) SLM-DB [33], which maintains a MemTable and a B+-tree index in PMem to accelerate the key searches. We measure the random write performance using `db_bench` with the configurations in Section IV-A. We make two observations from this analysis.

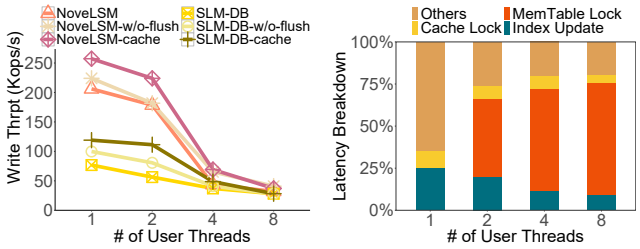
Ob1 (The direct utilization of the persistent CPU caches amplifies the write traffic). We first investigate the efficiency of NoveLSM and SLM-DB when both of them are directly deployed with the eADR-enabled Optane PMem (with persistent CPU caches). Note that the raw NoveLSM and SLM-DB both rely on the `store` instruction coupled with cacheline flush

[2]This feature is supported in the third generation of Intel® Xeon® Scalable Processors, which have entered the marketplace [42].



(a) NoveSLM and its variants. (b) SLM-DB and its variants.

Figure 4: Ob1 (The direct utilization of the persistent CPU caches amplifies the write traffic): removing the flush instructions and lifting MemTables into CPU caches both deteriorate the write hit ratios.



(a) Write Throughput. (b) Latency Breakdown.

Figure 5: Ob2 (Heavy software designs also compromise the performance gains provided by the Optane PMem:). The value size is set to 64 B. The "others" in (b) includes other necessary operations in writes, including syscalls, Linux kernel I/O stack, and media write latencies.

instructions to persist a cacheline from the volatile CPU caches to the PMem. Hence, when deploying them atop the eADR-enabled Optane PMem, we can generate another two variants of NoveLSM and SLM-DB, which remove the cacheline flush instructions (as they are no longer needed in the persistent CPU caches). We call them "NoveLSM-w/o-flush" and "SLM-DB-w/o-flush", respectively.

As the persistent CPU caches mainly affect the procedure of the write operations (especially the write path from the CPU caches to the Optane PMem), in this experiment we mainly focus on the *write hit ratio* (i.e., ratio of writes that hit the XPBuffer). This metric can be measured by the hardware counters of the Optane PMem [50] and is also used in previous studies to characterize the system efficiency with the Optane PMem [3] [53]. Generally, a higher write hit ratio indicates the better utilization of PMem bandwidth, as more write operations can directly update the pre-stored data in the XPLine without inducing any additional read-modify-write traffic. Since a large number of values manipulated by KV stores are often small-sized [54], [55] (e.g., from 57 B to 153 B [54] at Facebook), we vary the value size from 32 B to 256 B (with one single thread) and show the results in Figure 4. We surprisingly find that compared to the raw NoveLSM and SLM-DB, the NoveLSM-w/o-flush and SLM-DB-w/o-flush respectively reduce 43.5% and 45.2% of the write hit ratios on average, implying that directly deploying existing KV stores with the eADR-enabled Optane PMem amplifies the write traffic in the underlying PMem media.

Root cause (R1): We further delve into the underlying reasons and identify that the flush instruction not only forces the write back of the cachelines, but also maintains the flush sequence of them, which favors the combination of adjacent cachelines in XPBuffer and hence suppresses the write amplification. Without flush instruction, the cacheline eviction will be controlled by classical cacheline replacement algorithms (e.g., LRU algorithm). The small-sized (64 B) and randomized eviction will amplify the internal write traffic (Figure 3(c) in Section II-B) in the PMem.

Ob2 (Heavy software designs also compromise the performance gains delivered by the Optane PMem). To eliminate the write traffic caused by the evictions from the persistent CPU caches, we propose to lift the MemTable (with the size ranging from 8 MB [16], [56] to 64 MB [32], [33]) into the CPU caches (i.e., keeping the MemTable in LLC, which has the size of up to 60 MB [42]), so as to directly absorb the small-sized write operations in the CPU caches. We implement this approach [4] for both NoveLSM and SLM-DB, and generate another two variants of them, named "NoveLSM-cache" and "SLM-DB-cache", respectively (see Section IV-A for details about these two implementations). We find that this approach takes effect, where the reductions on the write hit ratio shrink to 5.5% and 15.9% when compared to the vanilla systems (i.e., NoveLSM and SLM-DB), respectively (Figure 4).

We further examine the write performance: we start random write benchmark and dispatch 10 million write requests, each of which has the value size of 64 B, with different number of user threads. We find that both NoveLSM-cache and SLM-DB-cache outperforms its vanilla systems by improving the write throughput (Figure 5(a)). The write throughput of all the six systems are all lower than 260 Kops/s with the single user thread (Figure 5(a)). For instance, even we increase the number of user threads, the accumulated write throughput still drops from 257.3 Kops/s (with a single thread) to 37.12 Kops/s (with eight threads in total) for NoveLSM-cache; other five systems showcase the similar trend.

Root cause (R2): We further analyze the breakdown of the average write latency of NoveLSM-cache (from the time when a newly written KV pair arrives to the time when it is recorded in the MemTable) and show the results in Figure 5(b). We identify that the index update and MemTable lock (used for regulating concurrent writes to the single shared MemTable) take up 46.3% (with two threads) and 67.0% (with eight threads) of the overall write latency, indicating that the index overhead and the synchronization on the shared data structures emerge to be the new performance bottleneck.

[3] A variant of the write hit ratio is called the *write amplification ratio* (or *effective write ratio*), which is also extensively employed in existing studies [45], [51], [52] for assessing the system performance with the PMem.

[4] We can place and control the MemTables in the CPU caches via employing "Cache Pseudo-Locking" [57] with *Intel Cache Allocation Technology* (Intel CAT) [58].

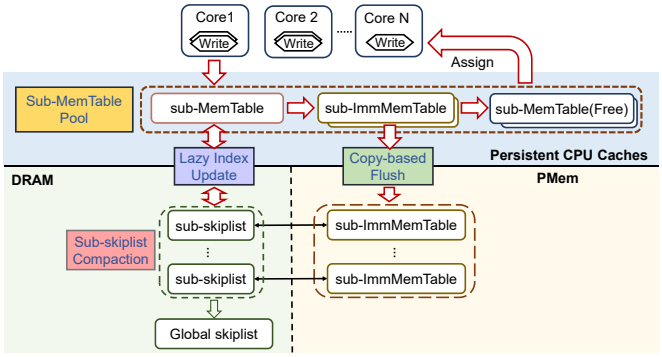


Figure 6: Overview of CacheKV. The techniques proposed in CacheKV are marked in colorful boxes.

We also notice that NoveLSM gains a higher write hit ratio (Figure 4(a)) but lower write throughput (Figure 5(a)) than NoveLSM-w/o-flush, as NoveLSM comprises the overhead induced by the flush instruction [59], [60].

Summary: Therefore, directly deploying existing KV stores (Ob1) or trimming them (Ob2) both cannot achieve expected write performance for the PMem with the persistent CPU caches, stemming mainly from the mismatch of the hardware access granularities (R1) and the heavy software designs (i.e., the index update and the synchronization mechanism to the shared MemTable, see R2).

III. CacheKV Design

We present CacheKV, a high-performance KV store that is designed for the Optane PMem with the persistent CPU caches. Figure 6 shows the architecture of CacheKV.

Overview: To eliminate the synchronization overhead on the shared MemTable, CacheKV partitions a MemTable into multiple small-sized *sub-MemTables*, which are organized into a sub-MemTable pool in the CPU caches and shared across CPU cores (Section III-A). CacheKV then assigns a sub-MemTable as well as the associated *sub-skiplist* to each CPU core, such that the incoming KV pairs served by different CPU cores can be directly appended to the corresponding sub-MemTables without incurring expensive synchronization overhead, thereby improving the access parallelism (R2 is addressed). To avoid frequent updates to the sub-skiplists when multiple KV pairs are written, CacheKV designs a *lazy index update mechanism* (Section III-B), which defers the updates to the sub-skiplist and performs them in batch using background threads. For a CPU core, once the assigned sub-MemTable is filled up, CacheKV transforms it into a *sub-ImmMemTable*, which is then flushed to the Optane PMem using *copy-based flush mechanism* (Section III-C) without relying on the classical cacheline replacement algorithms, so as to resolve the mismatch of the access granularities between the CPU caches and the Optane PMem (R1 is addressed). Finally, CacheKV regularly performs the *sub-skiplist compaction* (Section III-D), so as to eliminate useless search efforts and accelerate the search efficiency. We finally implement a prototype of CacheKV atop LevelDB [2].

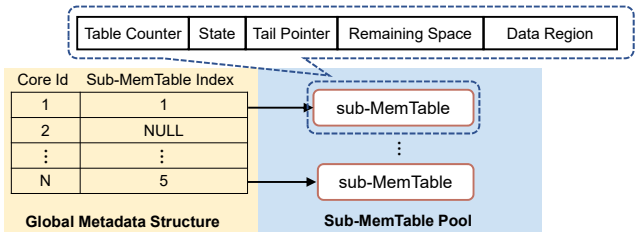


Figure 7: Organization of the sub-MemTable pool and a global metadata structure.

A. Per-Core Sub-MemTable

Previous KV stores [2], [3], [7], [14]–[16], [32], [33] often maintain a single MemTable in DRAM (or PMem) to serve the incoming write requests. Since the CPU caches become persistent in the eADR-enabled Optane PMem, we can treat them as a new persistent storage layer (with ultra-low access latency) to complement existing storage hierarchy. To leverage this feature, simply lifting the single MemTable from the memory up to the CPU caches cannot support highly concurrent accesses due to the expensive synchronization overhead (Ob2 in Section II-C).

In view of this, CacheKV proposes to organize a sub-MemTable pool in the CPU caches which contains multiple small-sized sub-MemTables. Each CPU core will be assigned with a dedicated sub-MemTable on demand (i.e., when a write request arrives), hence improving the service parallelism without inducing the synchronization overhead. CacheKV then fixes the capacity of the sub-MemTable pool, while allowing the sub-MemTables to have varied sizes, so as to adapt to the dynamically changing workloads (see Exp#6 in Section IV-C).

Specifically, CacheKV allocates the space to serve as the sub-MemTable pool in PMem and specifies it to a dedicated CPU cache space by using Intel CAT [58], which grants CacheKV the software-programmable control over the given amount of cache space, such that no application (except CacheKV) is allowed to fill this cache space [5]. CacheKV logically breaks the reserved cache space into fixed-size areas, where each area will be used to store a sub-MemTable. Hence, the sub-MemTables constructed in the reserved cache space collectively constitute a sub-MemTable pool.

Structure: Figure 7 presents the organization of a sub-MemTable pool and the associated global metadata structure. The global metadata structure establishes the mapping between the CPU cores and the associated sub-MemTables. For the cores that have not been assigned with sub-MemTables yet, CacheKV sets the associated indices as NULL. Besides, a sub-MemTable comprises five components: (i) a `table counter` field (with the size of 38 bits), which records the number of KV pairs currently stored in the sub-MemTable and can also be served as a version tag; (ii) a `state` field

[5] Although some instructions (e.g., `clflush` and `invd`) will still evict the data in the "locked" area of the CPU caches, the per-core sub-MemTable is exclusive to other applications and protected by the memory management of OS.

(with the size of 2 bits), which implies if the sub-MemTable has been assigned to a CPU core and is initialized to be "Free" at the very beginning; (iii) a `tail pointer` field (with the size of 24 bits), which records the beginning offset of this sub-MemTable that can be written by the next arrived KV pair; (iv) a `remaining space` field (with the size of 64 bits), which stores the residual size of the available area to accommodate the new KV pairs at present; and (v) a `data region` field, which stores the KV pairs that have been inserted till now. CacheKV further saves the metadata overhead and promises the consistency via encapsulating the values of `table counter`, `state`, and `tail pointer` into a unit of 64 bits, which can be updated together using a 64-bit atomic compare-and-swap operation [12].

Write process: When a write request arrives, CacheKV first checks the global metadata structure, which records the index and the address of the associated sub-MemTable. The global metadata structure is kept in DRAM to avoid write amplification to the Optane PMem. If there is no sub-MemTable assigned to this CPU core at present, CacheKV will pick out a free one from the pool and set its state as "Allocated". Otherwise, it learns the beginning offset of the associated sub-MemTable for writes (by reading the `tail pointer`). If the remaining space is enough to accommodate the newly inserted KV pair, then it appends the KV pair to the tail of the associated sub-MemTable. CacheKV finally updates the `tail pointer` and the `table counter` in an atomic operation, so as to promise the consistency even in the presence of system crashes during the updates to the metadata structure. When a sub-MemTable runs out of space, it will be transformed into a sub-ImmMemTable with the `State` being set to "Immutable". This sub-ImmMemTable is then flushed to the Optane PMem using the copy-based flush mechanism (Section III-C); after that, its `State` is modified to "Free", indicating that it can be reused and ready to be re-assigned to a CPU core on demand. We also elaborate on the read process of CacheKV in Section III-D.

Elasticity: The number of the sub-MemTable (as well as their sizes) should be carefully adjusted so as to balance between the write concurrency (improved with more sub-MemTables) and the background flush overhead (reduced with larger sub-MemTable sizes). Here, CacheKV elastically tunes the number of the sub-MemTable in response to the dynamically changing workloads. Specifically, CacheKV maintains a global variable called `miss counter`, which records the times that a CPU core cannot find a free sub-MemTable to accommodate the newly written KV pairs (e.g., when all the sub-MemTables have been occupied or become full). Hence, the value of the miss counter indicates the contention on the sub-MemTables. When the value of the `miss counter` exceeds a pre-configured threshold, CacheKV will seek to increase the number of the free sub-MemTables by shrinking their sizes (e.g., halving the size of a free sub-MemTable). By doing so, CacheKV can ensure that there are enough free sub-MemTables to supply even in the face of bursty writes. The value of the `miss counter` is re-initialized to be zero after CacheKV adjusts the number of the sub-MemTables. Conversely, CacheKV can also trim the number of the sub-MemTables by enlarging their sizes so as to reduce the background flush overhead.

Discussion: Existing studies [23], [37], [61] also employ multi-shard memory component to improve the write parallelism, which organizes multiple MemTables for multiple *shards* and each shard covers an equal range of hashed-key space. Hence, when a new KV pair arrives, it will be dispatched to the corresponding MemTable based on the hash value of its key. Compared to this multi-shard design, CacheKV has the following advantages. First, when multiple KV pairs are directed to the same shard simultaneously, it still has to rely on the synchronization mechanism to regulate the writes. As a comparison, CacheKV suppresses the synchronization overhead by assigning a sub-MemTable to each CPU core. Second, the multi-shard design lacks of flexibility, in that the number of shards is pertinently fixed once the key space is partitioned, which is hard to adapt to the dynamically changing workloads; conversely, CacheKV can elastically tune the size and the number of sub-MemTables in response to the workload changes. However, the sub-MemTable design in CacheKV also comes with read amplification, since CacheKV has to search across multiple sub-MemTables to find the wanted KV pair. Therefore, the search complexity in the memory component is $O(m \cdot \log_2 \frac{N}{m})$, where N is the number of KV pairs stored in the memory component and m is the number of the sub-MemTables used. To mitigate this issue, we reduce unnecessary search efforts in Section III-D. For the correctness of multi-key transaction [62], we can bind each transaction thread to a specified core, ensuring that the inserted KV pairs in this transaction are stored within the same sub-MemTable.

B. Lazy Index Update

In traditional MemTable designs (Section II-A), the incoming KV pairs are directly appended to the MemTable for fast writes and hence the KV pairs in the MemTable are often out-of-order. To enable fast searches over those disordered KV pairs, existing KV stores usually attach an index structure (usually skiplist [2], [3], [32], [33]) with the MemTable. Yet it also comes with additional latency (e.g., around 24.7% of the write latency observed in Section II-C) to synchronously update the index structure whenever a new KV pair is written. In view of this, CacheKV proposes a *lazy index update mechanism*, whose main idea is to maintain a skiplist (called "sub-skiplist") with each sub-MemTable for fast searches but postpone the updates to the index structures, which will then be performed in batch via background threads.

Index of the sub-MemTable: As CacheKV couples a sub-skiplist with each sub-MemTable, the updates to multiple *independent* sub-MemTables can be recorded by the associated sub-skiplists in parallel without inducing any synchronization overhead. Instead of keeping the sub-skiplists along with the sub-MemTable in the CPU caches, CacheKV chooses to

maintain them in DRAM, which can bring forth the following advantages: (i) the updates to the sub-skiplists can be performed in parallel to exploit the fast access performance of DRAM [34], hence greatly shortening the update latency; (ii) the write amplification problem caused by the mismatch of the access granularities can be suppressed, especially for the small updates to the sub-skiplists; and (iii) CacheKV can save the the cache footprints to absorb more key-value pairs. Although DRAM is volatile (i.e., the sub-skiplists will be lost once encountering system crashes), CacheKV can reconstruct the sub-skiplists based on the KV pairs stored in the persistent CPU caches (Section III-E).

CacheKV also keeps a `list counter` and a `list tail pointer` for each sub-skiplist, which track the number of KV pairs currently stored in the sub-skiplist and the offset of the sub-MemTable ended at last synchronization, respectively; similarly, the `list counter` can be used to check the consistency between a sub-skiplist and the associated sub-MemTable, by simply comparing the values of the `list counter` and the `table counter`.

Lazily updating sub-skiplists in background: CacheKV then detaches the synchronization of the sub-skiplists from the critical write path. It proposes to lazily update the sub-skiplists in the background, with the objective of incurring no interference to the ongoing write operations. CacheKV also provides three strategies to trigger the skiplist synchronization, with different preferences on read consistency, synchronization overhead, and memory consumption, respectively. First, it must start the synchronization whenever a read operation arrives. The rationale is that to serve a read request, CacheKV has to pinpoint the location of the freshest value across multiple sub-MemTables. At this time, the consistency between the sub-skiplist and the corresponding sub-MemTable must be strictly guaranteed; otherwise, CacheKV may mistakenly fetch the outdated one for a given key, even though its newest value is currently stored in a sub-MemTable yet is not recorded in the corresponding sub-skiplist. This synchronization will incur low overhead for the workloads that are either write-dominated or read-dominated. Second, for the write-dominated workloads, CacheKV can also launch a new synchronization for a sub-skiplist, when the number of write operations (to the corresponding sub-MemTable) after last synchronization reaches a pre-configured threshold (whose value is often tunable). Third, when the capacity of a sub-MemTable is exhausted (i.e., it is incapable of storing any new KV pair), CacheKV can synchronize the newly written KV pairs to the corresponding sub-skiplist.

We now elaborate on the process of the synchronization. CacheKV first fetches the `table counter` from the operated sub-MemTable and also the `list counter` from the associated sub-skiplist, respectively. It then compares the values of the two counters: if they are unequal (indicating that there are some KV pairs updated since the last synchronization yet are not tracked by the sub-skiplist), then CacheKV updates the sub-skiplist in the following steps: (i) it fetches the KV pairs from the sub-MemTable, starting from the offset to which the `list tail pointer` points; (ii) it then inserts the address of the KV pairs into the sub-skiplist; and (iii) it finally increases the value of the `list counter` and updates the `list tail pointer` accordingly. CacheKV repeats the synchronization until the values of the `list counter` and the `table counter` are equal. Note that, during the index updates, the sub-MemTables can continue serving the incoming write requests without being affected by the synchronization.

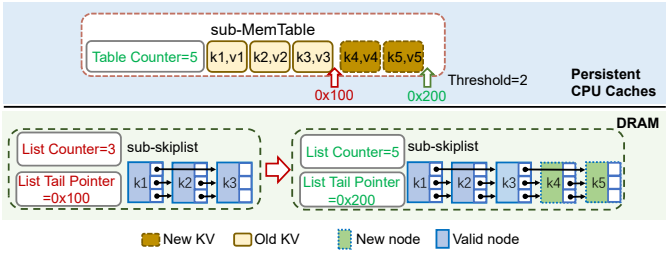


Figure 8: Example of the lazy index update mechanism. The valid nodes here denote the nodes that index the valid KV pairs.

Figure 8 shows an example of the lazy index update. CacheKV compares the values of the `table counter` (i.e., 5) and the `list counter` (i.e., 3), learning that two KV pairs are written in the sub-MemTable without being tracked by the sub-skiplist. It then updates sub-skiplist based on the sub-MemTable (i.e., adding k_4 and k_5) and renews its tail pointer accordingly.

C. Copy-based Flush

To suppress the write amplification caused by the mismatch of the access granularities (Ob1 in Section II-C), CacheKV proposes a *copy-based flush mechanism*, whose main idea is to write back the data from the CPU caches to the Optane PMem in unit of the sub-MemTables (with the size of multiple MBs), so as to suppress the write amplification induced to the PMem.

When designing the flush mechanism, a native approach is to keep all the sub-ImmMemTables in the CPU caches and flush them to the Optane PMem until the sub-MemTable pool is fully exhausted (i.e., CacheKV cannot dispatch any free sub-MemTable to a CPU core). However, this approach is rather inefficient, as no available sub-MemTable can be used to serve the incoming write requests during the flush operation. Consequently, CacheKV has to experience write stalls when flushing all the sub-ImmMemTables to the Optane PMem.

To retain the write parallelism, CacheKV proposes to launch a flush operation whenever a sub-MemTable becomes full. Remember that once a sub-MemTable runs out of space, it will be transformed into a sub-ImmMemTable and not be updated in the cache again (Section III-A). CacheKV then employs a *modified memory copy* operation, which replaces the originally used `store` instruction with the `non-temporal store` instruction in the memory copy operation. Hence, when a sub-MemTable is full, CacheKV calls the modified memory copy operation, which takes the address of sub-ImmMemTable and the destination address (in the Optane PMem) as input, such that the sub-ImmMemTable (resided in

the CPU caches) can be copied (flushed) to the given address of the Optane PMem directly without being controlled by the cacheline eviction, hence favoring the write performance of the Optane PMem. Once the copy-based flush operation completes, CacheKV reclaims the space occupied by the sub-ImmMemTable by re-initializing the metadata information and setting its state as "`Free`" by using an atomic operation (Section III-A).

D. Compaction of Sub-Skiplists

Till now, CacheKV mainly considers improving the write performance of the KV stores with the persistent CPU caches. When a read request comes, CacheKV first goes through the global metadata structure (Figure 7) to find the sub-MemTables in used and then scans each sub-MemTable for key lookups. CacheKV may lengthen the read operation, as a read operation may trigger the update to the sub-skiplist (Section III-B). To further improve the read efficiency, CacheKV proposes to proactively compact the sub-skiplists.

Sub-skiplist compaction: Remember that when the sub-MemTable is full, it will be transformed into a sub-ImmMemTable and flushed to the Optane PMem (Section III-C). Hence, the Optane PMem may store sub-ImmMemTables along with the associated sub-skiplists (in DRAM), which still host a large number of valid KV pairs. At this time, if a requested KV pair cannot be found across the sub-MemTables in the CPU caches, then CacheKV will turn to look it up across the sub-ImmMemTables. However, directly searching a given key across these sub-ImmMemTables is extremely inefficient, since they may comprise a large number of out-dated (or invalid) KV pairs (either deleted or invalidated), which inevitably prolongs the search time. To improve the search efficacy, we propose to merge the sub-skiplists of the sub-ImmMemTables into a *global skiplist*, which removes the useless nodes in the sub-skiplists to reduce the redundant search trials. When merging the sub-skiplists, the invalid KV pairs in the corresponding sub-ImmMemTable are not removed immediately, which is rather different from the compaction in traditional LSM-tree-based KV stores (Section II-A). CacheKV defers the space reclamation until the sub-ImmMemTables are flushed to the L_0 level of the LSM-tree (this operation is triggered once the total size of sub-ImmMemTables reaches a pre-configured threshold).

Figure 9 shows an example of the sub-skiplist compaction. There are two sub-skiplists, where the `sub-skiplist-1` has one invalid node (filled in black color) and the `sub-skiplist-2` includes another two invalid nodes. Hence, when performing the compaction, we can delete the three invalid nodes and generate a global skiplist comprising all the valid nodes, so as to reduce the search time in next read operations.

E. Crash Recovery

CacheKV provides crash consistency guarantee for the data that have been successfully committed to the sub-MemTable in persistent CPU caches. When a failure occurs, the whole sub-MemTable pool (resided in the CPU caches) will be written back to the PMem, while the sub-skiplists and global skiplist resided in DRAM will be lost.

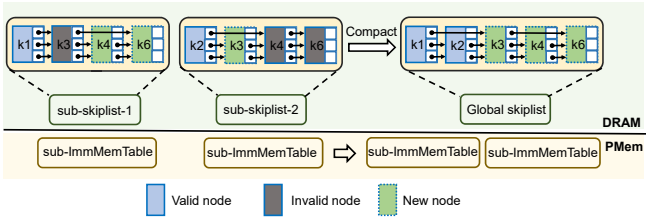


Figure 9: Example of the sub-skiplist compaction.

CacheKV rebuilds the sub-skiplists and the global skiplist in the background: (i) for each sub-MemTable, it initializes a sub-skiplist and proactively synchronizes it with the corresponding sub-MemTable; and (ii) it performs the sub-skiplist compaction to rebuild the global sub-skiplist. After that, CacheKV can serve the incoming read requests.

To continue serving the write requests, CacheKV restores the sub-MemTable pool in the CPU caches using Intel CAT. It also identifies all the allocated sub-MemTables and changes their states to "Free", such that they can be re-assigned to the CPU cores again once the write requests arrive.

IV. Evaluation

We conduct extensive testbed experiments to evaluate CacheKV and summarize the major findings as below:

- Compared to NoveLSM [32] and SLM-DB [33], CacheKV improves the write throughput by 17.2× and 26.1× on average, respectively (Exp#1 and Exp#3 in Section IV-B). Besides, CacheKV achieves similar read throughput (Exp#2 in Section IV-B). Hence, we suggest deploying CacheKV in the write-dominated scenarios.
- The design techniques in CacheKV are mutually complementary (Exp#1 in Section IV-B);
- The numbers of user threads and background flush threads should be collectively considered to maximize the performance of CacheKV (Exp#3 in Section IV-B and Exp#5 in Section IV-C).
- The performance of CacheKV is not definitely improved with the increase of the pool size (Exp#7 in Section IV-C). Hence, CacheKV is also effective when given limited cache space.

A. Experimental Setup

Testbed: We conduct experiments on a single machine equipped with two 2.10 GHz Intel Gold 5318Y CPUs (with 24 cores each), 128 GB of DRAM memory, and four Optane PMem DIMMs of 200 series (128 GB per DIMM and 512 GB in total). The Optane PMem DIMMs are configured in interleaved `App Direct Mode`, which are connected to one processor. Each processor has a shared 36 MB LLC. The machine runs Ubuntu 20.04 with the kernel version of 5.4.0.

Configurations: Without otherwise specified, we select the following configurations throughout the evaluation. We set

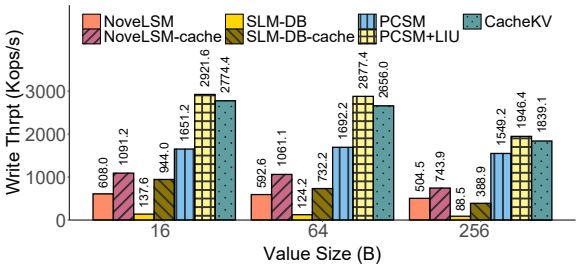
the size of the sub-MemTable pool to 12 MB, ensuring that its size is smaller than that of the LLC. We then configure the size of a sub-MemTable as 2 MB, which is observed to achieve good access performance [45]. We also investigate the performance under different sub-MemTable sizes in Exp#6. We use one background thread for the copy-based flush (Section III-C) and employ another thread to perform the lazy index update (Section III-B) and the sub-skiplist compaction (Section III-D). As the values are often small-sized in practical storage systems, we set the value size to 64 B (also used in FlatStore [38] and P^{2}KVS [23]). We perform the lazy index update when a read request arrives or the sub-MemTable is transformed into a sub-ImmMemTable. The cacheline size is 64 B in our testbed. When measuring the throughput, we launch 10 million requests in each test. We then repeat each test for five runs and average the results.

Comparison systems: We compare CacheKV against another two open-source KV stores that are also designed for the PMem, namely NoveLSM [32] and SLM-DB [33]. We summarize their main ideas as below.

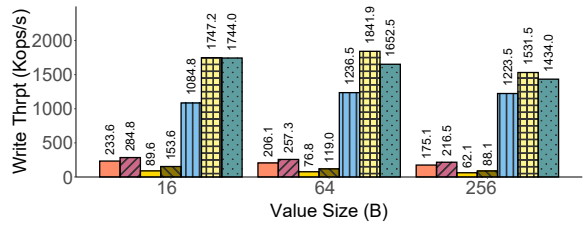
- **NoveLSM [32]:** It is an LSM-based KV store that maintains MemTables both in DRAM and PMem. Specifically, it allocates large MemTables in PMem for reducing the KV pairs flushed to the storage component of the LSM-trees. The MemTable in PMem also absorbs KV pairs once the MemTable in DRAM is full.
- **SLM-DB [33]:** It employs a persistent MemTable (with the size of 64 MB by default [33]) to aggregate incoming KV pairs, while storing KV pairs on disks with a single-level LSM-tree organization, so as to reduce the write amplification induced from the compactions. It also maintains a global B+-tree in PMem, which is used to accelerate the search process for the KV pairs on SSTables.

Both NoveLSM and SLM-DB are designed under the volatile CPU caches and hence they heavily employ flush instructions to persist data. For fair comparison, we generate another two variants for them (called "NoveLSM-cache" and "SLM-DB-cache"), both of which can leverage the persistent CPU caches without changing the main designs. We elaborate on the main extensions of the generated variants as follows.

- **NoveLSM-cache:** As the MemTable kept in PMem is configured as 4 GB in NoveLSM by default [32], which far exceeds the size of the CPU caches in our testbed (the size of LLC is 36 MB), we partition the MemTable into multiple *segments* with 12 MB each. For each time, we specify a constant size of the CPU cache space and pre-load a segment into the cache space to serve access requests. When a segment is full, it will be flushed to the PMem by using the `clflush` instruction and the successive segment is loaded into the CPU cache. The MemTable in DRAM is set to 64 MB as its vanilla system.
- **SLM-DB-cache:** We amend SLM-DB in the similar way. As it only maintains a single MemTable in PMem, we adjust the MemTable to 4 GB for a fair comparison, which is the same with the size of the MemTable in NoveLSM.



(a) Sequential writes.



(b) Random writes.

Figure 10: Exp#1 (Write performance).

Besides, we place all the SSTables of the KV stores in the Optane PMem (as in NoveLSM [32] and ChameleonDB [37]), so as to maximize the utilization of the fast access and the large capacity provided by the Optane PMem.

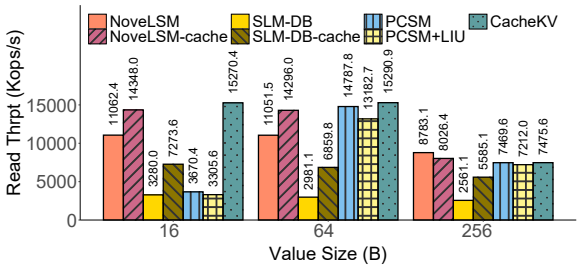
B. Experiments on CacheKV Property

We select two widely used benchmarks, namely `db_bench` [2] and YCSB [63], to evaluate the performance of CacheKV.

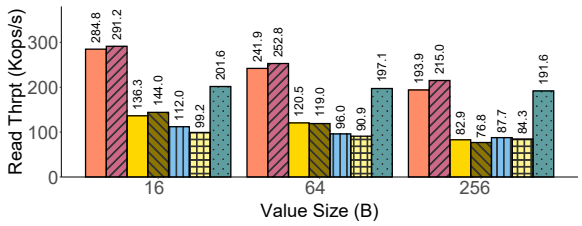
Breakdown of CacheKV: To demonstrate the efficiency gained by each design technique of CacheKV, we decompose CacheKV and abbreviate the techniques as follows: (i) *per-core sub-MemTable* (PCSM), which simply creates a sub-MemTable pool in the CPU caches and diligently updates the corresponding sub-skiplist whenever a KV pair is written (Section III-A); (ii) *lazy index update* (LIU), which defers the updates to the sub-skiplists and performs them in batch in the background (Section III-B); and (iii) *sub-skiplist compaction* (SC), which periodically compacts the sub-skiplists to reduce search efforts (Section III-D). Both PCSM and LIU employ the *copy-based flush* (CF) (Section III-C) to write back the sub-ImmMemTable. Hence, CacheKV can be treated as the combination of the three techniques (i.e., PCSM+LIU+SC).

Exp#1 (Write performance): We first assess the write throughput of the KV stores. We start a single thread and dispatch 10 million insert requests. We concern both the sequential writes and the random writes, where the keys are generated by following a sequential order and a uniformly random order, respectively. We vary the value size from 16 B to 256 B and set the key size to 16 B, as small KV pairs are predominant in real-world workloads [38], [55], [64]. Figure 10 shows the write throughput in operations per second. We make three observations.

First, CacheKV can improve 5.1× and 20.2× of the write throughput on average when compared to NoveLSM and

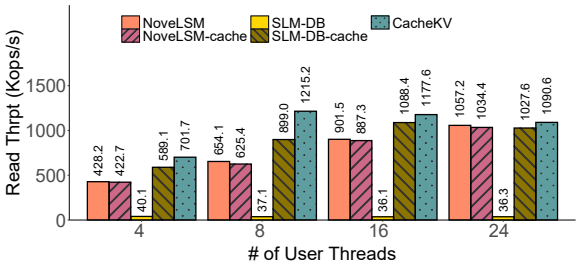


(a) Sequential reads.

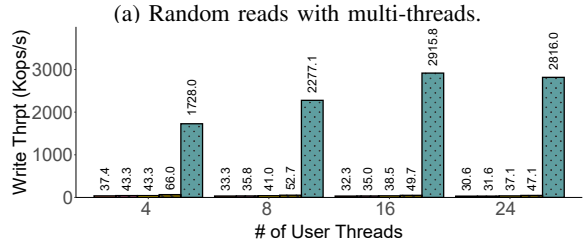


(b) Random reads.

Figure 11: Exp#2 (Read performance).



(a) Random reads with multi-threads.



(b) Random writes with multi-threads.

Figure 12: Exp#3 (Multi-threading performance).

SLM-DB, respectively. The underlying reasons are two-fold. On one hand, CacheKV utilizes the persistent CPU caches to aggregate the newly written KV pairs in the sub-MemTables, while at the same time suppressing the consistency overhead. On the other hand, CacheKV performs the lazy index update (Section III-B) which defers the updates to the sub-skiplists and performs them in batch in the background.

Second, CacheKV can improve 3.4× and 7.8× of the write throughput on average when compared to NoveLSM-cache and SLM-DB-cache, respectively. The reason is that CacheKV employs the sub-MemTables and the copy-based flush mechanism to concurrently absorb the incoming writes without introducing the write amplification.

Third, the techniques in CacheKV are mutually complementary to each other. In particular, compared to NoveLSM, solely employing the per-core sub-MemTable (i.e., PCSM) can improve 5.8× of the random write throughput (Figure 10(b)), while coupling the per-core sub-MemTable together with the lazy index update (i.e., PCSM+LIU) further gains 8.3× of the improvement on the random write throughput. This is because PCSM and LIU decouple the write procedures and optimize them independently: PCSM employs the sub-MemTables to immediately serve the incoming writes, while LIU mainly optimizes the metadata updates of the sub-MemTables. The designs of the sub-skiplist compaction (i.e., SC) only downgrade 8.3% of the write throughput compared to PCSM+LIU (by comparing CacheKV with PCSM+LIU), since SC mainly aims for accelerating the reads (Section III-D and Exp#2) by periodically compacting the sub-skiplists resided in DRAM.

Exp#2 (Read performance): We compare CacheKV with other KV stores on the sequential reads and the random reads. We start one thread and dispatch 10 million read requests. We vary the value size from 16 B to 256 B and set the key size to 16 B. Figure 11 shows the read throughput.

Figure 11 indicates that CacheKV achieves almost the same read throughput as both PCSM and PCSM+LIU in the sequential reads (Figure 11(a)), and outperforms them in the random reads (Figure 11(b)). This is because CacheKV incorporates SC (i.e., sub-skiplist compaction), which optimizes reads by saving the search trials, hence shortening the search time (especially when the keys are randomly retrieved). The effectiveness of SC decades in the sequential reads, since `db_bench` sequentially reads KV pairs from given sub-skiplists. PCSM+LIU gains lower read throughput than PCSM, since PCSM+LIU needs to update the sub-skiplists before serving the reads.

Besides, in random reads (Figure 11(b)), CacheKV gains lower read throughput compared to NoveLSM, since more KV pairs are staged in sub-MemTables, which introduce read amplification. Overall, CacheKV merely decreases 3.7% of the read throughput when compared to NoveLSM and its variant (NoveLSM-cache). Conversely, it increases 1.4× of the read throughput when compared to SLM-DB and its variant (SLM-DB-cache).

Exp#3 (Multi-threading performance): We also compare CacheKV against other KV stores on multi-threading performance, in which the number of user threads is varied from 4 to 24 (the number of physical cores within one socket). We set the key size to 16 B and the value size to 64 B.

Figure 12(a) shows that CacheKV outperforms its counterparts on the random reads under multiple user threads. The reason is that CacheKV maintains the index structures in DRAM while other systems store it along with the MemTable in the PMem, which introduces longer access latency. With respect to SLM-DB, it always showcases the lowest read throughput; this is because its MemTable is usually much smaller and intensive access requests are prone to competing for the shared SSTable metadata, hence limiting the access parallelism [33].

Figure 12(b) indicates that CacheKV gains higher throughput on the random writes. The write throughput of CacheKV

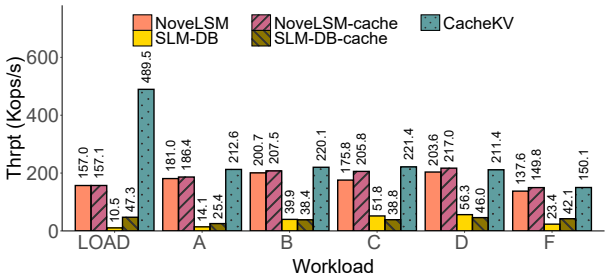


Figure 13: Exp#4 (YCSB evaluation).

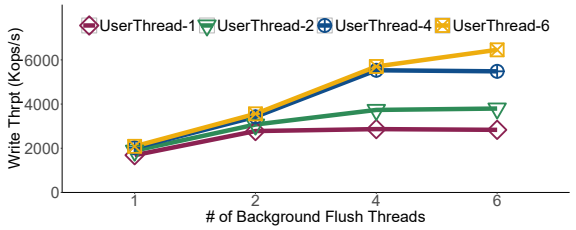


Figure 14: Exp#5 (Impact of number of background flush threads).

first climbs with the number of user threads (from 1,728.0 Kops/s to 2,915.8 Kops/s when the number of user threads increases from 4 to 16) and then slightly drops. This is because the bottleneck of the write performance shifts from the request serving to the background flushing. We further investigate the impact of the number of background flush threads in Exp#5 (Section IV-C). The write performance of the competitor systems degrades with the increase of the user threads, because the write contention to the shared data structure induces heavy synchronization overhead (Section II-C).

Exp#4 (YCSB evaluation): We evaluate the performance using YCSB [63] by launching five million requests with a single user thread. Each request comprises a 16-B key and a 64-B value. We select six representative YCSB workloads, which are briefly summarized as follows: (i) `YCSB-Load`, which performs 100% of insert (write) operations in Uniform pattern; (ii) `YCSB-A`, which comprises 50% of read operations and another 50% of update (write) operations in Zipfian distribution ($\alpha = 0.99$); (iii) `YCSB-B`, which performs 95% of read operations and additional 5% of update (write) operations in Zipfian distribution ($\alpha = 0.99$); (iv) `YCSB-C`, which purely performs read operations (i.e., with 100% of read operations) in Zipfian distribution ($\alpha = 0.99$); and (v) `YCSB-D`, which performs 95% of read operations to access the latest keys and conducts another 5% of insert (write) operations in Latest pattern; and (vi) `YCSB-F`, which performs 50% of read operations and additional 50% of read-modify-write operations in Zipfian distribution ($\alpha = 0.99$). We measure the throughput and show the results in Figure 13.

CacheKV can improve the throughput by 42.9% and 8.9× on average when compared to NoveLSM (NoveLSM-cache) and SLM-DB (SLM-DB-cache), respectively. The advantage of CacheKV is more salient for the write-dominated workloads (e.g., `YCSB-Load`). The rationale is that CacheKV improves the write concurrency via assigning a sub-MemTable to each CPU core for serving the incoming write operations. Meanwhile, CacheKV also employs the copy-based flush (Section III-C) to suppress the write amplification caused by the mismatch of the access granularities (Section II-B).

Even for the read-dominated workloads (i.e., `YCSB-B`, `YCSB-C`, and `YCSB-D`), CacheKV still increases 8.4% and 3.9× of the throughput on average when compared to NoveLSM (NoveLSM-cache) and SLM-DB (SLM-DB-cache), respectively. The underlying reasons are two-fold. First, CacheKV proactively compacts the sub-skiplists to remove the invalid nodes, so as to reduce the search efforts. Second, CacheKV maintains the sub-skiplists in DRAM and places the sub-MemTables in caches to exploit fast access performance, while NoveLSM and SLM-DB both maintain their skiplists and MemTables in PMem, which takes longer access latency.

C. Experiments on Sensitivity

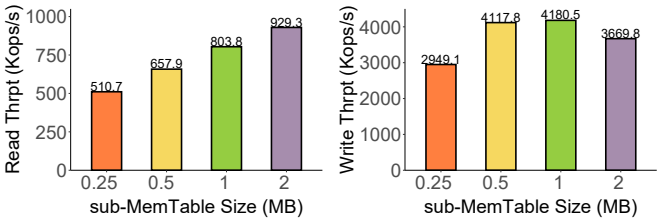
We finally study the impact of the configured parameters on the access performance, via varying the number of background flush threads (Exp#5), the sub-MemTable size (Exp#6), and the sub-MemTable pool size (Exp#7), respectively.

Exp#5 (Impact of number of background flush threads): We then measure the write throughput under different number of background flush threads, which are responsible for flushing the sub-ImmMemTables from the CPU caches to the Optane PMem (Section III-C). We vary the number of the background flush threads from one to six and show the results in Figure 14.

We can find that with more background flush threads, the write throughput climbs at first and then stabilizes. For example, when there are two user threads, CacheKV can improve the write throughput by 97.4% via increasing the number of background flush threads from one to four; furthermore, the write throughput merely increases 1.7% even we further launch two additional background flush threads (i.e., from four to six). The underlying reason is that the constant number of user threads will gradually become the performance bottleneck with the increase of the background flush threads.

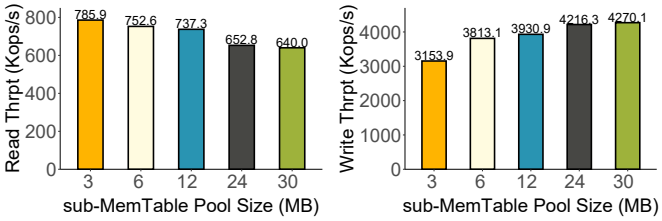
On the other hand, we can observe that the numbers of the user threads and background flush threads collectively determine the write throughput of CacheKV. Usually, when there are more user threads, CacheKV can achieve higher write throughput by accordingly dispatching more background flush threads. For example, when there are two user threads, CacheKV can reach the peak write throughput (around 3,797.7 Kops/s) via running six background flush threads; moreover, the peak write throughput in this experiment comes to 6,455.0 Kops/s when the numbers of the user threads and the background flush threads are both six.

Exp#6 (Impact of sub-MemTable size): We further study the impact of the sub-MemTable size. We fix the size of the sub-MemTable pool to 12 MB and vary the size of the sub-MemTable from 0.25 MB (i.e., with 48 sub-MemTables) to 2 MB (i.e., with six sub-MemTables). We then measure the throughput of the random reads and the random writes. We launch 12 user threads and 4 background flush threads. Figure 15 shows the results.



(a) Random reads. (b) Random writes.

Figure 15: Exp#6 (Impact of sub-MemTable size).



(a) Random reads. (b) Random writes.

Figure 16: Exp#7 (Impact of sub-MemTable pool size).

Figure 15(a) shows that the read throughput of CacheKV increases with the size of the sub-MemTable. The reason is that a read operation has to scan all the sub-skiplists to fetch the freshest value. Hence, given a sub-MemTable pool with the constant size, fewer sub-skiplists are scanned when the sub-MemTable becomes larger, reducing the search complexity.

Figure 15(b) further indicates that with the increase of the sub-MemTable size, the write throughput of CacheKV first increases (from 2,949.12 Kops/s to 4,180.5 Kops/s) and then declines (from 4,180.5 Kops/s to 3,669.8 Kops/s). This is because when the sub-MemTable is small (e.g., 0.25 MB), it is easily filled up by the newly written KV pairs, making the write performance bottlenecked by the background flush. On the other hand, when its size becomes too large (e.g., 2 MB), only a few sub-MemTables can be supplied at this time, which in turn restricts the performance improvement of the write operations. Based on this experiment, we suggest configuring the size of the sub-MemTable to 1 MB when deploying CacheKV in the write-dominated scenarios.

Exp#7 (Impact of sub-MemTable pool size): We finally assess the impact of the sub-MemTable pool size. We set the sub-MemTable size as 1 MB and vary the pool size from 3 MB (with three sub-MemTables) to 30 MB (with 30 sub-MemTables). We use the same configurations as in Exp#6. Figure 16 depicts the throughput of the random reads and the random writes. We have two findings.

First, when the sub-MemTable pool becomes larger, the read throughput of CacheKV gradually declines (Figure 16(a)). The reason still lies in the increased sub-skiplists to go through when there are more sub-MemTables used in CacheKV.

Second, the write throughput of CacheKV increases with the size of the pool size (Figure 16(b)). This is because when the pool size increases, more sub-MemTables can be used to accommodate the KV pairs in parallel. The improvement becomes marginal especially when the pool size exceeds 6 MB, since the background flush becomes the performance bottleneck. To summarize, we suggest allocating six sub-MemTables to balance the read and write throughput.

V. Related Work

LSM-tree-based KV stores: A massive number of LSM-tree-based KV stores [2], [3], [7], [14]–[20], [43] propose to aggregate random writes into sequential writes for improving the write performance of block storage. However, they require an additional on-disk log for data durability.

The release of the PMem also attracts a number of studies [15], [32]–[36] to build efficient KV stores with the PMem. Most of the time, they simply treat the PMem as an alternative of DRAM with persistence guarantee. NoveLSM [32] maintains large mutable MemTables in PMem, such that the in-place updates can be realized without incurring additional expensive compactions. SLM-DB [33] maintains a B+-tree for accelerating read performance and persists the inserted key-value pairs in the PMem buffer for achieving high write throughput. Instead of entirely replacing traditional DRAM with the PMem, MatrixKV [15] organizes the L_0 level of LSM-trees in PMem to throttle the compaction traffic. As a comparison, CacheKV redesigns the high-concurrent MemTable to fully utilize the persistent CPU caches.

Some studies also leverage the write unit size of the Optane PMem in system designs. FlatStore [38] decouples a KV store into a volatile index (kept in DRAM) and a persistent data log (kept in PMem) to deliver high-throughput performance. ChameleonDB [37] employs LSM-tree to aggregate writes and uses an in-DRAM hash table to enable fast reads. While both FlatStore and ChameleonDB still batches KV pairs in DRAM, CacheKV chooses to cope with them with the persistent CPU cache, thereby avoiding additional consistency overhead.

Studies on eADR-enable Optane PMem: Gugnani et al. [52] present a performance characterization study to categorize the idiosyncratic behavior of ADR- and eADR-enable PMem systems. Zardoshti et al. [65] compare the performance of five persistent transactional memory applications on ADR and eADR persistent domains. PACTree [66] propose *packed asynchronous concurrency* (PAC) guidelines on their findings of the PMem hardware and system software for designing high-performance persistent index structures. NBTree [67] is a lock-free PM-friendly B+-Tree to deliver high scalability with low PMem overhead. Different from previous studies, CacheKV constructs high-performance KV stores based on the characteristics of the eADR-enabled Optane PMem.

VI. Conclusion

We present CacheKV, the first KV store running atop the Optane PMem with persistent CPU caches. CacheKV allocates a sub-MemTable pool in the CPU caches to dynamically assign a dedicated sub-MemTable to each CPU core, so as to parallelize the concurrent writes. It then lazily updates the sub-skiplists and flushes the full sub-ImmMemTables to the PMem for further reducing the write latency. CacheKV finally compacts the sub-skiplists for improving the read efficiency. Extensive testbed experiments show the efficiency of CacheKV under real-world benchmarks.

REFERENCES

[1] 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.

[2] "LevelDB," https://github.com/google/leveldb, 2021.

[3] "RocksDB," https://github.com/facebook/rocksdb, 2021.

[4] R. Sumbaly, J. Kreps, L. Gao, A. Feinberg, C. Soman, and S. Shah, "Serving Large-Scale Batch Computed Data with Project Voldemort." in *Proc. of USENIX FAST*, 2012.

[5] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li *et al.*, "TAO: Facebook's Distributed Data Store for the Social Graph," in *Proc. of USENIX ATC*, 2013.

[6] N. Elyasi, C. Choi, and A. Sivasubramaniam, "Large-Scale Graph Processing on Emerging Storage Devices," in *Proc. of USENIX FAST*, 2019.

[7] 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.

[8] J. Shi, Y. Yao, R. Chen, H. Chen, and F. Li, "Fast and Concurrent RDF Queries with RDMA-based Distributed Graph Exploration," in *Proc. of USENIX FAST*, 2016.

[9] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "NetCache: Balancing Key-Value Stores with Fast In-Network Caching," in *Proc. of ACM SOSP*, 2017.

[10] Y. Zhang, Z. Liu, R. Wang, T. Yang, J. Li, R. Miao, P. Liu, R. Zhang, and J. Jiang, "CocoSketch: High-Performance Sketch-based Measurement over Arbitrary Partial Key Query," in *Proc. of ACM SIGCOMM*, 2021.

[11] "Redis," https://redis.io/, 2021.

[12] J. Chen, L. Chen, S. Wang, G. Zhu, Y. Sun, H. Liu, and F. Li, "HotRing: A Hotspot-Aware In-Memory Key-Value Store," in *Proc. of USENIX FAST*, 2020.

[13] B. Fan, D. G. Andersen, and M. Kaminsky, "MemC3: Compact and Concurrent Memcache with Dumber Caching and Smarter Hashing," in *Proc. of USENIX NSDI*, 2013.

[14] H. Chen, C. Ruan, C. Li, X. Ma, and Y. Xu, "SpanDB: A Fast, Cost-Effective LSM-tree Based KV Store on Hybrid Storage," in *Proc. of USENIX FAST*, 2021.

[15] 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 *Proc. of USENIX ATC*, 2020.

[16] 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 *Proc. of USENIX ATC*, 2021.

[17] Y. Li, C. Tian, F. Guo, C. Li, and Y. Xu, "Elasticbf: Elastic Bloom Filter with Hotness Awareness for Boosting Read Performance in Large Key-Value Stores," in *Proc. of USENIX ATC*, 2019.

[18] H. H. Chan, C.-J. M. Liang, Y. Li, W. He, P. P. C. Lee, L. Zhu, Y. Dong, Y. Xu, Y. Xu, J. Jiang *et al.*, "HashKV: Enabling Efficient Updates in KV Storage via Hashing," in *Proc. of USENIX ATC*, 2018.

[19] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham, "PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees," in *Proc. of ACM SOSP*, 2017.

[20] S. Sarkar, T. I. Papon, D. Staratzis, and M. Athanassoulis, "Lethe: A Tunable Delete-Aware LSM Engine," in *Proc. of ACM SIGMOD*, 2020.

[21] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (lsm-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[22] I. Absalyamov, M. J. Carey, and V. J. Tsotras, "Lightweight cardinality estimation in lsm-based systems," in *Proc. of ACM SIGMOD*, 2018.

[23] Z. Lu, Q. Cao, H. Jiang, S. Wang, and Y. Dong, "P2KVS: A Portable 2-Dimensional Parallelizing Framework to Improve Scalability of Key-Value Stores on SSDs," in *Proc. of ACM EuroSys*, 2022.

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

[25] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting Phase Change Memory as a Scalable Dram Alternative," in *Proc. of ISCA*, 2009.

[26] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung, and C. H. Lam, "Phase-Change Random Access Memory: A Scalable Technology," *IBM Journal of Research and Development*, vol. 52, no. 4.5, pp. 465–479, 2008.

[27] S. Kargar and F. Nawab, "Extending the lifetime of NVM: challenges and opportunities," *Proceedings of the VLDB Endowment*, vol. 14, pp. 3194–3197, 2021.

[28] I. Baek, M. Lee, S. Seo, M. Lee, D. Seo, D.-S. Suh, J. Park, S. Park, H. Kim, I. Yoo *et al.*, "Highly Scalable Non-volatile Resistive Memory using Simple Binary Oxide Driven by Asymmetric Unipolar Voltage Pulses," in *Proc. of IEEE IEDM*, 2004.

[29] 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.

[30] F. T. Hady, A. 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.

[31] "Intel® Optane™ Persistent Memory," https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html, 2021.

[32] S. Kannan, N. Bhat, A. Gavrilovska, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Redesigning LSMs for Non-Volatile Memory with NoveLSM," in *Proc. of USENIX ATC*, 2018.

[33] O. Kaiyrakhmet, S. Lee, B. Nam, S. H. Noh, and Y.-r. Choi, "SLM-DB: Single-Level Key-Value Store with Persistent Memory," in *Proc. of USENIX FAST*, 2019.

[34] F. Xia, D. Jiang, J. Xiong, and N. Sun, "HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems," in *Proc. of USENIX ATC*, 2017.

[35] Y. Huang, M. Pavlovic, V. Marathe, M. Seltzer, T. Harris, and S. Byan, "Closing the Performance Gap between Volatile and Persistent Key-Value Stores using Cross-Referencing Logs," in *Proc. of USENIX ATC*, 2018.

[36] L. Benson, H. Makait, and T. Rabl, "Viper: An Efficient Hybrid PMem-DRAM Key-Value Store," *Proceedings of the VLDB Endowment*, vol. 14, no. 9, pp. 1544–1556, 2021.

[37] W. Zhang, X. Zhao, S. Jiang, and H. Jiang, "ChameleonDB: A Key-Value Store for Optane Persistent Memory," in *Proc. of ACM EuroSys*, 2021.

[38] Y. Chen, Y. Lu, F. Yang, Q. Wang, Y. Wang, and J. Shu, "FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory," in *Proc. of ACM ALSPLOS*, 2020.

[39] S. Lee, A. Lerner, A. Ryser, K. Park, C. Jeon, J. Park, Y. H. Song, and P. Cudré-Mauroux, "X-SSD: A storage system with native support for database logging and replication," in *Proc. of ACM SIGMOD*, 2022.

[40] H. Liu, L. Huang, Y. Zhu, and Y. Shen, "LibreKV: A Persistent In-Memory Key-Value Store," *IEEE Transactions on Emerging Topics in Computing*, vol. 8, no. 4, pp. 916–927, 2017.

[41] L. Lersch, I. Schreter, I. Oukid, and W. Lehner, "Enabling Low Tail Latency on Multicore Key-Value Stores," *Proceedings of the VLDB Endowment*, vol. 13, no. 7, pp. 1091–1104, 2020.

[42] "Third Generation Intel® Xeon® Processor Scalable Family Technical Overview," https://www.intel.com/content/www/us/en/developer/articles/technical/intel-xeon-processor-scalable-family-overview.html, 2021.

[43] W. Zhong, C. Chen, X. Wu, and S. Jiang, "REMIX: Efficient Range Query for LSM-trees," in *Proc. of USENIX FAST*, 2021.

[44] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks using Write-Ahead Logging," *ACM Transactions on Database Systems*, vol. 17, no. 1, pp. 94–162, 1992.

[45] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An Empirical Guide to the Behavior and Use of Scalable Persistent Memory," in *Proc. of USENIX FAST*, 2020.

[46] P. Zuo, Y. Hua, and J. Wu, "Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory," in *Proc. of USENIX OSDI*, 2018.

[47] "eADR: New Opportunities for Persistent Memory Applications," https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html, 2021.

[48] J. Liu, S. Chen, and L. Wang, "Lb+ trees: Optimizing persistent index performance on 3dxpoint memory," *Proceedings of the VLDB Endowment*, vol. 13, no. 7, pp. 1078–1090, 2020.

[49] "Intel® Xeon® Processor Scalable Family Technical Overview," https://www.intel.com/content/www/us/en/developer/articles/technical/xeon-processor-scalable-family-technical-overview.html, 2021.

[50] "Intel® PMWatch," https://github.com/intel/intel-pmwatch, 2021.
[51] S. Han, D. Jiang, and J. Xiong, "LightKV: A Cross Media Key Value Store with Persistent Memory to Cut Long Tail Latency," in *Proc. of IEEE MSST*, 2020.
[52] S. Gugnani, A. Kashyap, and X. Lu, "Understanding the Idiosyncrasies of Real Persistent Memory," *Proceedings of the VLDB Endowment*, vol. 14, no. 4, pp. 626–639, 2020.
[53] L. Xiang, X. Zhao, J. Rao, S. Jiang, and H. Jiang, "Characterizing the Performance of Intel Optane Persistent Memory: a Close Look at its on-DIMM Buffering," in *Proc. of ACM EuroSys*, 2022.
[54] Z. Cao, S. Dong, S. Vemuri, and D. H. Du, "Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook," in *Proc. of USENIX FAST*, 2020.
[55] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *Proc. of ACM SIGMETRICS/Performance*, 2012.
[56] P. J. Shetty, R. P. Spillane, R. R. Malpani, B. Andrews, J. Seyster, and E. Zadok, "Building Workload-Independent Storage with VT-Trees," in *Proc. of USENIX FAST*, 2013.
[57] "User Interface for Resource Control feature," https://www.kernel.org/doc/html/v5.9/x86/resctrl_ui.html#cache-pseudo-locking, 2016.
[58] "Introduction to Cache Allocation Technology in the Intel® Xeon® Processor E5 v4 Family," https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-cache-allocation-technology.html?wapkw=intel%20cat, 2016.
[59] Y. Zhang and S. Swanson, "A study of application performance with non-volatile main memory," in *Proc. of IEEE MSST*, 2015.
[60] D. S. Rao, S. Kumar, A. S. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Proc. of ACM EuroSys*, 2014.
[61] L. Baptiste, B. Oana, G. Karan, and Z. Willy, "Kvell: The design and implementation of a fast persistent key-value store," in *Proc. of ACM SOSP*, 2019.
[62] R. Escriva, B. Wong, and E. G. Sirer, "Warp: Lightweight multi-key transactions for key-value stores," *arXiv preprint arXiv:1509.07815*, 2015.
[63] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *Proc. of ACM SOCC*, 2010.
[64] J. Yang, Y. Yue, and R. Vinayak, "Segcache: A Memory-Efficient and Scalable In-Memory Key-Value Cache for Small Objects," in *Proc. of USENIX NSDI*, 2021.
[65] P. Zardoshti, M. Spear, A. Vosoughi, and G. Swart, "Understanding and Improving Persistent Transactions on Optane™ DC Memory," in *Proc. of IEEE IPDPS*, 2020.
[66] W.-H. Kim, R. M. Krishnan, X. Fu, S. Kashyap, and C. Min, "PACTree: A High Performance Persistent Range Index Using PAC Guidelines," in *Proc. of ACM SOSP*, 2021.
[67] B. Zhang, S. Zheng, Z. Qi, and L. Huang, "NBTree: a Lock-free PM-friendly Persistent B+-Tree for eADR-enabled PM Systems," *Proceedings of the VLDB Endowment*, 2022.