ChameleonDB: a Key-value Store for Optane Persistent Memory

Wenhui Zhang
wenhui.zhang@uta.edu
University of Texas at Arlington
Arlington, Texas

Xingsheng Zhao
xingsheng.zhao@mavs.uta.edu
University of Texas at Arlington
Arlington, Texas

Song Jiang
song.jiang@uta.edu
University of Texas at Arlington
Arlington, Texas

Hong Jiang
hong.jiang@uta.edu
University of Texas at Arlington
Arlington, Texas

Abstract

The emergence of Intel's Optane DC persistent memory (Optane Pmem) draws much interest in building persistent key-value (KV) stores to take advantage of its high throughput and low latency. A major challenge in the efforts stems from the fact that Optane Pmem is essentially a hybrid storage device with two distinct properties. On one hand, it is a high-speed byte-addressable device similar to DRAM. On the other hand, the write to the Optane media is conducted at the unit of 256 bytes, much like a block storage device. Existing KV store designs for persistent memory do not take into account of the latter property, leading to high write amplification and constraining both write and read throughput. In the meantime, a direct re-use of a KV store design intended for block devices, such as LSM-based ones, would cause much higher read latency due to the former property.

In this paper, we propose ChameleonDB, a KV store design specifically for this important hybrid memory/storage device by considering and exploiting these two properties in one design. It uses LSM tree structure to efficiently admit writes with low write amplification. It uses an in-DRAM hash table to bypass LSM-tree's multiple levels for fast reads. In the meantime, ChameleonDB may choose to opportunistically maintain the LSM multi-level structure in the background to achieve short recovery time after a system crash. ChameleonDB's hybrid structure is designed to be able to absorb sudden bursts of a write workload, which helps avoid long-tail read latency.

Our experiment results show that ChameleonDB improves write throughput by 3.3× and reduces read latency by around 60% compared with a legacy LSM-tree based KV store design. ChameleonDB provides performance competitive even with KV stores using fully in-DRAM index by using much less DRAM space. Compared with CCEH, a persistent hash table design, ChameleonDB provides 6.4× higher write throughput.

CCS Concepts • **Information systems → Information storage systems**; **Key-value stores**.

Keywords key-value store, persistent-memory, Optane DC

ACM Reference Format:
Wenhui Zhang, Xingsheng Zhao, Song Jiang, and Hong Jiang. 2021. ChameleonDB: a Key-value Store for Optane Persistent Memory. In *Sixteenth European Conference on Computer Systems (EuroSys '21), April 26–28, 2021, Online, United Kingdom.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3447786.3456237

EuroSys '21, April 26–28, 2021, Online, United Kingdom

ACM ISBN 978-1-4503-8334-9/21/04…$15.00
https://doi.org/10.1145/3447786.3456237

1 Introduction

Intel Optane DC persistent memory, or Optane Pmem for short, is the first commercially available persistent byte-addressable memory [37]. Compared with traditional storage devices, it has higher write throughput, much lower read latency, and can be accessed by processors directly. With the emergence of Optane Pmem, it becomes possible to build a key-value (KV) store with high write throughput, low read latency, low DRAM footprint, and rapid recovery and restart after a system crash. However, existing KV store designs face three key challenges in their exploitation of opportunities enabled by the Optane Pmem to build such a KV store.

1.1 Challenge 1: Optane Pmem is a Block Device

Researchers had proposed a number of KV store designs for persistent memory before the release of the Optane Pmem [6] in 2019. These designs are based on the assumption that persistent memory is just a "slower, persistent DRAM" [37]. Accordingly, such a design usually builds a persistent hash table or a persistent tree to index KV items in a storage log.

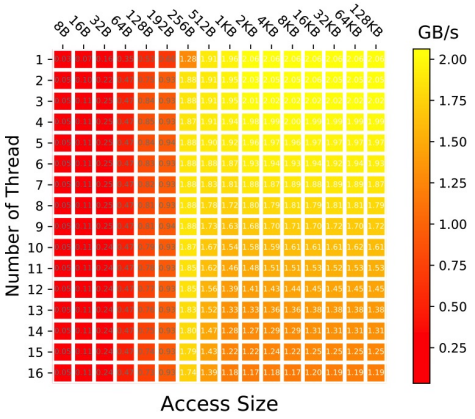


Figure 1. Random write performance on one Optane Pmem using different access size. For writes, we use *ntstore* followed with a *sfence* instruction to ensure data persistency. Performance degradation with a larger number of threads and larger access sizes is due to contention in the iMC (integrated Memory Controller).

While new KV items are written to the log in batches according to their arrival order, corresponding updates on the index are individually made at (usually) non-contiguous memory locations (determined by hash functions or tree structures). Examples include Level hashing [40] and CCEH [28] that use hash tables, and wB$^+$-Trees [1] and FAST&FAIR [15] that use tree-structured indexes on the persistent memory with proper schemes to guarantee data consistency. Unfortunately, the aforementioned assumption on persistent memory is not consistent with findings from studies on performance characteristics of the first commercial persistent memory – the Intel Optane Pmem.

It has been reported that Optane Pmem has a write unit size of 256B [37]. To understand the implication of this performance characteristic, we write data of a particular size to randomly selected addresses that are aligned with the 256B unit size. In the experiment, we vary the write size from 8B to 128KB and use different numbers of threads so that the memory's peak bandwidth can be reached. Details of the system configuration are described in Section 3. As shown in Figure 1, when the write size is much smaller than 256B (the unit size), the write throughput is much lower than that with large writes (256B or larger). More interestingly, throughput of 256B-writes roughly doubles that of the 128B-writes, which further doubles that of the 64B-writes. This strongly confirms the existence of the 256B access unit in the device. Any non-contiguous writes of data smaller than the unit size requires a read-modify-write operation to generate a 256B write to the memory media, leading to write amplification and reduced effective memory bandwidth.

This property causes KV-store designs assuming Optane Pmem to be a DRAM with persistence to suffer from performance loss on writes [3], in principle similar to that experienced with small writes to other block devices such as hard disks and SSDs [33]. In particular, in the KV stores that employ persistent hash table or tree-based indexes, each update to an index structure is usually much smaller (e.g., 16 bytes) than Optane's 256B unit size during key insertion, rehashing, or tree re-balancing, leading to large write amplification (e.g, 16). Failing to adequately take the device's write unit into account, these designs are unable to provide high write performance.

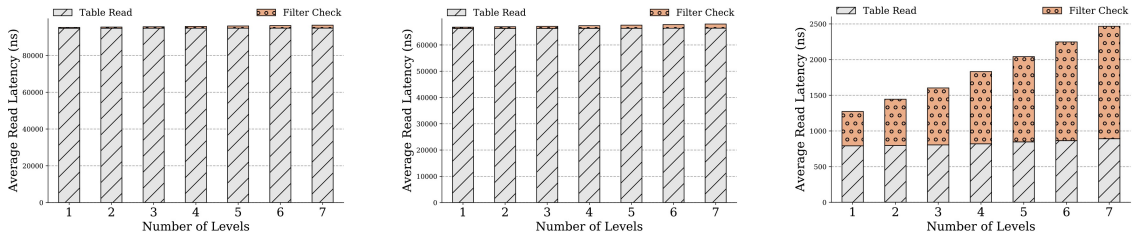
1.2 Challenge 2: Optane Pmem is of High Speed

Major efforts have been made to address the issue of small writes in a KV store on block storage devices, such as hard disks and SSDs. Among them, LSM (Log-Structured Merge)-tree based designs, such as LevelDB [13], RocksDB [11], Cassandra [23], LSM-trie [34], and PebblesDB [31], are examples in the research community and industry deployments. They aggregate recent updates and batch-write them to the disk sequentially in an order determined either by key comparison or by a hash function. As it is too expensive to maintain one big sorted list, multiple and exponentially longer sorted lists are maintained. Each list is in its dedicated level, starting at level 0 ($L0$) for the shortest one. Each lower level (e.g., $L(k{+}1)$) has a capacity r times larger than its immediate higher level ($L(k)$), where $k \geq 0$. The r value varies in different KV stores. For example, in LevelDB and RocksDB $r = 10$ [11, 13], while in LSM-trie $r = 8$ [34]. After KV items are written into the store, they are initially in $L0$, and move through the hierarchy level by level in a sequence of compaction operations until reaching the last level.

There are two major compaction schemes for the LSM-tree structure with different implications on write amplification and read performance, which are leveling [11, 13, 25] and size-tiering [4, 11, 23, 27, 31, 34]. For the leveling compaction, KV items in two adjacent levels are merge-sorted and then re-inserted into the lower level. As the number of keys in the lower level can be multiple times more than that in the upper level, write amplification for each of such compactions can be as large as 10 (in LevelDB as an example)[1]. In the size-tiering compaction, each level consists of multiple sub-levels with overlapping key range. Key merge-sort operation is conducted among the sub-levels and the result KV items are written as a new sub-level of its lower level. In this way, write amplification in a compaction is always 1 [1]. While size-tiering can significantly reduce extra writes, it also significantly increases the number of (sub-)levels and potentially increases cost to search a key in the store.

While an LSM-based design seems to be a good candidate for deployment at the Optane Pmem with its built-in multi-level structure designed for block devices [21, 38], it is unfortunately incompatible with Optane's high read performance. With its multi-level design, reading a key in an LSM tree requires searching a sequence of levels, starting

[1]For the total write amplification, we should multiply this number with the number of levels.



(a) SATA SSD (b) PCIe SSD (c) Optane Pmem

Figure 2. Average read latency of a multi-level hash table design on various devices.

from $L0$, until the key is found or the last level is reached. Aiming to have only one disk read per key search, it maintains in-DRAM Bloom filters for each block of KV items to know if a key is likely to exist in the block at a level before actually reading on-disk data from the level. Compared to the millisecond-level disk access time, the nanosecond-level cost of operations on the filters is negligible. As long as only one disk read actually occurs, such a design achieves the best possible read latency on KV items on the disk. However, the situation becomes vastly different when the storage device is the Optane Pmem, whose read latency itself is at the nanosecond level and is only about 3× of DRAM's read latency [37].

To understand the implication of Optane's high-speed access, we choose to build a hash-based KV store, LSM-trie [34], with 7 levels on an SSD connected with a SATA interface, a second one on an SSD with a PCIe interface, and a third one on the Optane Pmem. We read keys at different levels and report their read latency in Figure 2. As shown, the time to read items from tables (denoted as "Table Read" in the figures) is highly consistent no matter which level the keys reside as only one disk (or Pmem) read is required. As shown by Figures 2(a) and 2(b), the time spent on the filters (denoted as "Filter Check" in the figures) occupies a tiny portion when the store is running on the SSDs. Therefore, with the help of Bloom filters on an on-disk KV store, using the multi-level structure doesn't compromise read performance. However, as Figure 2(c) shows, when the Optane Pmem is used, the time spent on the filters becomes significant (relative to the Pmem's read time). It keeps increasing with KV items at lower levels and finally becomes unacceptable. This observation indicates that multi-level structure becomes a major barrier to achieving consistently low read latency. Meanwhile, the very same structure is also essential for enabling batched writes to accommodate block devices.

1.3 Challenge 3: Optane Pmem is Non-volatile

To avoid the two aforementioned challenges, researchers have proposed to move the index structure to the DRAM while using the Pmem only for storing KV items as a storage log [2, 22, 24]. As KV items are batch-written to the log without write amplification, and all reads and updates of the index take place in the DRAM, such a design provides high write throughput and low read latency.

However, leaving the entire or a majority of the index in the volatile memory cancels an essential benefit of Optane Pmem as a persistent memory that promises an instant recovery and restart after an incident such as power failure or system crash. For a KV store storing multi-billion KV items, the index in the DRAM can grow as large as over 100GB, which is a considerable demand on limited DRAM space shared by many systems and application functionalities [10]. Once the index is lost with an unexpected shutdown, rebuilding such a large index from the storage log, which may take an unacceptably long time, is required to resume the store's service. In the meantime, a speedy recovery and restart is important, especially in a virtualized environment where a KV store service is hosted in virtual machines or containers whose own launch time can be as little as a few seconds or even in the sub-second level. It is noted that the idea of periodically saving the latest updates on the index to the Optane Pmem and making the index on the Pmem organized and ready to use is actually the one motivating the LSM-tree-based design, whose drawback has been elaborated in Subsection 1.2.

1.4 Our Solution

While existing KV store designs cannot simultaneously achieve the multiple objectives expected on an Optane Pmem (high write throughput, low read latency, well-bounded read tail latency for highly dynamic workloads, small DRAM footprint, and speedy recovery and restart), we propose a KV store design, named ChameleonDB, that can achieve all of the objectives in one system. To illustrate ChameleonDB's strengths, in Figure 3 we compare it with a store with its hash-based index in the Pmem, named Pmem-Hash, a store with its LSM-tree-based index in the Pmem, named Pmem-LSM, and a store with its hash-based index in the DRAM, named Dram-Hash on four performance measures, namely, write amplification, which is highly correlated to write throughput, read latency, memory footprint size, and recovery time.

In summary, this paper makes three major contributions:

1. We analyze the shortcomings of existing KV store designs on the Optane Pmem, demonstrating that none

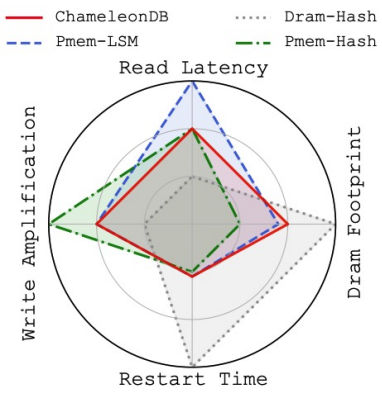


Figure 3. Comparison of KV store designs on the Optane Pmem in four measures, where smaller values indicate more desired readings in each of the four measures. Among them, Pmem-LSM corresponds to legacy LSM-tree-like KV store design with Bloom filters. It has long read latency. Dram-Hash corresponds to the in-DRAM index design with in-Pmem log. It has a large DRAM footprint and a long restart time. Pmem-Hash corresponds to the persistent hash table design. It has large write amplification, leading to low write throughput. In contrast, ChameleonDB receives better result in every of the four measures. At each measure, all measurements are normalized to the one of the largest (worst) value.

of them can achieve high write throughput, low read latency, low DRAM footprint, and fast restart at the same time. In particular, we reveal the dilemma of deploying an LSM-based KV store on the Optane Pmem, which has not yet been discussed in the open literature to the best of our knowledge.

2. We propose ChameleonDB, a novel KV store designed for the Optane Pmem. To some extent it is a hybrid design – it leverages respective strength of one of the stores (Pmem-Hash, Pmem-LSM, and Dram-Hash) to address the weakness of the others. Specifically, it uses a multi-level structure to efficiently persist updates on the index. It uses an in-DRAM hash table to speed up read of a small set of recently updated index. And it uses an in-Pmem hash table to retrieve a majority of the keys in the store. It also has an operation mode for opportunistically curtailing long read tail latency.
3. We implement ChameleonDB and evaluate it in comparison with other state-of-the-art designs representing Pmem-Hash, Pmem-LSM, and Dram-Hash. Our experiment results show that ChameleonDB successfully achieves the aforementioned list of objectives simultaneously, outperforming each of the other stores on one or more of the objectives, as shown by Figure 3.

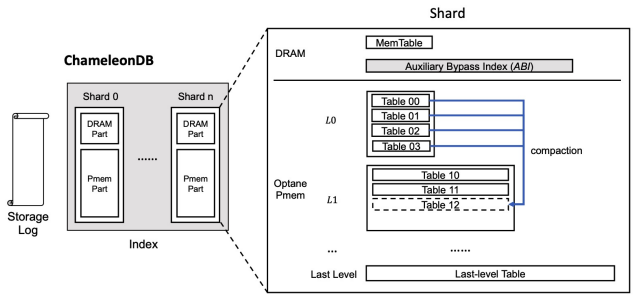


Figure 4. Structure of ChameleonDB.

2 The Design of ChameleonDB

ChameleonDB is a KV store where values are stored in a storage log, while keys (or their hash values) and the locations of their corresponding values in the log are stored in a persistent index, as illustrated in Figure 4. KV items are written to the storage log in batches according to their arrival order. The persistent index is a highly parallel structure with multiple shards, in which each shard has its own multi-level structure with its compaction operations. Keys are distributed evenly across these shards according to their hash values.

2.1 A Multi-shard Structure

The index of ChameleonDB is organized as a multi-shard structure, where each shard covers an equal range of hashed-key space. A shard is a multi-level LSM-like structure, as shown in Figure 4. Each level has multiple sub-levels, named tables, each of which is organized as a fixed-size hash table with linear probing as key collision resolution. Like other LSM-tree based KV stores, each shard has an in-DRAM MemTable to aggregate KV items. When the MemTable is full (i.e., its load factor exceeds a threshold), it is flushed to the Optane Pmem as a persistent and immutable table in the $L0$ level. The maximum number of tables that can be held in each level is specified by a between-level ratio, r, except for the last level that contains only one table. For the instance depicted in Figure 4, the ratio r is 4. Thus, the $L0$ level is full after four MemTables have been flushed to the Pmem. Compaction will be triggered when a level is full.

As each of the two compaction schemes, leveling and size-tiering, has its advantage and disadvantage, ChameleonDB uses both compaction schemes at different LSM levels to provide low write amplification and low read latency. In each shard, the size-tiering compaction is used to compact tables in upper levels (all levels in the Pmem except the last one), while the leveling compaction is used to compact tables to the last level. This hybrid compaction scheme is also adopted in [8] as Lazy Leveling, which strikes a balance between write amplification and read latency, and performs better than using either of the two schemes alone.

A compaction in the LSM-tree based KV stores usually involves only two adjacent levels. For the instance depicted in

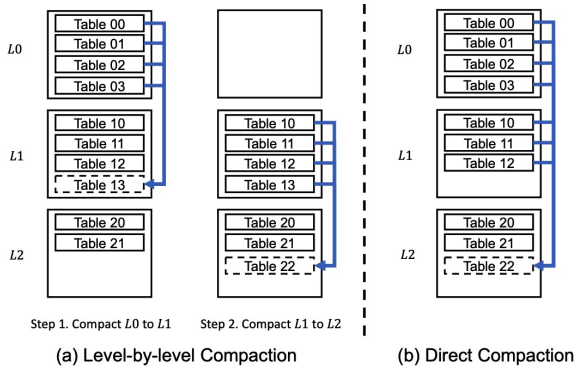


Figure 5. ChameleonDB uses Direct Compaction scheme to reduce compaction overhead.

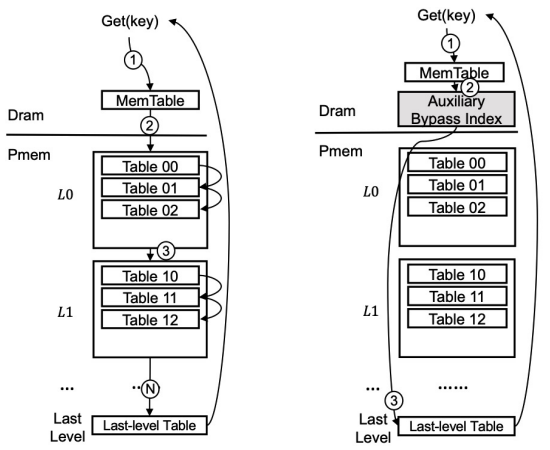
Figure 5(a), $L0$ is full and triggers a compaction to $L1$, which makes $L1$ full and triggers another compaction from $L1$ to $L2$. In ChameleonDB, we introduce a Direct Compaction scheme that can reduce compaction overhead by allowing a compaction to involve multiple levels. To complete the sequence of compactions shown in Figure 5(a), with Direct Compaction ChameleonDB triggers only one compaction covering $L0$ and $L1$ to $L2$, as shown in Figure 5(b). Likewise, last level compaction takes place when $L0$ is full and each of the upper levels has $(r-1)$ tables, where r is the maximum number of tables that can be held in each level. After a last level compaction in a shard, all tables in the upper levels of the shard are cleared as their items are all moved to the last level table.

As the size of each table (including MemTable in the DRAM) in a shard is (often much) larger than 256B (the access unit size of the Optane Pmem), and is aligned to 256B, flushing/compacting tables can fully utilize the write bandwidth of Optane Pmem, which addresses Challenge 1 detailed in Subsection 1.1. Furthermore, as ChameleonDB persists changes on the LSM structure after each flushing of MemTable, a restart only needs to recover MemTables that have not been persisted, which addresses Challenge 3 detailed in Subsection 1.3.

2.2 The Auxiliary Bypass Index (*ABI*) in a Shard

As each shard is a multi-level structure, a get operation[2] may have long latency as it needs to check the tables level-by-level until it finds the target key or reaches the last level, as illustrated by Figure 6(a). Because ChameleonDB uses size-tiering compaction in the upper levels for reduced write amplification, the number of (sub-)levels has been significantly increased. As we have indicated in Section 1.2, Bloom filter, which is usually used in an LSM tree structure, does not provide an efficient solution to this long get latency issue because checking filters accounts for 50% or more of Optane Pmem read latency. That's why we need a new solution to

[2]In this paper, the terms of *get*/*put* and *read*/*write* regarding KV stores have the same meaning and are used interchangeably.



(a) The get operation without an *ABI* needs to check the tables one-by-one, leading to long get latency.

(b) The get operation with an *ABI* checks at most three tables, leading to short and more consistent get latency.

Figure 6. ChameleonDB uses Auxiliary Bypass Index to provide short and stable get latency.

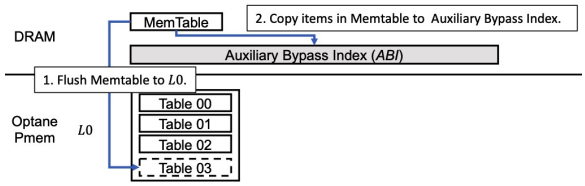


Figure 7. Flush operation in a shard.

reduce the get latency, which is ChameleonDB's *Auxiliary Bypass Index* (***ABI*** for short).

Each shard in ChameleonDB has its own *ABI*, which is an in-DRAM hash table that indexes the keys in all upper levels. By using *ABI*, at most three tables need to be checked when searching keys in ChameleonDB, which are the MemTable, *ABI*, and the last level table, as depicted in Figure 6(b).

ABI contains and only contains KV items that exist in the upper levels. When keys are persisted to $L0$, they should also be inserted to *ABI*. After items have been merged to the last level, they should be removed from *ABI*. To make this happen, ChameleonDB adds items to *ABI* during flushing MemTable to $L0$, as shown in Figure 7. It removes all items in *ABI* after the last level compaction as all tables in the upper levels are going to be cleared after a last level compaction (detailed in Subsection 2.1). PinK [16] also tries to pin the upper levels in the DRAM to reduce the get latency. However, it pins the upper levels as a multi-level LSM structure while *ABI* organizes keys in the upper levels as a hash table with O(1) access time in the DRAM.

In addition to reducing the get latency, *ABI* can also help accelerate the last level compaction. *ABI* contains all the items in the upper levels in a shard. A Direct Compaction in ChameleonDB compacts all upper level tables to the last

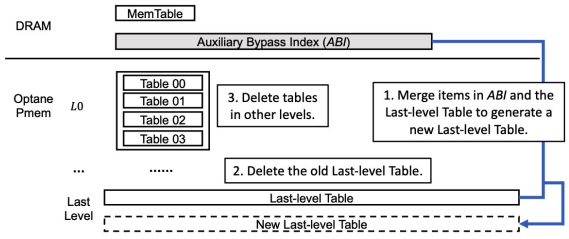


Figure 8. Efficient last level compaction with *ABI*.

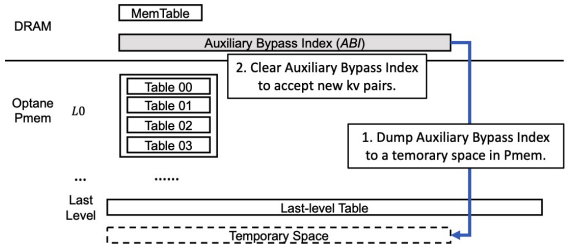


Figure 9. The Get-Protect Mode to suspend merging *ABI* with last level during the read-intensive periods.

level table (detailed in Subsection 2.1). Instead of reading and merging all persistent tables in the upper levels to the last level, ChameleonDB merges items that have been in the DRAM into the last level table, as shown in Figure 8, so as to reduce the overhead of last level compaction. By using *ABI*, we avoid checking multiple levels in ChameleonDB so as to address Challenge 2 specified in Subsection 1.2.

2.3 Trade Restart Time for the Put Performance

The multi-level structure of shards in ChameleonDB helps to reduce the restart time as only MemTable needs to be recovered upon restart, where MemTable contains a very small portion of the total items in the store (e.g., 1/256 for a 4-level shard with a between-level ratio of $r = 4$). However, maintaining the multi-level LSM structure consumes read bandwidth, CPU cycles, and write bandwidth as compactions need to be constantly conducted along with put requests, which decreases the put performance. For traditional LSM-tree based KV store designs like LevelDB [13] and RocksDB [11], the multi-level structure has to be always maintained. Otherwise, get latency would be seriously compromised (as more and more MemTables are flushed to $L0$ without being compacted producing numerous levels) or the DRAM footprint would increase rapidly (as MemTable grows bigger and bigger to hold more KV items). In contrast, the structure of ChameleonDB, whose upper levels are used only for fast recovering, offers an opportunity to not maintain the multi-level structure without compromising get latency and DRAM footprint.

In scenarios where the systems are mostly reliable and the system rarely crashes, ChameleonDB can trade the risk for longer restart time upon a system crash for higher put performance by suspending maintenance of the multi-level structure temporarily without upper level compactions during intensive put workload. This execution choice is named *Write-Intensive Mode* in ChameleonDB.

During Write-Intensive Mode, MemTables are no longer flushed to $L0$ when they are full and, as such, compaction from $L0$ to $L1$ will not be triggered, and likewise, all compactions to other upper levels will not be conducted either. Only when *ABI* is full is the last level compaction triggered to clear *ABI*. If a system crash happens during the Write-Intensive Mode, the restart time can be long as all items in the *ABI* must be recovered from the storage log before the store can resume its service. Our experiment results show that the restart time of ChameleonDB during the Write-Intensive Mode is still much lower than Dram-Hash, whose entire index needs to be recovered upon a restart. Besides, the impact for such a long recovery time of ChameleonDB can be controlled by considering the Write-Intensive Mode as a user option.

2.4 Handling Put Bursts

Tail get latency (the 99th Percentile latency) is usually used to measure the quality of service (QoS) in a storage system. Background compactions have high impacts on the tail get latency as they consume read and write bandwidth of the Optane Pmem. During a put burst period, get latency can increase significantly when background compactions are triggered (check experiment results in Subsection 3.6).

To provide better quality of service, in ChameleonDB we introduce a dynamic *Get-Protect Mode* that monitors the tail get latency and adjust compaction timing accordingly. Specifically, when the tail get latency reaches a threshold, ChameleonDB suspends all upper level compactions (including flushing MemTables to $L0$) and postpones the last level compactions. A side effect of suspending upper level compactions is that the restart time can be longer as we need to recover not only the MemTable but also the *ABI* by scanning the storage log, similar to the side effect of the Write-Intensive Mode. When the *ABI* is full, its items need to be compacted to the last level, assuming the DRAM space is limited and cannot hold a second *ABI*. However, compaction of the items to the last level entails reading existing last level items, merging them with items in the *ABI*, and writing it back to the Optane Pmem. This can be very expensive and may affect tail get latency significantly. Therefore, in the Get-Protect Mode ChameleonDB only dumps the content of *ABI* to the Optane Pmem as a new level without merging it with the last level table, as illustrated by Figure 9. This will increase the number of levels that a get request has to be checked. However, this side effect is modest relative to the cost of last level compaction according to our experiment results. Furthermore, we limit the number of *ABI*s that can be dumped to the Optane Pmem (one by default). The dumped

tables will gradually be merged with the last level table after the put burst subsides. The Get-Protect Mode is canceled when the tail latency subsides below a pre-defined threshold.

2.5 Implementation Details

Randomized Load Factors. Hash tables are usually used as an extendable structure whose size scales along with items being inserted. Once a table is regarded as full, it will be expanded and items in it will be rehashed to new positions. The rehashing is a time-consuming process. Therefore, in a shard of ChameleonDB, we use a fixed-size hash table for keys in the upper levels to avoid rehashing, which can be frequent as the hash table's size is highly variable.

Each table (or a sub-level) in a ChameleonDB's shard is also a hash table. It resolves collision by linear probing [12]. A common way to determine whether such a hash table is full is the number of times it has been probed when inserting a new item. However, using probe count to determine if a table is full can be in conflict with the compaction operation in ChameleonDB. For instance, the items in the four tables in a level may not be able to be inserted to a new table four times larger due to insertion of one item that has a probing number larger than the threshold. On the other hand, allowing unlimited probe times is neither an acceptable choice, as in the worst case a get request may need to scan the whole table, resulting in too-long get latency.

In ChameleonDB, we limit the load factor of MemTable and thus control the load factors of all persistent tables (as all their items are from MemTable). For instance, when we limit the load factor of MemTable to no more than 75%, one in four slots of the table are empty. MemTable is regarded as full when its load factor reaches the predefined threshold. As a scan of a hash table stops when an empty slot is reached, lowering the load factor to increase empty slots in tables can improve get latency at the cost of lower space efficiency and more frequent compactions. Using a unified load factor threshold for all shards will result in compaction bursts as insertions are usually evenly distributed to the shards. During a compaction burst, all the shards are required to do upper level compactions or even worse, last level compactions. The KV store would experience a significant performance degradation period. To mitigate compaction bursts, ChameleonDB uses randomized load factors for the shards so as to stagger the timings of compactions of different shards.

DRAM footprint. A major concern about using the *ABI* to improve get latency is its DRAM footprint. It is known that the level size in each shard increases exponentially. Though the *ABI* contains items in all upper levels, it has only a moderate portion (e.g., $1/r$ for a between-level ratio of r) of the total index. Detail DRAM footprint for ChameleonDB is discussed in Section 3.

Write Amplification. The write amplification (excluding writes to the storage log) in ChameleonDB is related to the number of levels (denoted as l), the between-level ratio r and the load factor (denoted as f). The write amplification of writing a hash table is $1/f$. For instance, a hash table of 1MB size with a load factor of 75% has only 0.75MB user data, so writing the table has a write amplification of $1/0.75$. As ChameleonDB uses size-tiering for middle-level compactions and uses leveling for last level compaction, the write amplification of ChameleonDB is $(l-1+r)/f$.

KV items in the storage log. Each entry in the storage log has the form of $\{key, value\_size, value\}$, where the sizes of the key and $value\_size$ fields are fixed at 8 bytes while the size of $value$ is variable (the $value\_size$ number). Log items are first buffered in the DRAM to form a batch. The batch is then appended to the tail of log when its size reaches a predefined threshold (e.g., 4KB).

3 Evaluation

To observe and understand ChameleonDB's performance behaviors in a real system and know whether it achieves all of its design objectives (high write throughput, low read latency, resilience to impact of request spikes on read tail latency, and speedy recovery and restart), we developed a ChameleonDB prototype on the Intel Optane persistent memory (Pmem).

3.1 The System Setup

Intel Optane Pmem has two operating modes (Memory Mode and App Direct Mode). In the Memory Mode, Optane Pmem is treated as a volatile device and becomes the only memory space visible to programmers with DRAM as its L4 cache. In the App Direct Mode, Optane Pmem is used as a non-volatile storage device. In this mode, it can be either accessed via standard file system APIs as a traditional storage device or accessed directly via load and store instructions [35]. ChameleonDB runs in its App Direct Mode and accesses the memory with load and store instructions through calling functions in the open-sourced libraries of the Persistent Memory Development Kit (PMDK) [30].

All the experiments were conducted on a server with two 8-core Intel Xeon Silver 4215 processors, 64GB DRAM, and two 128GB Optane Pmem. The two Optane Pmem DIMMs are connected to the same socket and operated in an interleaved mode. In our experiments, KV stores are running on the processor with local access to the Optane Pmem for higher access efficiency.

3.2 The KV Stores in Comparison

In addition to ChameleonDB, we include three other representative design strategies, or the KV stores built out of them, in the evaluation. In the KV stores, KV items are stored in a storage log in the Optane Pmem in the order of their arrivals. The difference is on where their index structures are located and how the structures are organized. The KV

Table 1. Configuration of ChameleonDB

Parameter	Value
# of Shards	16384
MemTable Size	8KB per shard (128MB in total)
# of Levels	4 (including last level)
Between-level Ratio	4
Load Factor	randomly from 0.65 to 0.85
Auxiliary Bypass Index Size	512KB per shard (8GB in total)

stores in comparison are Pmem-Hash, which maintains a persistent hash table in the Pmem as an index, Pmem-LSM, which maintains an LSM-based KV stores in the Pmem, and Dram-Hash, which maintains a hash table in DRAM.

For Pmem-LSM, we develop a KV store using hash-keys for the placement of items in the LSM tree. To reveal impact of using Bloom filter on the performance, we include two Pmem-LSM's variants: Pmem-LSM-F with Bloom filters and Pmem-LSM-NF without using Bloom filters. Inspired by the PinK KV store [16] that pins upper levels of an LSM-tree in the DRAM so as to improve read and compaction efficiency, we introduce Pmem-LSM-PinK, which pins all of its LSM levels except the last level in the DRAM, in the experiments for comparison. As construction of Bloom filters is expensive, it may become a performance bottleneck and more than offset the benefit of a larger DRAM, Pmem-LSM-PinK doesn't use Bloom filters, the same as Pmem-LSM-NF. Furthermore, to retain Pmem-LSM's advantage on persistency of the LSM structure for faster recovery, Pmem-LSM-PinK writes each level of KV items to the Pmem once it is produced in a compaction while a copy of the levels stays in the DRAM for faster read. In this way, Pmem-LSM-PinK represents a more comparable counterpart to ChameleonDB. Both include the LSM structure in their designs and use the same amount of DRAM. A comparison between these two designs help reveal the critical weakness with the LSM-tree structure itself.

For Dram-Hash, we use an open-source implementation of the high-performance robin-hood hash table [26] as its in-DRAM index. The Pmem-Hash is actually CCEH, a state-of-the-art persistent hash table [28]. We use CCEH source code provided by its designers with its default configuration. All the KV stores in comparison are developed with C/C++.

We list detailed configuration of ChameleonDB in Table 1. In all experiments, we use 8B keys and 8B values and Write-Intensive Mode is not enabled if not otherwise specified. In the following, we first present overall performance of ChameleonDB compared to its counterparts in Subsection 3.3, then present results on the YCSB workloads in Subsection 3.4. Finally, we will experimentally demonstrate benefit of Write-Intensive Mode and Get-Protect Mode in ChameleonDB in Subsections 3.5 and 3.6, respectively. We also compare ChameleonDB with two state-of-the-art LSM-tree key-value store designs in Subsection 3.7.

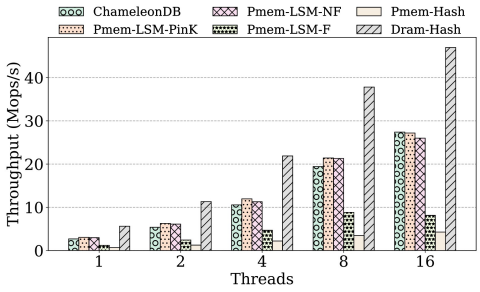


Figure 10. Put throughput.

3.3 Overall Performance

Put Throughput. We first evaluate the put throughput with different number of threads. Each put thread in ChameleonDB, Pmem-LSM-NF, Pmem-LSM-F, and Pmem-LSM-PinK is paired with a compaction thread to do compaction, while there is no compaction threads for Pmem-Hash and Dram-Hash. For fairness, we let the compaction threads and their corresponding put threads share the same core. As shown in Figure 10, we see that ChameleonDB outperforms Pmem-Hash consistently by about a factor of 5×. And among all the stores, Pmem-Hash shows the lowest put performance. Pmem-Hash allows small random writes to take place directly in the Optane persistent memory, which is not compatible with the Pmem's 256B block access constraint and results in a large write amplification.

ChameleonDB, Pmem-LSM-NF, and Pmem-LSM-PinK have similar put throughput, and they also show 2-3× higher put throughput than Pmem-LSM-F. The put throughput gap between Pmem-LSM-NF and Pmem-LSM-F highlights the impact of constructing Bloom filters with intensive use of CPU cycles on the I/O performance. As LSM structure enables efficient sequential Pmem access and Pmem has a much higher bandwidth than the SSD, CPU-intensive operations become much more impactful than previously believed and must be carefully considered in the design to avoid any potential CPU performance bottleneck.

ChameleonDB, Pmem-LSM-NF, and Pmem-LSM-PinK share some common performance advantages. Their writes are always large sequential ones thanks to the LSM structure. They remove Bloom filers to prevent CPU from becoming a performance bottleneck against high-performance Pmem access. Among the KV stores, ChameleonDB and Pmem-LSM-PinK replace some Pmem reads with DRAM reads during compactions. In the last level compaction ChameleonDB reads the upper level KV items from its in-DRAM Auxiliary Bypass Index(*ABI*). In Pmem-LSM-PinK any reads of upper levels for compactions are from the DRAM. Accordingly, as shown in Figure 10, Pmem-LSM-NF's put throughput is mostly lower than the other two stores. However, their performance gaps are small. When the high Pmem bandwidth enabled by sequential reads in the compactions has been in place (around 12GB/s), the performance impact of reads is substantially

Table 2. Tail Put Latency (ns)

KV Stores	99%	99.9%	99.99%
ChameleonDB	572	9950	31136
Pmem-LSM-PinK	638	10734	39084
Pmem-LSM-NF	622	10314	40740
Pmem-LSM-F	624	10522	41920
Pmem-Hash	10764	225998	929868
Dram-Hash	710	1472	31280

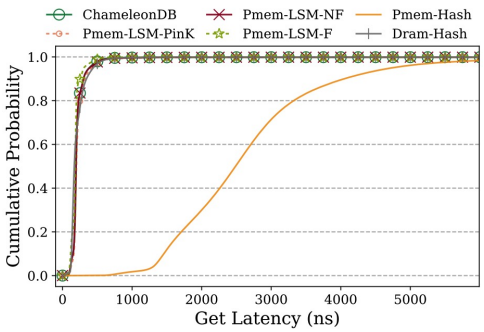


Figure 11. Put latency CDF.

reduced in comparison to more expensive CPU use and random in-DRAM access for hash operations during admission of KV items and compactions.

Dram-Hash shows 2× higher put performance than that of ChameleonDB (as well as Pmem-LSM-NF and Pmem-LSM-PinK). It completely avoids the LSM structure as well as its compaction operations, and leaves all hash index in the DRAM. While ChameleonDB keeps an in-DRAM hash index (the *ABI*) only for KV items in the LSM's upper levels, which are heavily involved in the compactions, Dram-Hash occupies about 3× larger DRAM space than ChameleonDB. Considering exponential growth of LSM level size, the gap on DRAM demand between them will keep increasing with the store size. A more serious consequence of keeping the index only in the DRAM is that it will take a long time to reconstruct it to resume its service after a system crash.

Put Latency. The cumulative distribution function (CDF) curves of put latency are shown in Figure 11. As can be seen, all curves except that for Pmem-Hash are clustered together in the left side of the figure, indicating that the medium latency and even the 99th percentile latency of these stores except Pmem-Hash are similarly low. Table 2 lists the precise tail latency measurements at the (very) high percentiles. The reason why Pmem-Hash has significantly higher put latency is that it persists KV items with small writes to persistent memory in individual put operations. The medium put latency of Pmem-Hash is around 12× that of ChameleonDB. The tail put latency of Pmem-Hash is around 18-29× that of ChameleonDB, as shown in Table 2. Pmem-LSM-F's medium and 99th percentile put latencies are close to those of ChameleonDB. However, its put throughput is much lower than that of ChameleonDB as shown in Figure 10. The reason is that some put operations

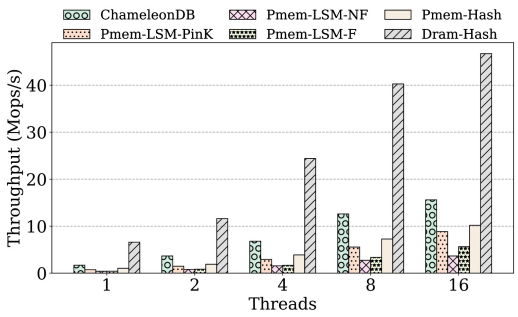


Figure 12. Get Throughput.

of Pmem-LSM-F have very long latency due to their concurrence with more expensive compactions requiring construction of Bloom filters. Pmem-LSM-F's largest put latency is up to 1.25sec in the experiments, which is 6× larger than those of ChameleonDB, Pmem-LSM-PinK, and Pmem-LSM-NF. Dram-Hash has lower p99.9 put latency than other stores, including ChameleonDB, with its in-DRAM index operations. However, its largest put latency is up to 3.23sec (13× of that of ChameleonDB) due to rehashing.

Get Throughput. Figure 12 shows the random get throughput of the stores with different number of threads. Among them, Pmem-LSM-NF exhibits the lowest get throughput as it needs to check multiple levels in the Pmem to find a KV item. Dram-hash shows the highest get throughput as a get operation requires only one check into the in-DRAM hash table and one Pmem read in the log. Except Dram-Hash, ChameleonDB shows the highest get throughput, which is 4.26×, 2.76×, 1.76×, and 1.53× of those for Pmem-LSM-NF, Pmem-LSM-F, Pmem-LSM-PinK, and Pmem-Hash, respectively. By using in-DRAM Bloom filters to avoid unnecessary Pmem reads, Pmem-LSM-F has higher get throughput than Pmem-LSM-NF. By pinning all levels except the last level in the DRAM, Pmem-LSM-PinK also receives higher get throughput than Pmem-LSM-NF. However, different from ChameleonDB which uses in-DRAM hash table (the *ABI*) to directly reach the target KV item in the upper levels (if exists), Pmem-LSM-PinK still has to take multiple checks in the DRAM as it retains the multi-level LSM structure in the DRAM, which makes its get throughput lower than that of ChameleonDB by 43% to 60%.

Get Latency. Figure 13 shows the get latency CDF curves for reading existing keys with one thread. As expected, Pmem-LSM-NF shows the largest get latency. Its medium get latency is 1.7× of that of Pmem-Hash. By using Bloom filters, Pmem-LSM-F reduces the get latency. However, its medium get latency is still 20% higher than that of Pmem-hash, which doesn't need a multi-level search. Without using Bloom filters, Pmem-LSM-PinK also needs to search multiple levels. However, the search in the LSM's upper levels is conducted in the DRAM, and the corresponding reads don't have the read amplification as that in the Pmem. Therefore, its get

Table 3. Tail Get Latency (ns)

KV Stores	99%	99.9%	99.99%
ChameleonDB	1506	2270	5560
Pmem-LSM-PinK	2544	3434	7774
Pmem-LSM-NF	4786	6082	17376
Pmem-LSM-F	4068	5554	18000
Pmem-Hash	2508	2994	11790
Dram-Hash	1608	2154	4000

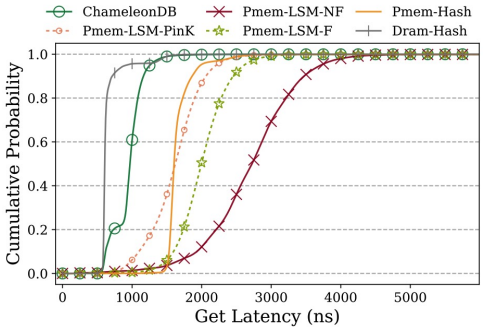


Figure 13. Get Latency CDF.

latency is better than Pmem-LSM-F. And its medium get latency is close to that of Pmem-Hash.

The CDF plot of ChameleonDB shows a two-stage curve, where the first stage corresponds to those get requests hitting in the *ABI* while the second stage corresponds to those for KV items in the LSM's last level. ChameleonDB shows 39%, 40%, 51%, and 64% lower medium get latency than Pmem-Hash, Pmem-LSM-PinK, Pmem-LSM-F, and Pmem-LSM-NF, respectively, highlighting the benefit of its use of *ABI* on reducing get latency. In particular, Pmem-LSM-PinK uses the same amount of DRAM space for improving its get performance. But still its get latency is significantly worse than that of ChameleonDB, indicating that not only amount of DRAM space but also the way to use the space matters in the effort of improving the stores' performance. Dram-Hash has the lowest medium get latency, which is 36% lower than that of ChameleonDB. However, without the in-Pmem LSM structure Dram-Hash loses all of its index on a system crash.

Table 3 shows the tail get latency at high percentile numbers. For the p99.9 get latency, ChameleonDB's latency is 24%, 33%, 59%, and 62% lower than those of Pmem-Hash, Pmem-LSM-PinK, Pmem-LSM-F, and Pmem-LSM-NF, respectively. And it is only 5% higher than that of Dram-hash.

DRAM Footprint and Restart Time. We show the DRAM footprint and restart time of the stores in Table 4. We record the DRAM footprint after inserting 1 billion unique keys to a store. A restart is considered complete when the KV store is ready to serve put/get requests.

Dram-Hash places a full index in the DRAM. It uses the the largest DRAM space. It also has the longest restart time as it needs to scan the entire log to reconstruct its index. For Pmem-Hash, all the user data and the hash table are immediately persisted in the Optane Pmem. It uses the DRAM to maintain some additional index components such as directories and segments in the CCEH hash table design. It needs to recover these structures upon restart. As shown in Table 4, its DRAM footprint and restart time are low. Pmem-LSM-NF's DRAM footprint and restart time are even lower, as it only keeps its MemTables (128MB) in the DRAM and recovers them upon a restart. Pmem-LSM-PinK has the same DRAM footprint as ChameleonDB but with lower get performance.

ChameleonDB has a higher DRAM footprint compared to Pmem-LSM-NF as it needs to keep the *ABI* in the DRAM to quickly reach KV items in the LSM's upper levels. But the DRAM footprint of ChameleonDB is only 26% of that used by Dram-Hash. Besides, the restart time of ChameleonDB is two orders of magnitude lower than that of Dram-Hash as ChameleonDB retains the advantage of LSM on short restart time, though its get performance will degrade to that of Pmem-LSM-NF until the *ABI* is fully rebuilt. The *ABI* is recovered along with serving front-end requests.

A Summary. We compare all the store designs in four measures, i.e., put throughput, get throughput, DRAM footprint, and restart time, in Table 4. In the table, we highlight the measurements that are significantly worse than their respective peers in the comparison. As shown, all of the stores, except ChameleonDB, have at least one "bad" performance measure. For instance, Dram-Hash has the best put throughput and get throughput. But it occupies the most DRAM space and has the longest restart time. Pmem-Hash requires a small amount of DRAM and can restart quickly. But its put throughput is the worst. Pmem-LSM-PinK, Pmem-LSM-NF, and Pmem-LSM-F have poor get throughput, among which Pmem-LSM-F also has poor put performance. The put and get throughput of ChameleonDB is only worse than that of Dram-Hash. But the DRAM footprint of ChameleonDB is much smaller than that of Dram-Hash. Furthermore, ChameleonDB can restart as quickly as the LSM-based stores. Also, the get latency of ChameleonDB is lower than all the other stores except Dram-hash, as shown in Figure 13. In summary, ChameleonDB is the only store that can achieve high put throughput, low get latency, small DRAM footprint, and speedy recovery/restart.

3.4 Experiments with the YCSB Workload

Yahoo Cloud Services Benchmark (YCSB)[5] is often used to comprehensively evaluate performance of KV stores. In this subsection, we evaluate ChameleonDB with six YCSB workloads. The workloads are detailed in Table 5. We do not include YCSB_E as it is for evaluation of the performance of range scan, which is not supported by the stores organized with hashed keys. For each of the workloads other than YCSB_LOAD and YCSB_D, we first run YCSB_LOAD to warm up the KV store with 1 billion puts, and then run the corresponding workload with 1 million requests. For YCSB_D, we issue 10K get requests for keys that are written

Table 4. Overall Comparison

Metrics	ChameleonDB	Pmem-LSM-Pink	Pmem-LSM-NF	Pmem-LSM-F	Pmem-Hash	Dram-Hash
Put Throughput (Mops/s)	27.40	27.19	26.00	**8.14**	**4.28**	46.96
Get Throughput (Mops/s)	15.60	**8.86**	**3.66**	**5.64**	10.19	46.71
DRAM Footprint (MB)	8325	8325	150	3277	729	**31744**
Restart Time (sec)	1	1	1	1	2	**102**

Table 5. YCSB Workloads

Workload	Description
YCSB_LOAD	100% put
YCSB_A	50% get / 50% update
YCSB_B	95% get / 5% update
YCSB_C	100% get
YCSB_D	Get most recently inserted keys
YCSB_F	50% get / 50% read-modify-write

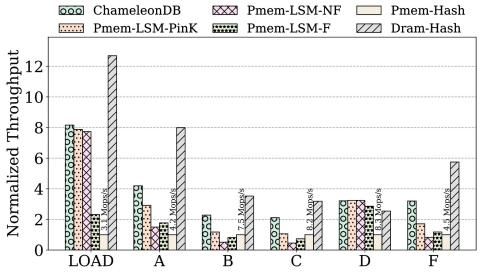


Figure 14. YCSB test results.

most recently right after YCSB_LOAD. In all the experiments 16 threads are used.

In Figure 14, the YCSB experiment results are shown in terms of normalized throughput, where the throughput of Pmem-Hash is normalized to 1. Pmem-Hash's actual throughput is displayed above the bar for the corresponding Pmem-Hash measurement. For all the workloads except YCSB_D, ChameleonDB outperforms any other stores except Dram-Hash. For YCSB_D, ChameleonDB, Pmem-LSM-PinK, and Pmem-LSM-NF show the similarly highest performance, because the most recently written keys are hit in MemTable. As expected, for the two write-intensive workloads (YCSB_LOAD and YCSB_A), Pmem-Hash shows the lowest throughput. On the other side, for the read-intensive workloads, Pmem-LSM-NF, which is known to have poor get performance, shows the lowest throughput. Dram-Hash shows the highest throughput for all workloads except YCSB_D, because checking the most recently written keys in a robin-hood hash table usually needs to probe more times due to collision resolution.

3.5 Impacts of Direct Compaction and Write-Intensive Mode

We introduced the Direct Compaction scheme in ChameleonDB that can reduce compaction overhead as detailed in Figure 5. Further, we also designed the Write-Intensive Mode for ChameleonDB, which is detailed in Subsection 2.3. Both Direct Compaction scheme and Write-Intensive Mode help to further improve the put performance of ChameleonDB.

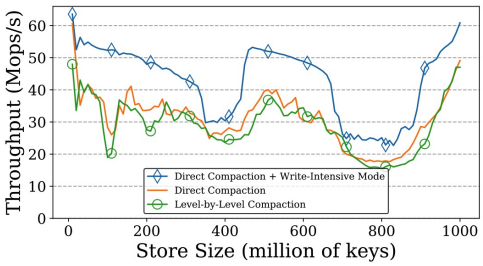


Figure 15. Impacts of different mode.

In this subsection, we experimentally assess how much they help to improve the put performance.

In the experiments we issue one billion put requests of unique keys to each of the stores. In Figure 15 we present the put throughput during service of the requests while using Level-by-Level Compaction, Direct Compaction, and Direct Compaction with the Write-Intensive Mode. Using the Direct Compaction scheme helps improve put throughput by about 7% on average over that using the Level-by-Level Compaction. By suspending maintenance of the multi-level structure (for LSM-tree's upper levels) in response to intensive writes, enabling Write-Intensive Mode of ChameleonDB with Direct Compaction further improves put throughput by 38% on average[3]. If the system crashes when ChameleonDB is running on the Write-Intensive mode, the restart time can be long as ChameleonDB has to recover the items in the *ABI* instead of only those in the MemTables. Restarting a ChameleonDB with one billion items that crashes during the Write-Intensive Mode takes about 30 seconds, which is longer than a standard restart of ChameleonDB but shorter than a restart of Dram-hash (102 seconds).

3.6 Benefit of Dynamic Get-Protect Mode

Tail latency of get requests is a critical QoS metric in many production systems. We conduct experiments to understand the impact of a burst of put requests on tail get latency (for the 99th percentile). In the experiments, we run Pmem-Hash and ChameleonDB with two 10-minute phases. In each phase's initial 30 seconds, only get requests are sent to the KV stores, and then a put burst with 100 million requests are sent to the stores to see how they impact the get operations.

As shown in the upper figure of Figure 16, the tail get latency of Pmem-Hash is around 2000ns when there are no put bursts, and then increases by a factor of 2.9× to 5800ns during

[3]Note that experiment results reported elsewhere about ChameleonDB are those on the store using only Direct Compaction

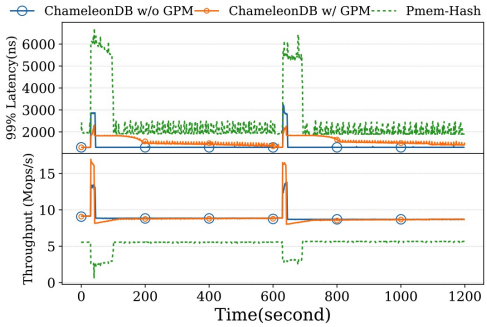


Figure 16. Get tail latency and overall throughput with two bursts put requests. In the experiment, ChameleonDB can be either without or with Dynamic Get-Protect Mode (GPM).

the put bursts. Similar trend is observed in ChameleonDB without dynamic GPM (Get-Protect Mode) where get tail latency increases from 1300ns to 2900ns. Though the impact of the put bursts in ChameleonDB without GPM is less than that of Pmem-Hash, the tail get latency still increases by 2.23×. A burst of put requests compete the Pmem's limited bandwidth and CPU cycles with get requests, resulting in a jump of get latency. In Pmem-Hash when a put request updates the hash table with multiple small writes, the small writes incur read-modify-writes in the Optane Pmem. In ChameleonDB, a burst of put requests lead to intensive LSM compactions which place high demand on Pmem bandwidth and CPU cycles. In the ChameleonDB's GPM mode, when a spike of read latency is detected, all compactions, which are induced by put requests, are temporarily suspended until the spike is weakened. The spike is defined by a pre-defined latency threshold, which is 2000ns in this experiment. The suspension includes pausing upper level compactions (the action taken in the Write-Intensive Mode) and postponing last level compactions. By applying the dynamic Get-Protect Mode, the peek tail latency of ChameleonDB is reduced to 2200ns, which is 24% lower than that without the mode[4]. However, as the postponed last level compactions are gradually carried out after the put burst period, the tail get latency takes some time to reduce to the pre-burst level. As shown in Figure 16, the tail latency during the reduction time period stays below 2000ns, which is even lower than the tail latency of Pmem-Hash during the get-only period.

In contrast, the impact of a burst of put requests on Pmem-Hash's tail get latency is much larger in terms of the spike's height and the length of the time period for it to reduce to the pre-spike level. During the put bursts, the throughput of Pmem-Hash is reduced by 55% because its put throughput is much lower than its get throughput. Due to the high put throughput of ChameleonDB and suspension of its compactions in the GPM, its overall throughput during the bursts is much higher and the duration of the bursts is much shorter.

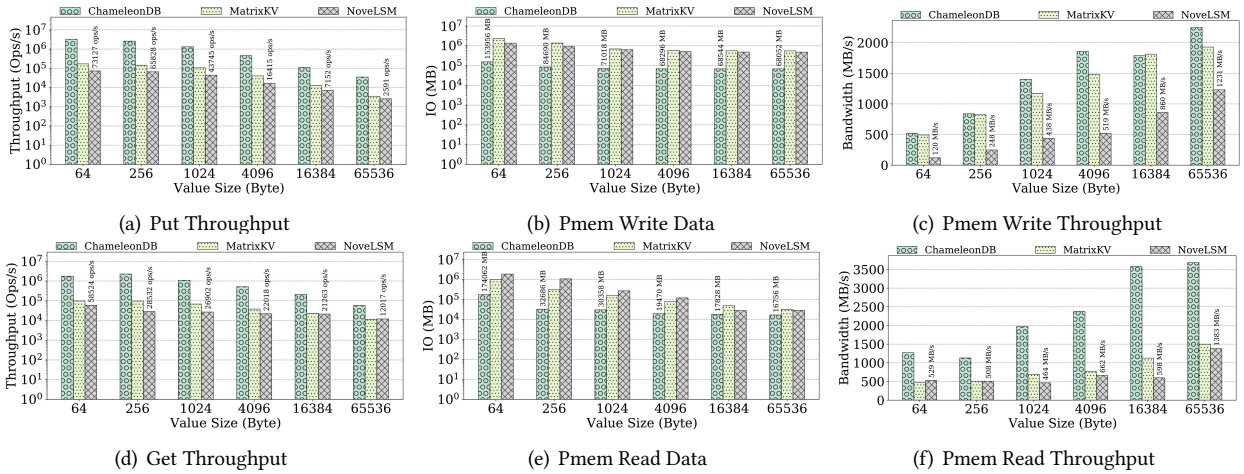
[4]Note that experiment results reported elsewhere about ChameleonDB are those on the store without applying the GPM mode.

3.7 Comparison with NoveLSM and MatrixKV on the Pmem use

With emergence of Pmem, there have been some recent studies on leveraging this fast and byte-addressable device to improve performance of LSM-based KV store designs. Two representatives in this direction of efforts are NoveLSM [21] and MatrixKV [38]. NoveLSM and MatrixKV are designed as hybrid KV stores on both Pmem and SSD. For NoveLSM, all levels are placed in the SSD. And it uses two in-Pmem MemTables to persist arriving KV items and sort them there. For MatrixKV, all levels except Level 0 ($L0$) are placed in the SSD. It allows the $L0$ levels in the Pmem to enable finer-grained compactions to the SSD to avoid write stalls. For the sake of fairness, we place all of their LSM levels in the persistent memory in the experiments.

In this subsection we compare ChameleonDB with NoveLSM and MatrixKV in terms of put and get throughput. All of them are configured with 128MB MemTables just like ChameleonDB. As ChameleonDB's *ABI* additionally uses 8GB DRAM, we configure NoveLSM and MatrixKV each with 8GB data cache in the DRAM. Note that both *ABI* and the data cache are used to improve get performance. By using the same amount of DRAM space, these stores can be compared in a fair manner. We use source code of NoveLSM [20] and MatrixKV [29] provided by their respective designers. As NoveLSM was developed based on LevelDB and supports only one background thread for compaction, we run the three stores with one thread to perform compaction for fair comparison. The key size is 16B while the value size varies.

In the put experiment, we write a total of 64GB data using put requests whose value size ranges from 64B to 64KB. The results are shown in Figure 17(a) (Note that the y axis is shown in the logarithmic scale). As shown, ChameleonDB outperforms NoveLSM and MatrixKV by up to 44× and 19×, respectively. This large performance gap comes from two aspects. First, the amount of data written by NoveLSM and MatrixKV to the Pmem are 8.8× and 15.4× higher than that by ChameleonDB, as shown in Figure 17(b). These data amounts are collected by intel's ipmwatch about raw data written to the Pmem's media, including amplified writes due to its 256B block access. There are several reasons why NoveLSM and MatrixKV write much more data to the Pmem. Both MatrixKV and NoveLSM use the leveling compaction that is known to have very high write amplification [7, 8, 27], while ChameleonDB uses the tiering compaction of low write amplification for all upper LSM levels. Furthermore, for NoveLSM, building mutable MemTable in the Pmem requires direct insertion of small KV items into its skip lists, resulting in a large write amplification, an issue similar to that with Pmem-Hash(CCEH). This is one of the major issues addressed by ChameleonDB. For MatrixKV, the size of its metadata in its RowTable for acceleration of get performance is significant, especially when the value size is small (e.g., 45%



(a) Put Throughput (b) Pmem Write Data (c) Pmem Write Throughput

(d) Get Throughput (e) Pmem Read Data (f) Pmem Read Throughput

Figure 17. Comparison with two state-of-the-art Pmem LSM-tree designs.

of the KV data size when the value is 64 bytes). As the RowTables are in the Pmem, writing these metadata increases traffic to the Pmem. Second, the throughput of NoveLSM and MatrixKV's writes to the Pmem are 77% and 5% lower than that of ChameleonDB, respectively, as shown in Figure 17(c). This is because they have more intensive use of CPU cycles, often making CPU become performance bottleneck, with more expensive key sorting and construction of Bloom filters. In particular, NoveLSM uses the filters in all of the LSM levels, and MatrixKV uses the filters in all of the levels except *L*0.

For the get experiment, we read 16GB data using get requests on random keys. The get throughput with various value size is presented in Figure 17(d). ChameleonDB outperforms NoveLSM and MatrixKV by up to 29× and 17×, respectively. The much higher get performance of ChameleonDB is attributed to its hash table and its *ABI* design. For NoveLSM, as it has a mutable MemTable in Pmem, a get operation requires checking its in-Pmem MemTable, which results in many random Pmem reads. For any random access, Pmem's performance is especially worse than that of DRAM. MatrixKV holds a big *L*0 level with multiple sublevels without Bloom filters. Though it conducts its search using so-called cross-row hints to eliminate binary search at each sub-level, it cannot avoid checking the sub-levels in a one-by-one manner. As shown in Figure 17(e), the amount of data read from Pmem for NoveLSM and MatrixKV are much larger than that of ChameleonDB. Furthermore, as a get operation in NoveLSM and MatrixKV involves searching MemTable, checking Bloom filters, binary searching index, and scanning data block, which are computationally intensive compared with checking a hash table in ChameleonDB, NoveLSM and MatrixKV have substantially lower Pmem read bandwidth compared with ChameleonDB, as shown in Figure 17(f). In the experiments, the data cache in the DRAM is only about 12% of the entire data set (8GB vs. 64GB), its impact on NoveLSM and MatrixKV's get performance is limited with random access. In a real-world use, we expect the Pmem and data set will be (much) larger and the ratio between the cache and the data set will be (much) smaller. Therefore, a data cache cannot fundamentally address the performance issue except when there exits very strong access locality. In Figure 17(e) we can find that the amount of data read from Pmem by ChameleonDB with 64B value size is much larger than that with 256B value size. With a 256B value size, there is a smaller number of keys with a fixed data set (64GB) in the experiments. ChameleonDB's *ABI*, a hash table whose size is capped, can accommodate most keys for fast key access. However, with a 64B value size there are much more keys, and the *ABI* is quickly filled up leading to more frequent compaction of keys to the LSM's last level in the Pmem. With more keys that have to be searched in the last level without help of the *ABI*, ChameleonDB's get throughput with the 64B value is even lower than that with 256B value size.

4 Related Works

Persistent Hashing. Existing works on persistent hashing mostly focus on building a hash index by directly updating the index in-place on persistent memory, such as PFHT [9], level hashing [40], and CCEH [28]. PFHT is a cuckoo hashing variant that is optimized to reduce memory writes during serving write requests by allowing at most one displacement in a write. Level hashing applies a two-level hash scheme so that each key can have three buckets as the candidates for insertion, which helps improve the load factor. Instead of double hashing, CCEH is an extendable hashing that uses a linear probing strategy so that a successful insertion only requires one memory write. All of these works try to reduce persistent memory writes during insertion in order to design a write-optimized hashing. However, for the existing persistent-memory device, Optane Pmem, the in-place update designs still cause large write amplifications as Optane

Pmem's physical media access granularity is 256B [37]. Unlike these works, ChameleonDB aggregates insertion in a small DRAM hash table, and flushes the hash table to the Optane Pmem, so as to leverage the high write bandwidth of Optane Pmem.

Key-value stores for Persistent Memory. SLM-DB [19] presents a single-level KV store by applying a global persistent B+-tree index. The KV data are appended to a log while each record is indexed by a B+-tree to speed up the search. As the B+-tree indexes all keys in the store, it is large and has to stay in the Pmem, which causes small updates on the tree with a large write amplification. ChameleonDB uses a hash table to speed up search. Its hash table indexes a much smaller number of keys. So it is much smaller and can reside in the DRAM to make small updates occur only in the DRAM. FlatStore [2] maintains a volatile global index to serve read requests while maintaining a data log on the persistent memory. Because the volatile index does not survive a system crash, FlatStore suffers from a long recovery time (40 seconds to recover a store with one billion records according to results in [2]). ChameleonDB does not have the long recovery issue because it only needs to rebuild the in-DRAM hash tables that contain a small portion of KV items (e.g., 1/256 for a four-level design).

It's critical to accurately understand a storage device's performance characteristics for an effective KV-store design on the device. Xia et al. [36] had assumed a Pmem with its writes slower than DRAM and reads of comparable latency with DRAM in their design of a hybrid KV store on the Pmem and SSD. However, after an extensive study of the Intel Optane Pmem, Yang et al. [37] find that this Pmem has write latency similar to that of DRAM. It's the small writes, which are incompatible with the Pmem's 256B access unit leading to write amplification, that seriously compromise its write performance. They also report that Optane Pmem's read latency is 2× to 3× lower than DRAM. ChameleonDB's design accommodates these real performance characteristics with its small writes only in the DRAM and reduced reads using an in-DRAM hash-table-based bypass index.

Bullet addresses the performance gap between optimized persistent memory and DRAM resident KV stores using a Cross-Referencing Logs (CRLs) technique to move persistent memory writes off the critical path [14]. ChameleonDB closes the gap as well with a different approach, which is reducing the access on the Optane Pmem with an in-DRAM hash table and batching small writes. NoveLSM [21] and MatrixKV [38] are LSM-tree KV stores exploiting persistent memory to improve write performance. NoveLSM proposes to maintain large mutable MemTables in persistent memory to enable direct small updates, while MatrixKV proposes to maintain $L0$ in the persistent memory to enable finer-grained compactions. We compared ChameleonDB with NoveLSM and MatrixKV. The results reveal that ChameleonDB outperforms them in both put and get performance because (1) it avoids use of small writes in the Pmem to accommodate its 256B access unit, (2) it uses a sharded structure to reduce LSM level count for fine-grained compaction and low write amplification, and (3) it uses both size-tiering and leveling compactions to reduce write amplification and also retain low read latency.

Key-Value Stores for Fast SSD. 3D-Xpoint is the non-volatile memory technology developed by Intel and Micron [18] and is also used to build some of the fastest SSDs, such as Intel Optane SSD 9 Series [17], which is much faster than the preceding SSDs. Meanwhile, Samsung also develops ultra-low latency SSD (Z-SSD [32]) to compete with Intel. With emergence of the new SSD technology, many works have been conducted to improve the efficiency of KV store as the bottleneck shifts from the storage to the CPU. KVell [24] applies shared-nothing and asynchronous I/O solutions to improve the performance and exploit full disk I/O bandwidth by avoiding a CPU bottleneck. Zhang et al. [39] proposes an FPGA-accelerated compaction for LSM-based KV store to reduce the CPU bottleneck for storing small KV items. PinK [16] eliminates upper layer bloom filters and removes the CPU cost for compaction by using FPGA-based SSD.

5 Conclusion

In this study, we present challenges of building KV stores with high put throughput, low get latency, low DRAM footprint, and rapid restart on the Intel Optane Pmem, and reveal shortcomings of existing KV store designs. As none of the existing designs can achieve all the above objectives, we propose ChameleonDB, a KV store design that leverages respective strengths of existing designs while addresses their drawbacks. By leveraging a multi-level structure, ChameleonDB aggregates writes to exploit the 256B write unit property of the Optane Pmem. By using an in-DRAM hash table to avoid multi-level checking, ChameleonDB provides competitive get latency with in-Pmem hash table designs. By introducing the dynamic Get-Protect Mode, ChameleonDB can reduce the tail get latency during put bursts. Our experiment results show that it outperforms state-of-the-art persistent hash table KV store designs by up to 8.5× in put throughput and 2.5× in get throughput. We also compared ChameleonDB with two state-of-the-art LSM-tree designs on Optane Pmem (NoveLSM and MatrixKV), demonstrating that ChameleonDB significantly outperforms them in both put and get throughput.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Vijay Chidambaram, for their valuable feedback. This work is supported by the National Science Foundation under grants CCF-1704504, CNS-2008835, and CCF-1815303.

References

[1] Shimin Chen and Qin Jin. 2015. Persistent B+-Trees in Non-Volatile Main Memory. *Proc. VLDB Endow.* 8, 7 (Feb. 2015), 786–797. https://doi.org/10.14778/2752939.2752947

[2] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. 2020. FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*. Association for Computing Machinery, New York, NY, USA, 1077–1091. https://doi.org/10.1145/3373376.3378515

[3] Brian Choi, Parv Saxena, Ryan Huang, and Randal Burns. 2020. Observations on Porting In-memory KV stores to Persistent Memory. arXiv:cs.DB/2002.02017

[4] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. 2020. SplinterDB: Closing the Bandwidth Gap for NVMe Key-Value Stores. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC'20)*. USENIX Association, 49–63. https://www.usenix.org/conference/atc20/presentation/conway

[5] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SOCC'10)*. Association for Computing Machinery, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[6] Intel Corporation. 2019. *Intel® Optane™ Persistent Memory 128GB Module*. https://www.intel.com/content/www/us/en/products/memory-storage/optane-dc-persistent-memory/optane-dc-128gb-persistent-memory-module.html

[7] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal Navigable Key-Value Store. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD'17)*. Association for Computing Machinery, New York, NY, USA, 79–94. https://doi.org/10.1145/3035918.3064054

[8] Niv Dayan and Stratos Idreos. 2018. Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD'18)*. Association for Computing Machinery, New York, NY, USA, 505–520. https://doi.org/10.1145/3183713.3196927

[9] Biplob Debnath, Alireza Haghdoost, Asim Kadav, Mohammed G. Khatib, and Cristian Ungureanu. 2015. Revisiting Hash Table Design for Phase Change Memory. In *Proceedings of the 3rd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads (INFLOW'15)*. Association for Computing Machinery, New York, NY, USA, Article 1, 9 pages. https://doi.org/10.1145/2819001.2819002

[10] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. 2018. Reducing DRAM Footprint with NVM in Facebook. In *Proceedings of the 13th EuroSys Conference (EuroSys'18)*. Association for Computing Machinery, New York, NY, USA, Article 42, 13 pages. https://doi.org/10.1145/3190508.3190524

[11] Facebook. 2020. *RocksDB: a persistent key-value store.* https://rocksdb.org/

[12] P. Flajolet, P. Poblete, and A. Viola. 1998. On the Analysis of Linear Probing Hashing. *Algorithmica* 22, 4 (1998), 490–515. https://doi.org/10.1007/PL00009236

[13] Google. 2020. *LevelDB: a fast key-value storage library written at Google that provides an ordered mapping from string keys to string values.* https://github.com/google/leveldb

[14] Yihe Huang, Matej Pavlovic, Virendra Marathe, Margo Seltzer, Tim Harris, and Steve Byan. 2018. Closing the Performance Gap Between Volatile and Persistent Key-Value Stores Using Cross-Referencing Logs. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC'18)*. USENIX Association, Boston, MA, 967–979. https://www.usenix.org/conference/atc18/presentation/huang

[15] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18)*. USENIX Association, Oakland, CA, 187–200. https://www.usenix.org/conference/fast18/presentation/hwang

[16] Junsu Im, Jinwook Bae, Chanwoo Chung, Arvind, and Sungjin Lee. 2020. PinK: High-speed In-storage Key-value Store with Bounded Tails. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC'20)*. USENIX Association, 173–187. https://www.usenix.org/conference/atc20/presentation/im

[17] Intel. 2020. Intel® Optane™ SSD 9 Series. https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/consumer-ssds/optane-ssd-9-series.html.

[18] Micron Intel. 2020. 3D XPoint Technology. https://www.micron.com/products/advanced-solutions/3d-xpoint-technology.

[19] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young ri Choi. 2019. SLM-DB: Single-Level Key-Value Store with Persistent Memory. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*. USENIX Association, Boston, MA, 191–205. https://www.usenix.org/conference/fast19/presentation/kaiyrakhmet

[20] Sudarsun Kannan. 2021. *NoveLSM.* https://github.com/sudarsunkannan/lsm_nvm

[21] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC'18)*. USENIX Association, Boston, MA, 993–1005. https://www.usenix.org/conference/atc18/presentation/kannan

[22] Kornilios Kourtis, Nikolas Ioannou, and Ioannis Koltsidas. 2019. Reaping the performance of fast NVM storage with uDepot. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*. USENIX Association, Boston, MA, 1–15. https://www.usenix.org/conference/fast19/presentation/kourtis

[23] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.* 44, 2 (April 2010), 35–40. https://doi.org/10.1145/1773912.1773922

[24] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2019. KVell: The Design and Implementation of a Fast Persistent Key-Value Store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)*. Association for Computing Machinery, New York, NY, USA, 447–461. https://doi.org/10.1145/3341301.3359628

[25] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. WiscKey: Separating Keys from Values in SSD-Conscious Storage. *ACM Trans. Storage* 13, 1, Article 5 (March 2017), 28 pages. https://doi.org/10.1145/3033273

[26] martinus. 2020. *robin_hood unordered map & set.* https://github.com/martinus/robin-hood-hashing

[27] Fei Mei, Qiang Cao, Hong Jiang, and Jingjun Li. 2018. SifrDB: A Unified Solution for Write-Optimized Key-Value Stores in Large Datacenter. In *Proceedings of the ACM Symposium on Cloud Computing 2018 (SoCC'18)*. Association for Computing Machinery, New York, NY, USA, 477–489. https://doi.org/10.1145/3267809.3267829

[28] Moohyeon Nam, Hokeun Cha, Young ri Choi, Sam H. Noh, and Beomseok Nam. 2019. Write-Optimized Dynamic Hashing for Persistent Memory. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*. USENIX Association, Boston, MA, 31–44. https://www.usenix.org/conference/fast19/presentation/nam

[29] PDS-Lab. 2021. *MatrixKV.* https://github.com/PDS-Lab/MatrixKV

[30] pmem.io. 2020. *Persistent Memory Programming.* https://pmem.io/

[31] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. PebblesDB: Building Key-Value Stores Using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*. Association for Computing Machinery, New York, NY, USA, 497–514. https://doi.org/10.1145/3132747.3132765

[32] Samsung. 2020. Z-SSD | Samsung Semiconductor Global Website. https://www.samsung.com/semiconductor/ssd/z-ssd/.

[33] Xingbo Wu, Zili Shao, and Song Jiang. 2015. *Selfie*: Co-Locating Metadata and Data to Enable Fast Virtual Block Devices. In *Proceedings of the 8th ACM International Systems and Storage Conference (SYSTOR'15)*. Association for Computing Machinery, New York, NY, USA, Article 2, 11 pages. https://doi.org/10.1145/2757667.2757676

[34] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. 2015. LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data Items. In *Proceedings of the 2015 USENIX Annual Technical Conference (ATC'15)*. USENIX Association, Santa Clara, CA, 71–82. https://www.usenix.org/conference/atc15/technical-session/presentation/wu

[35] Yinjun Wu, Kwanghyun Park, Rathijit Sen, Brian Kroth, and Jaeyoung Do. 2020. Lessons learned from the early performance evaluation of Intel Optane DC Persistent Memory in DBMS. arXiv:cs.DB/2005.07658

[36] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. 2017. HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC'17)*. USENIX Association, Santa Clara, CA, 349–362. https://www.usenix.org/conference/atc17/technical-sessions/presentation/xia

[37] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST'20)*. USENIX Association, Santa Clara, CA, 169–182. https://www.usenix.org/conference/fast20/presentation/yang

[38] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. 2020. MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC'20)*. USENIX Association, 17–31. https://www.usenix.org/conference/atc20/presentation/yao

[39] Teng Zhang, Jianying Wang, Xuntao Cheng, Hao Xu, Nanlong Yu, Gui Huang, Tieying Zhang, Dengcheng He, Feifei Li, Wei Cao, Zhongdong Huang, and Jianling Sun. 2020. FPGA-Accelerated Compactions for LSM-based Key-Value Store. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST'20)*. USENIX Association, Santa Clara, CA, 225–237. https://www.usenix.org/conference/fast20/presentation/zhang-teng

[40] Pengfei Zuo, Yu Hua, and Jie Wu. 2018. Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*. USENIX Association, Carlsbad, CA, 461–476. https://www.usenix.org/conference/osdi18/presentation/zuo