Pacman: An Efficient Compaction Approach for Log-Structured Key-Value Store on Persistent Memory

Jing Wang, Youyou Lu, Qing Wang, and Minhui Xie, *Tsinghua University;*
Keji Huang, *Huawei Technologies Co., Ltd;* Jiwu Shu, *Tsinghua University*

https://www.usenix.org/conference/atc22/presentation/wang-jing

This paper is included in the Proceedings of the
2022 USENIX Annual Technical Conference.

July 11–13, 2022 • Carlsbad, CA, USA

978-1-939133-29-8

Open access to the Proceedings of the
2022 USENIX Annual Technical Conference
is sponsored by

Pacman: An Efficient Compaction Approach for Log-Structured Key-Value Store on Persistent Memory

Jing Wang[†] Youyou Lu[†] Qing Wang[†] Minhui Xie[†] Keji Huang[‡] Jiwu Shu[*†]

[†]*Department of Computer Science and Technology, BNRist, Tsinghua University*
[‡]*Huawei Technologies Co., Ltd*

Abstract

Recent persistent memory (PM) key-value (KV) stores adopt the log-structured approach to reap PM's full potential. However, they fail to sustain high performance at high capacity utilization due to inefficient compaction. The inefficiency results from the unawareness of PM's characteristics. This paper proposes Pacman, an *efficient PM-aware compaction approach* for log-structured KV stores on PM. Pacman (1) offloads reference search during compaction to service threads, so as to mitigate the onerous index traversal overhead, (2) leverages tagged pointer and DRAM-resident compaction information to avoid excessive PM accesses introduced by garbage collection, (3) redesigns the compaction pipeline based on the PM peculiarities to lower the persistence overhead, and (4) separates hot and cold objects in a lightweight manner to reduce PM data copying in compaction. We apply Pacman to state-of-the-art PM-based log-structured KV stores and evaluate Pacman using various benchmarks. Our evaluations show that Pacman curtails the performance degradation at high capacity utilization, increases the compaction bandwidth by 2-4×, and boosts the performance of the state-of-the-art systems by 1.5-1.8× under write-intensive workloads.

1 Introduction

The log-structured approach has been adopted in key-value (KV) stores on PM [8, 12, 30, 44] for benefits like high capacity utilization (low fragmentation), small device-level write amplification on PM, and low failure atomicity overhead. Despite having these benefits, the log-structured approach needs to reclaim free space (i.e., garbage collection or compaction) by dedicated background threads (called *cleaners*), which contributes to the major bottleneck, especially under a high capacity utilization [39].

Over-provisioning can alleviate this problem but is not cost-efficient. Datacenters place more emphasis on space utilization in recent years [15] and try hard to fully utilize their storage to reduce the total cost of ownership (TCO) [40]. Thus, it is important for storage systems to keep high performance under high space utilization, especially for PM that has a much higher cost than traditional storage devices.

The compaction overhead is already severe in DRAM-based log-structured KV stores. For example, RAMCloud's throughput could drop by up to 50% at high capacity utilization [39]. Nibble, a concurrent log-structured KV store, enables 8 compaction threads to carry out the compaction work per socket which has only 15 cores; even so, its throughput drops to nearly a quarter in dynamic workloads [35].

Worse still, we observe that PM's idiosyncrasies exacerbate the bottleneck of compaction. Our evaluation on state-of-the-art PM-based log-structured KV stores[1] (including FlatStore [12] and Viper [8]) shows that their performance drops significantly at high capacity utilization. With abundant CPU resources for compaction (foreground thread count to background cleaner thread count is 3:1), when the capacity utilization increases from 50% to 80%, the system throughput drops by up to 75% under write-intensive workloads. Unfortunately, simply adding more CPU resources for compaction is inefficient, because the performance of PM does not scale well with high thread count [46]; in our experiment, the decline is still up to 60% even with doubling cleaner threads.

We analyze that there are four deficiencies in the conventional compaction approaches of these KV stores accounting for the performance degradation. First, after copying valid objects from the log segment being reclaimed, cleaners need to update object references in the index, which incurs a huge overhead, especially when the index resides on PM due to PM's high access latency (3× of DRAM in terms of random read [46]). Second, service threads (i.e., threads performing user requests, such as Get and Put) generate quantities of small random PM accesses for compaction (e.g., marking deleted flags). These small random accesses result in I/O amplification of PM, wasting PM's limited bandwidth (1/3 and 1/6 of DRAM in terms of read and write, respectively [46]). Third, cleaners need to perform many expensive persistence

[*]Jiwu Shu is the corresponding author (shujw@tsinghua.edu.cn).

[1]Note that this paper targets PM-based log-structured KV stores but not LSM-tree-based KV stores.

instructions to guarantee crash consistency. Fourth, excessive data copying in compaction contends limited PM bandwidth with service threads. All these deficiencies boil down to *unawareness of PM's characteristics*.

To solve the problems above, this work proposes PACMAN, an efficient PM-aware compaction approach for log-structured KV stores on PM. PACMAN comprises a series of techniques to improve the compaction efficiency of log-structured KV stores on PM. ① PACMAN introduces a technique called *shortcut*, which offloads reference search operations during compaction to service threads. Therefore cleaners can locate and update references without traversing the index. ② PACMAN reduces excessive PM random accesses. Specifically, PACMAN leverages *tagged pointer* to reduce high-latency PM reads, and stores frequently-accessed metadata in DRAM to avoid small random PM writes. ③ PACMAN redesigns the compaction pipeline in a batch pattern and leverages several optimizations according to the characteristics of PM, which accelerates the compaction and reduces the persistence overhead. ④ PACMAN adopts a lightweight hot-cold separation method to reduce the amount of valid data copying on PM and corresponding reference updates. Consequently, PACMAN boosts compaction efficiency, decreases CPU resources for compaction, and enhances system performance at high capacity utilization.

We apply PACMAN to state-of-the-art PM-based log-structured KV stores, FlatStore [12] with different indexes and Viper [8], and evaluate PACMAN using a variety of benchmarks. Our evaluation shows PACMAN enhances the compaction bandwidth by up to 4× and system performance by 2.4-4.6× under write workloads at high capacity utilization. Besides, PACMAN has nearly no side-effects under read-intensive workloads and has little overhead on recovery.

In summary, this paper makes the following contributions:

- We analyze the deficiencies of existing compaction approaches for log-structured KV stores on PM.
- We propose PACMAN, an efficient PM-aware compaction approach for PM-based log-structured KV stores, which enables them to achieve high performance even at high capacity utilization.
- We apply PACMAN to state-of-the-art PM-based log-structured KV stores and conduct a series of experiments to show the efficiency of PACMAN.

2 Background and Motivation

2.1 Log-Structured KV Stores on PM

Benefits of the log-structured approach on PM. The log-structured approach has been adopted by state-of-the-art KV stores on PM [8, 12, 30, 44] for the following benefits. First, a log-structured approach to memory management supports high capacity utilization (i.e., the percentage of space used by alive data) of 80-90%, which is unfeasible for non-copying allocators having great memory fragmentation [39]. Second,

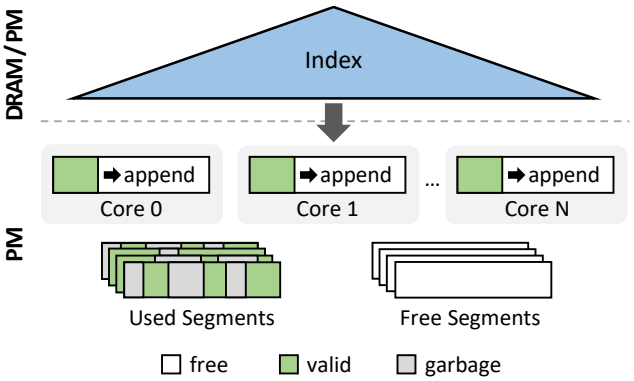

Figure 1: Overview of a log-structured KV store.

due to the mismatch of access granularities between cache lines and PM media (64 bytes vs. 256 bytes of Optane DIMM, the only available PM production for now), small random writes would cause write amplification. The log-structured approach adopts a sequential write pattern, thus alleviating the write amplification and improving the write throughput. Third, in comparison with update-in-place approaches which need expensive logging operations, the log-structured approach makes it easy to commit an arbitrarily-sized persistent write.

Storage structure. Figure 1 shows the basic structure of log-structured KV stores on PM. The whole log space locates on PM and is divided into small-sized (e.g., 4 MB) pieces called *segments*. Each service thread maintains a thread-local segment to append KV objects. Compared to writing to a global log tail, using per-thread segments not only avoids contention but also limits the number of concurrent threads accessing an Optane DIMM [8, 46]. Once a service thread has run out of its local segment, it requests a new free segment from the free segments pool. A global index stores *references* which point to the actual address of KV objects in the log. The index is put in DRAM or PM for different requirements. A volatile index in DRAM delivers better performance but needs more time to restore after a restart. On the contrary, a persistent index in PM provides instant usability after a restart but relatively lower performance.

Garbage collection. Despite having numerous benefits, log-structured systems are obliged to tackle garbage collection by compaction, the main culprit of performance degradation. Update or delete operations make prior objects stale in the log. These stale objects occupy the memory space until being reclaimed. When there is no free space left, service threads stall and wait for new free space produced by compaction. In other words, the system's throughput at high capacity utilization is nearly limited to the compaction throughput.

A qualified compaction process is described as follows. When the fraction of free space is low, the compaction is triggered, and candidate segments are selected to compact through a certain strategy such as cost-benefit score[2]. During

[2] $score = \frac{(1-u)\times age}{u}$, where u is the segment's utilization (fraction of data alive), and age is the time since the segment running out [39].

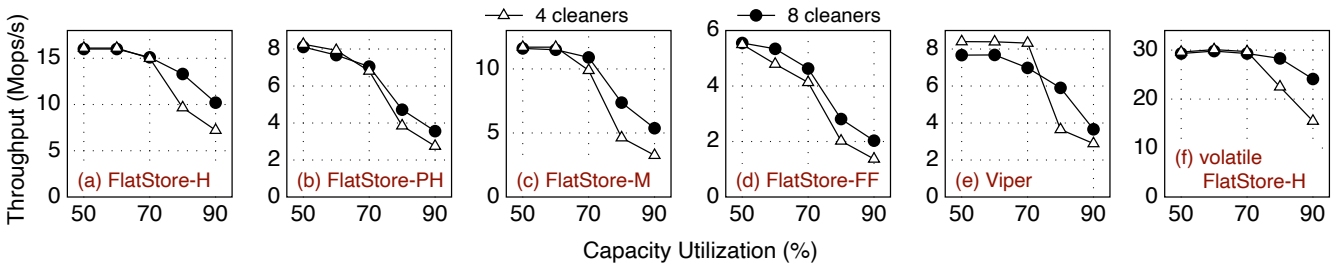

Figure 2: Throughput decline at different capacity utilizations. *FlatStore-H: with CCEH in DRAM. FlatStore-PH: with CCEH in PM. FlatStore-M: with Masstree in DRAM. FlatStore-FF: with FastFair in PM. Viper: with CCEH in DRAM. volatile FlatStore-H: both the index (CCEH) and log in DRAM.*

compaction, cleaners copy all alive objects from the old segment to a reserved segment and update references to these objects in the index. There are two methods to identify the liveness of each object in the segment. One method is checking whether the object is still pointed by the reference in the index (e.g., FlatStore [12]). Another one is checking the deleted flag (usually 1 bit) in the metadata header of objects; the deleted flag is set when the corresponding object is updated or deleted (e.g., Viper [8]). Both of the two methods have shortcomings on PM, which we will analyze in §2.2. After all valid objects in the old segment have been copied to the reserved segment and their references have been updated, the old segment is cleaned and turned into a free segment.

2.2 Compaction Overhead Analysis on PM

The compaction overhead already matters in DRAM-based log-structured KV stores [35, 39]. Worse still, *the compaction overhead becomes more severe on PM*, as cleaner threads need to contend PM's limited bandwidth with foreground service threads. Further, necessary but expensive persistence instructions and PM's high access latency make the overhead of copying objects and updating references higher, especially when the index is persistent.

Experiments. To analyze the compaction overhead of PM-based log-structured KV stores in depth, we evaluate state-of-the-art systems including FlatStore [12][3] and Viper [8][4].

We evaluate four versions of FlatStore, including FlatStore-H (FlatStore with CCEH [37], a hash table, in DRAM as a volatile index), FlatStore-PH (FlatStore with CCEH in PM as a persistent index), FlatStore-M (FlatStore with Masstree [34], a trie-like concatenation of B+-trees, in DRAM), and FlatStore-FF (FlatStore with FastFair [20], a B+tree, in PM) to show different cases. Like FlatStore-H, Viper also uses CCEH in DRAM as its volatile index. We measure the performance with a YCSB-A workload, where 200 million KV objects are randomly loaded first, then service threads perform a write-intensive workload (50% Get and 50% Put) with Zipfian distribution (skewness parameter 0.99) until the system throughput converges to a stable value. The value size is 48 bytes, a representative value of small objects according to recent real-world workloads analysis [9]. We restrict the capacity utilization from 50% to 90% (the percentage of space occupied by alive data). We set the service thread count to 12, and the cleaner thread count to 4 or 8. All threads are bound to a single socket, which is equipped with three Intel Optane DCPMMs.

Figure 2 shows their throughput at different capacity utilizations. From the results we observe that: (1) *The throughput of all systems drops significantly at high capacity utilization*, especially for systems with a tree-based or persistent index. (2) *Simply augmenting more CPU resources for compaction (8 cleaner threads) has limited improvements for log-structured KV stores on PM*, and also intensifies the contentions. Note that the garbage collection overhead will be much larger under uniform workloads; see detailed results in §4.2.3.

We also evaluate the situation that both index and log locate on DRAM (volatile FlatStore-H, Figure 2(f)) to simulate an in-DRAM log-structured KV store. The throughput decline is mitigating than the PM-based KV stores. Using 8 cleaners has obvious improvements on DRAM because DRAM has abundant bandwidth. However, adding more threads on compaction is not cost-effective and has less benefit in PM-based systems as PM does not scale well with multiple threads due to PM's idiosyncrasies [46]. This shows that PM's peculiarities aggravate garbage collection overhead.

Overhead analysis. Taking test cases with 4 cleaners and 80% capacity utilization above as examples, we analyze the compaction overheads and find out that there exist four inefficiencies on compaction.

(1) The high latency of random access in the index. Lots of random accesses are introduced by two operations on the index, checking references for identifying liveness and updating references after copying alive objects. These two operations require multiple random accesses in the index and these accesses have no cache locality since alive objects in compaction are usually cold.

Viper sidesteps checking references with deleted flags in PM; yet, updating references costs half of the compaction

[3] As the original FlatStore is a networked system, we implement it as an embedded KV store, and remove the network-related features for simplicity.

[4] We use the variable-sized object version of Viper and enhance it with multi-threaded compaction.

time. FlatStore identifies the liveness of each object by checking the reference instead of using the deleted flag to avoid small random writes. Thus, cleaners in FlatStore need to access the index twice for each alive object, one for checking liveness and one for updating the reference. FlatStore spends 60% of compaction time on the index in FlatStore-H. The overhead becomes more severe when it comes to tree-based or persistent indexes, which reach about 80% and 90% of compaction time in FlatStore-M and FlatStore-FF, respectively. The latency of a search or an update operation in FastFair reaches several microseconds due to PM's high latency.

(2) Excessive small random access on PM. Service threads perform excessive PM accesses for garbage information. To maintain the size of garbage data of each candidate segment, service threads read the metadata of the stale object to get its size when the object is updated or deleted. These reads on PM not only incur high latency but also pollute cache. In addition, marking deleted flags in segments would introduce extra small random PM writes. Though marking deleted flags facilitate garbage collection, it is harmful to limited PM bandwidth.

(3) Expensive persistence instructions. Though copying alive objects conducts sequential reads and writes, it still costs heavier than copying on DRAM, not only due to the low bandwidth but also because of necessary but expensive persistence instructions (i.e., `flush` and `fence`).

(4) A large amount of data copying on PM. The performance slowdown presents superlinear scaling with capacity utilization. Cleaners have to do compaction much promptly at high utilization, and segments to compact have less time to accumulate stale objects. Accordingly, at high capacity utilization, cleaners have to copy bulk of data in segments to reclaim free space. The large amount of data copying contends the limited PM bandwidth with service threads.

To summarize, traditional compaction approaches do not consider the peculiarities of PM, hence squandering limited PM bandwidth. On the other hand, existing log-structured KV stores completely decouple the index and log. Therefore, there has little room to facilitate garbage collection without particular assistance from the index.

3 Design

Motivated by the analysis above, we propose Pacman, a PM-aware compaction approach for log-structured KV stores on PM. Pacman solves the deficiencies in conventional compaction approaches according to the characteristics of PM, boosting the system performance at high capacity utilization. Pacman introduces several core design principles to realize efficient compaction.

- **Avoid onerous index traversal.** Pacman offloads reference search operations during compaction to foreground service threads without extra effort. With the information offered by service threads, cleaner threads can locate and update references effortlessly (§3.1).
- **Reduce excessive small PM accesses.** Pacman removes avoidable PM accesses by leveraging tagged pointer and storing frequently-accessed metadata in DRAM (§3.2).
- **Redesign the compaction pipeline to cater to peculiarities of PM.** Pacman divides the compaction pipeline into two major phases, copying valid objects and updating references. For each phase, cleaner threads process objects in a batched manner. Subsequently, Pacman can reduce the number of persistence orderings and leverage non-temporal stores and software prefetching (§3.3).
- **Separate hot and cold objects to reduce excessive data copying on PM.** Pacman uses a hotspot set to distinguish hot and cold objects and stores them in different segments to facilitate compaction. Pacman can also replace the set silently to handle hotspot shift without blocking foreground service threads (§3.4).

3.1 Traversing Index with Shortcut

Traversing the index has expensive overhead due to the random access pattern, especially for persistent index. Pacman introduces a technique called `shortcut` to alleviate this overhead in compaction. In this section, we take tree-based (including trie-based) indexes as an example.

In existing compaction approaches, after copying a valid object from an old segment to a new segment, the cleaner needs to traverse the index, locate and update the reference. When locating an entry, the cleaner starts from the root node and then traverses multiple internal nodes to the deepest leaf node. The pointer-chasing path contains several random accesses, and has much higher latency on PM. According to our analysis (§2.2), these reference update operations constitute the most considerable part of overhead in compaction.

However, we observe that the root-to-leaf path was already traversed when the object was created or updated. To prevent the cleaner from traversing the high latency path again, Pacman leverages the traversal that was done before.

When creating or updating an object, in addition to insert or update of the reference in the index, service threads also record the address of leaf node which contains the reference in the log. We name this additional information `shortcut`. Objects thereupon have a `shortcut` to their reference in the index. During compaction, cleaners could take the shortcut to quickly locate the leaf node in the index. Then, cleaners still need to search the exact entry in the leaf node. To accelerate the last mile, Pacman also records the position number of the entry in the shortcut. In this way, the reference search operation is offloaded to service threads but without extra effort. Figure 3 shows an example of a shortcut. The `Node Addr` points to the node, and the `KV Pos` records the entry position number of the array.

Handling shortcut invalidation. In addition, Pacman needs to handle the possible invalidation of the shortcut in two situations. ① The address of an index entry may change (e.g.,

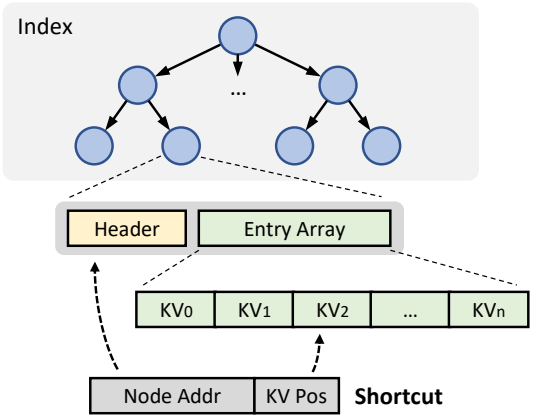

Figure 3: Structure of shortcut (taking a tree-based index as an example). *The value in an index's KV pair is a reference which points to the object in the log.* `KV Pos` *is the position number of the associated KV pair (i.e., 2 in this figure).*

caused by shift operations in a sorted tree-based index) and thus the shortcut may not point to the original reference. ② The original node pointed by shortcut may have been deallocated, and the original address space may be reclaimed or re-allocated for other usages. Accessing the address wrongly could result in program crash.

The first situation (i.e., `Node Addr` points to a valid node, but `KV Pos` is wrong) can be handled easily by conducting some checks. The cleaner will check the header of the node and compare the key indicated by `KV Pos` to infer that whether the shortcut is correct. These checks are identical as in a normal insert operation. PACMAN attempts to reduce penalty of an incorrect shortcut. For example, in a tree-based index, the cleaner will check if the key still exists in the original leaf node or the sibling node. Thus, shortcuts can tolerate shift operations to a certain extent. If the shortcut is completely invalid, the cleaner updates the reference by falling back to the normal update operation. The penalty of an invalid shortcut is about one or several useless memory accesses and can be further reduced by prefetch technique (§3.3).

To avoid the second situation (i.e., `Node Addr` does not point to a valid node), instead of directly freeing the space of deleted nodes, PACMAN reserves the deallocated space for future allocations. The deleted nodes are marked as deleted by some means (e.g., a deleted bit in node header or all bytes of the header are set to 0). When creating a new node, PACMAN first attempts to re-use a reserved and same-typed node space. Generally, an index typically has only one or a few fixed types of nodes (e.g., 4 types of nodes in ART [29]), so it is easy to realize the re-allocation. In this way, PACMAN guarantees that the address space of deleted nodes is still valid and this situation turns into the first situation.

Optimizing space overheads. PACMAN stores shortcuts inside the same log segment with their associated objects, which squeezes the available space and increases compaction pressure. Thus, the benefits of shortcuts are overshadowed especially when the capacity utilization is extremely high and the average object size is small.

PACMAN reduces the space overheads of shortcuts to minimize the punishment of storing shortcuts. First, the size of a shortcut is compressed to 48 bits, including 43-bit `Node Addr` and 5-bit `KV Pos`. The `Node Addr` is compressed based on two opportunities: 1) Current virtual address only uses 47 bit (for user-space virtual address, the 47-63th bits are 0); 2) Memory allocators (e.g., malloc [1], PMDK [4]) allocate objects with at least 16-byte alignment by default. The size of `Node Addr` can be further reduced according to the specific alignment of nodes in the index (e.g., 512-byte leaf nodes in FastFair). Second, PACMAN doesn't store shortcuts for objects reclaimed in compaction and hot objects with the help of hot-cold data separation (§3.4). The objects reclaimed in compaction are almost coldest, and the reserved segment are less likely to be compacted again. Hot objects tend to be updated soon and become stale, and shortcuts are useless for stale objects. Besides, shortcut should be disabled when its acceleration cannot cover the punishment of sacrificing more space (e.g., for a hash table-based index and at an extremely high capacity utilization).

Limitation. Shortcut is unsuitable for KV stores with a LSM-based index (e.g., ChameleonDB [50]). Since KV entries in the LSM-based index are moved frequently due to LSM compactions, shortcuts can only stay valid for a transient time.

3.2 Reducing Excessive PM Accesses

Existing garbage collection approaches do not fit persistent memory management as they take no notice of random memory accesses. The introduced small random accesses will cause I/O amplification in PM. To address this issue, PACMAN 1) embeds size information in the reference to reduce PM read, and 2) stores frequently-accessed metadata in DRAM to reduce small random writes on PM.

Embedding size information in the reference. When updating an object, the service thread needs to update the size of total garbage data of each segment for compaction candidate selection. However, getting the size of the stale object needs to read its metadata from PM. To avoid these random PM reads, PACMAN embeds the size of an object in the upper 16 bits of its reference. Therefore, the address and size of the object's stale version can be acquired from the index together. Because 16 bits can express 64 KiB at most (or larger if objects are allocated obeying to some alignments), PACMAN sets the upper bits of reference to 0 for objects larger than 64 KiB and has to read their metadata when updated or deleted. Fortunately, most objects have small size according to the recent real-world workloads analysis [9].

DRAM-resident garbage information. Since checking references to distinguish the liveness of objects has significant

overhead, especially when the index is on PM, PACMAN turns to deleted flags to store the liveness information. Marking deleted flags in objects' headers on PM will introduce numerous small random writes. To eschew the detrimental effects of small random writes on PM, PACMAN adopts a bitmap on DRAM for each segment. PACMAN locates the corresponding bit of a variable-sized object with its reference directly. We denote the minimum size of an object as `MIN_SIZE` (8 bytes of key, 8 bytes of value, plus the size of metadata header). PACMAN reserves one bit per `MIN_SIZE`. The position of the deleted flag is calculated by dividing the offset within the segment by `MIN_SIZE`. Though this approach leaves some bits unused thereby wasting a small amount of DRAM space, PACMAN can quickly locate the corresponding deleted flags for variable-sized log items. Even in the most extreme case, where the size of the key, value, and header are both 8 bytes, the bitmaps consume DRAM about 0.5% of the log space.

Besides bitmaps, PACMAN stores the size of garbage data of each segment in DRAM which is frequently updated.

3.3 Redesigning the Compaction Pipeline

Traditional compaction algorithms (e.g., memory compaction in RAMCloud and Viper) update the reference right after copying a valid object. Nevertheless, this pattern has several shortcomings on PM without consideration of PM's idiosyncrasies. PACMAN reorganizes the pipeline in a batch pattern as shown in Figure 4. The new algorithm separates the phases of copying objects and updating references, and processes objects in a batched pattern. The cleaner first collects all valid objects from the old segment to a volatile buffer (step ①), and then copies them together to PM (step ②). After that, the cleaner updates their references. Different from using a normal index update operation, the cleaner updates references by a special index update operation (`update_pacman`, lines 43-52) that takes the object's shortcut to locate the reference. Besides, to handle race condition on the index, `update_pacman` carries an extra old value of the reference (`old_addr`) to update the reference in a compare-and-swap semantics (explained later). Subsequently, PACMAN applies the following three optimizations to cater to PM's idiosyncrasies.

(1) Reducing ordering and launch concurrent flushes. In traditional algorithms, for each relocated object, an ordering point is required to ensure that the object has been flushed to PM before updating its reference. These fences are expensive as they stall CPU pipelines.

However, after separating the copying phase and the updating phase, only one fence instruction is needed between the two phases (line 15). PACMAN eliminates the ordering points (`fence`) after update references (step ④). Doing so will not break crash consistency. This is because the old segment is still available until the compaction is finished. Even if an inopportune crash happens before reference updates being flushed to PM (during step ⑤), after restart, for those objects whose references have not been updated or persisted, they can still be acquired from the old segment.

```
1  NUM_BATCH_FLUSH = 32; // number of concurrent flushes
2  void compact_pacman(Segment segment) {
3    Buffer buffer;  // temporal buffer in DRAM
4    vector<ObjectMeta> meta_vec;
5    // iterate objects in this segment
6    for (valid object old_obj : segment) {
7      // ①. generate temporal new object into buffer
8      tmp_new_obj = make_object(old_obj, buffer.offset);
9      buffer.append(tmp_new_obj);
10     meta_vec.push_back(ObjectMeta(old_obj, tmp_new_obj));
11   }
12
13   // ②. copy buffer to reserved segment by ntstore
14   ntstore(reserved_segment, buffer);
15   fence();
16
17   vector<EntryAddr> entry_addr_vec; // for batch persist
18   // iterate valid objects in buffer
19   for (size_t i = 0; i < meta_vec.size(); i++) {
20     // ③. prefetch next object's entry in index
21     prefetch(meta_vec[i + 1].shortcut);
22
23     // ④. update reference
24     (shortcut, key, old_addr, new_addr) = meta_vec[i];
25     EntryAddr entry_addr;
26     index.update_pacman(shortcut, key, new_addr, old_addr,
27                         &entry_addr);
28
29     // ⑤. batch persist
30     entry_addr_vec.push_back(entry_addr);
31     if (entry_addr_vec.size() >= NUM_BATCH_FLUSH) {
32       // launch concurrent flushes
33       for (entry_addr : entry_addr_vec) {
34         persist(entry_addr);
35       }
36       fence();
37       entry_addr_vec.clear();
38     }
39   }
40 }
41
42 // customized index update operation
43 bool Index::update_pacman(Shortcut shortcut, KeyType key,
44           ValueType new_addr, ValueType old_addr,
45           EntryAddr *entry_addr) {
46   // find entry by key with the help of shortcut
47   ...
48   *entry_addr = &entry; // record entry address
49   // update only if old_addr matched, e.g., using CAS
50   bool success = CAS(&entry.value, old_addr, new_addr);
51   return success;
52 }
```

Figure 4: Pseudo-code of the PACMAN compaction algorithm and customized index update operation.

Furthermore, PACMAN adopts lazy and batched flushes on reference updates to take advantage of concurrent asynchronous flushes [17], such as `clwb` and `clflushopt`. PACMAN records addresses of updated entries in the index, and launches multiple asynchronous flushes on them, which reduces the average flush latency (step ⑤).

(2) Using non-temporal store to copy valid objects. In copying phase, PACMAN first collects valid objects in volatile segments (step ①), then uses non-temporal store (`ntstore`) to copy them to PM (step ②) for three benefits. First, `ntstore`

has higher bandwidth for large (over 256 B) write than normal store [46]. Second, `ntstore` bypasses the cache and avoids unnecessary cache pollution. Third, `ntstore` can also avoid repeated flushes on the same cache line due to non-aligned writes, which incurs dramatical delay [12].

Though flushes are not necessary for data persistence in new generation CPUs with support of eADR [2], sequential writes (e.g., copying objects) without flushes will turn into random writes on PM due to the random eviction of CPU cache [23]. Therefore, the batch pattern is still beneficial on new CPUs with eADR for adopting `ntstore`.

(3) Leveraging prefetching on PM. Since most valid objects being collected are cold, their references in the index are less likely to stay in cache, which results in high memory access latency, especially for PM indexes. Fortunately, with the shortcut, PACMAN can easily prefetch the index node or entry of the next valid object when updating the current object's reference (step ③), hiding the high access latency with update operation. Thanks to shortcuts and the redesigned pipeline, the reference update operations are quite lightweight, which makes prefetching'effect much more evident.

Handling race condition on the index. The contention between cleaner threads and service threads should be handled properly. A service thread may update an object while a cleaner thread copies the old version of this object and updates its reference. In such a case, the new version of the object updated by the service thread will be covered by the old version of the object. The batched compaction pipeline increases the possibility of this race condition.

One naive approach is holding a lock and blocking service threads during the whole time of relocating an object (lookup reference, copy object, and update reference) [39]. PACMAN minimizes the critical section on updating references, which updates references in a compare-and-swap pattern. Specifically, cleaner threads update references via a customized index update operation (i.e., `update_pacman`) that carries an extra old value of the reference (`old_addr` in step ④). In `update_pacman`, only when the original value of the index entry matches the old value `old_addr`, which means no race condition has happened, the reference can be updated to `new_addr`. If the update fails, the reclaimed object in new segment is marked as stale.

Comparing with FlatStore's compaction pipeline. Though FlatStore [12] also optimize the compaction pipeline by separating the copying phase and the reference updating phase to reduce the fence instructions, it suffer from severe performance issue. Specifically, as FlatStore conducts the index traversal twice for each valid object (i.e., check the liveness and update the reference), the batch pattern exacerbates the overhead on the index. Results in a long reuse distance between the two index traversals for each alive object, causing updating reference not to take the benefit of cache brought by checking reference. By contrast, PACMAN eliminates the first index traversal by checking in-DRAM deleted flags and reduces the second traversal overhead by prefetching and shortcuts.

3.4 Separating Hot-Cold Data

PACMAN leverages hot-cold data separation to reduce the amount of valid data copying on PM and corresponding reference updates during compaction. Specifically, service threads append hot objects in their per-thread segment and cold objects in another per-thread cold segment. Though hot-cold data separation is a well-known technique to improve garbage collection efficiency [10,18,36], we elaborately design how to 1) identify hotspots and 2) handle hotspots shift in the context of the low-latency key-value store on PM.

Identifying hotspots in a lightweight pattern. PACMAN uses a small read-only hash set of hot objects for service threads to distinguish hotspots. The identification of hotspots should be lightweight enough to not increase too much extra latency. Previous work could distinguish the hotspots without maintaining the hotness of objects, such as by data type [28] or hash-based partition [10]. However, these methods are not feasible for PACMAN due to the lack of type semantics or the per-thread logs. Dynamically maintaining hotspots set is not ideal, because it introduces extra operations to another index, resulting in contention and crash thrashing.

Handling hotspots shift and generating new hotspots set. As the access patterns keep changing in real-world workloads, PACMAN uses a lightweight mechanism to detect hotspots and generate new hotspots set. First, service threads keep counting the hit ratio of the old hotspots set to detect whether a hotspots shift has happened. Second, if the hit ratio is lower than a customized threshold which means the hotspots have shifted and the original hot set is stale, service threads record the keys of updated objects in their local circular buffers by sampling. A background thread collects the keys recorded by service threads, sorts these keys using a heap, and generates a new hot set. Third, the background thread changes the pointer of hot set to the new one using `CAS`. After waiting for a grace period through an RCU-like barrier, the background thread confirms that no service threads are accessing or will access the old hot set, then frees the old hot set safely. Note that the background thread consumes negligible CPU resources as the hotspots shift in real-world workloads is not frequently (at least second-level) [9,21]. It doesn't matter if service threads identify hot keys inaccurately since it only determines the location of objects. Moreover, cold objects get another chance to be separated from hot objects by compaction.

Though checking the read-only hash set is lightweight, it would bring no benefit but extra overheads in uniform workloads. In the second step above, if the occurrence of the hottest key is close to the occurrence of the coldest key (e.g., less than 3×), the background thread clears the new hot set.

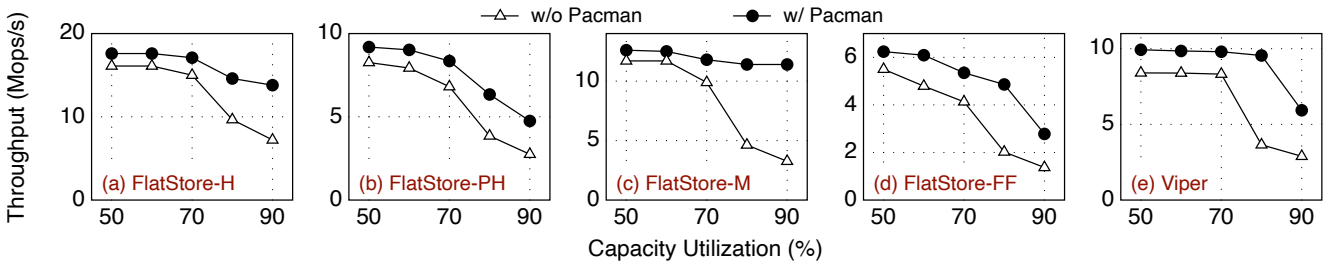

Figure 5: Impact of capacity utilization.

3.5 Recovery

The recovery of the log and the index is similar to existing work [12]. We mainly discuss recovery related to PACMAN.

Shortcut. For KV stores with a volatile index, all shortcuts are invalid after a restart. However, since all segments have to be scanned to rebuild the index, new valid shortcuts are rebuilt in the meantime. For KV stores with a persistent index, to make shortcuts still valid after a restart, PACMAN stores the offset from the base address of the PM pool for the index (e.g., `PMEMobjpool` in PMDK [4]) in the `Node Addr` instead of the virtual address of the node.

DRAM-resident information. Though the DRAM-resident information is unavailable after recovery, the loss of this information has no impact on service threads.

For KV stores with a volatile index, since the index needs to be recovered by scanning all segments, the bitmaps are recovered together with the index at the same time. To distinguish the latest and stale objects, PACMAN compares their version number and retains the latest reference in the index. For KV stores with a persistent index, the volatile bitmaps and metadata only cripple the garbage collection after recovery for a while. Cleaner threads scan the segments and recover the bitmaps by checking references in the index. Then the scanned segments become candidates for compaction.

Crash of compaction. After a restart, unfinished compaction does not need to resume. If a crash happens before the reference updating phase (before line 16 in Figure 4), after a restart, the new segment is still marked as a free segment as nothing has happened. If a crash happens in the middle of the reference updating phase (after line 16 in Figure 4), after a restart, background threads will scan these segments and distinguish redundant objects as described above.

4 Evaluation

In this section, we use a series of experiments to evaluate PACMAN. After describing our setup (§4.1), we first conduct experiments to show the overall performance of PACMAN on PM-based log-structured KV stores under various workloads (§4.2). Then, we analyze the benefit of each optimization of PACMAN with different cases (§4.3). Last, we compare log-structured KV stores with PACMAN against other KV stores on PM with a production workload (§4.4).

4.1 Experimental Setup

All experiments are conducted on a server with Intel Xeon Gold 6240 CPUs. Each CPU has 18 physical cores (36 logical cores with hyper-threading). Each socket is equipped with three 128 GB Intel Optane DC Persistent Memory (DCPMM) DIMMs and 96 GB DRAM. We bind all threads to a single socket to avoid NUMA effect [25, 46]. The Optane DIMMs are configured in App Direct mode.

We apply PACMAN to four versions of FlatStore [12] and Viper [8] we have evaluated in §2.2. FlatStore adopts a log batching technique to reduce the persisting overhead. Viper leverages PM-specific access patterns and employs CCEH [37] in DRAM as its index.

We set the upper limit of the hotspot set size (§3.4) to 128K in all experiments. We co-locate the background thread for generating new hotspot set (§3.4) with a random cleaner thread. The `MIN_SIZE` (§3.2) is set to 32, which means that the deleted flag bitmaps use 1 bit in DRAM for every 32 bytes in segments, which is 0.4% of the whole log size. We use 8-byte keys in all evaluated systems. Unless otherwise stated, we restrict the capacity utilization to 80%. Also, the value size is fixed at 48 for simplicity, because performance mainly depends on the average object size but not the exact size distribution [39] and the value size is corresponding to the recent real-world workloads [9].

4.2 Overall Performance

4.2.1 Impact of Capacity Utilization

We show how PACMAN mitigates the performance decline at high capacity utilization with the same experiments as in §2.2, in which 12 service threads perform write-intensive (50% Put, Zipfian distribution with parameter 0.99) workloads and 4 cleaners conduct the compaction work. Figure 5 shows the results, from which we make three observations.

First, PACMAN obviously curtails the performance decline at high capacity utilization. FlatStore-M with PACMAN maintains throughput above 90% even at extremely high capacity utilization (90%). Due to the huge overhead of compaction, original systems can not fully utilize their performance at high utilization. PACMAN improves the efficiency of compaction according to PM's peculiarities, and therefore reduces the compaction overhead. Though the performance declines of FlatStore-PH

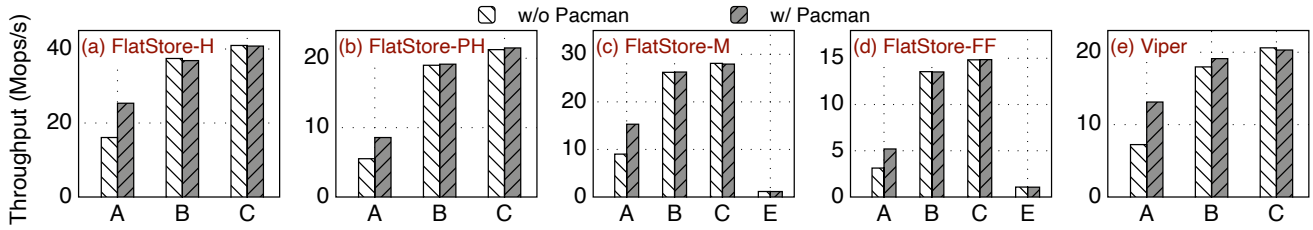

Figure 6: YCSB workloads performance. *(FlatStore-H, FlatStore-PH, and Viper don't support scan operations in YCSB-E.)*

and FlatStore-FF with PACMAN at 80% utilization still exceed 20%, they are much better than the original systems.

Second, PACMAN also enhances the performance at low capacity utilization, which is because PACMAN reduces small random accesses on PM and saves PM's limited bandwidth. The improvement on Viper is more evident since Viper marks deleted flags and modifies locks on PM.

Third, systems using PACMAN with 4 cleaners also outperform the original systems with 8 cleaner threads (see §2.2). PACMAN saves CPU resources for compaction but brings more improvements, which is not only due to the efficient compaction, but also because of the reduction in contentions on both the index and PM resources [46].

4.2.2 YCSB Benchmark

In this section, we evaluate the basic performance with YCSB [13] benchmark. Table 1 shows the characteristics of workloads. We omit the YCSB-D as it has similar traits to YCSB-B. We set 24 threads to perform workloads after random prefilling 200 million objects, and 4 cleaners for all evaluated systems. Each service thread performs 20 million operations. The capacity utilization is restricted to 80%.

Workload	Feature	Read-Write-Scan %
A	write-intensive	50-50-0
B	read-intensive	95-5-0
C	read-only	100-0-0
E	scan-intensive	0-5-95

Table 1: YCSB workloads description.

The experimental results are presented in Figure 6, from which we have two observations.

First, under write-intensive workload (YCSB-A), PACMAN improves the performance of each system by 1.5-1.8×. PACMAN improves Viper most among these systems, which is because PACMAN on Viper not only increases the compaction efficiency but also reduce small random writes on PM.

In this workload, 4 cleaner threads are insufficient for systems without PACMAN. The cleaner threads' CPU utilizations are all above 95%. For evaluated systems with PACMAN, there are a few unused CPU cycles. For example, with PACMAN the cleaner threads' CPU utilization is 81% in FlatStore-H and 92% in FlatStore-FF. The random prefilling phase invalidates a part of shortcuts. For example, the invalidation ratio of shortcuts is about 25% in FlatStore-FF and 58% in FlatStore-M. FlatStore-FF has lower invalidation ratio because FastFair has larger leaf node than Masstree and can tolerate more shift operations. Note that we regard the shortcut as valid if the entry can be found the in the node indicated by the shortcut or its sibling node.

Second, under read-dominated workloads (YCSB-B, C, and E), systems with PACMAN have similar performance with the original systems. This is because PACMAN does not directly influence read and scan operations, their performance under read-dominated workloads is similar.

4.2.3 Sensitivity Analysis

In this section, we evaluate how workload characteristics affect PACMAN. The default configurations are the same as in §4.2.1. We only show results of FlatStore-H with PACMAN in Figure 7 as a representative sample.

Uniform workloads. In uniform workloads, the system's performance drops compared with that in skewed workloads (Figure 5(a)), which is because of poor locality and much severe garbage collection overhead. The raw FlatStore-H drops more than half at 80% capacity utilization. However, PACMAN still curtails the throughput decline within 15% at 80% utilization and 50% at 90% utilization.

Thread scalability. Due to the compaction overhead, the raw FlatStore-H can not scale well with multiple threads. On the contrary, with only 4 cleaners, FlatStore-H with PACMAN can scale linearly up to about 20 threads.

Value size. The throughput of both systems drops with the value size getting larger because larger objects consume more bandwidth. PACMAN enhances the throughput of FlatStore-H by 50-70% as the value size varies from 32 to 512, which shows that value size has little impact on PACMAN.

Write ratio. Since the compaction overhead increases with write ratio, the improvement of PACMAN on FlatStore-H is more significant with higher write ratio, reaching to 2.3× when the write ratio is 100%. A strange phenomenon is that FlatStore-H with PACMAN has higher throughput when the write ratio is 80% than 60%. We find that cold segments have shorter ages due to the faster write rate. Thus, cleaners are more likely to select hot segments to compact according to the cost-benefit strategy, which has lesser overhead.

Number of objects. With the capacity utilization fixed at 80%, we vary the number of objects to prefill. The number of ob-

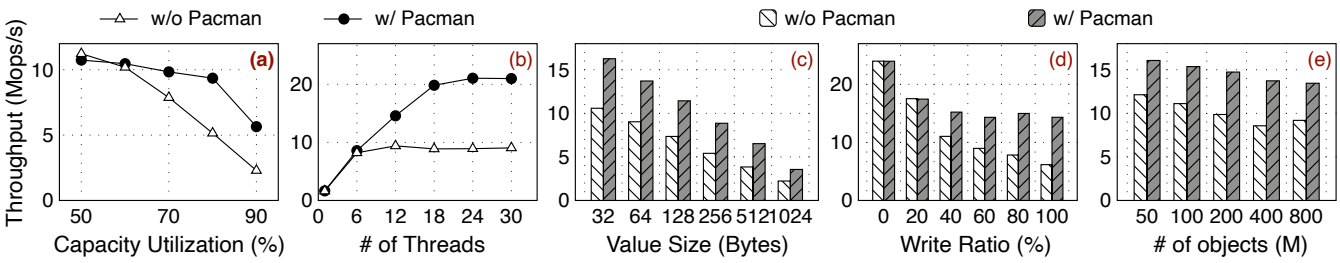

Figure 7: Sensitivity Analysis (on FlatStore-H). *(a) Uniform workloads. (b) Thread scalability. (c) Different value sizes. (d) Different write ratios. (e) Different numbers of objects.*

jects has two effects on the performance. On one hand, the CPU cache miss rate gets higher with the larger memory footprint (both the index and the log space). On the other hand, as the log space gets larger, the system writes more segments when compaction is triggered and has more time to accumulate stale objects. Thus, the candidate segments for compaction have less live data and the compaction is less expensive. For different numbers of objects, Pacman can improve the performance by 35-60%.

4.3 Analysis of Techniques

In this section, we analyze the performance benefit of each technique by applying them one by one. To differentiate results more obviously, we stress the system with an artificial workload at a high capacity utilization level (80%) modeled after prior work [39], in which the system throughput is heavily limited by compaction. After loading 200 million KV objects, 12 service threads overwrite these objects following a Zipfian access distribution with skewness parameter 0.99, and 2 cleaner threads perform the compaction work. The workload proceeds for a while until the system throughput and compaction overhead converge to a stable value.

We only show results of Pacman on FlatStore-H and FlatStore-FF since their results are representative.

Figure 8 shows the contribution of each technique to the throughput and corresponding compaction bandwidth (bytes of cleaned segments per second). The applying order is determinded by the dependencies between these techniques (e.g., the shortcut relies on hot-cold separation to reduce its space overhead). Specifically, we gradually apply to the ***raw*** systems with ***reducing*** avoidable PM accesses (§3.2), hot-cold data ***separation*** (§3.4), ***shortcut*** (§3.1), and the redesigned ***batching*** compaction pipeline (§3.3).

The results show that all techniques contribute to the improved performance more or less in some cases.

① ***Reducing.*** Storing garbage information in DRAM and embedding size information in reference eschew the avoidable PM random accesses. Especially in FlatStore, this technique avoids checking references in the index for identifying liveness of objects. Therefore, it brings about 80% improvements on FlatStore with a tree-based index (i.e., FlatStore-FF and

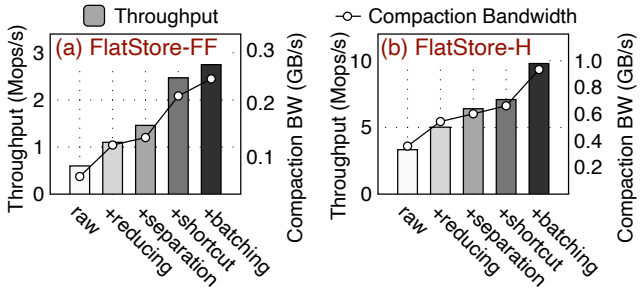

Figure 8: Contributions of techniques to throughput and compaction bandwidth.

FlatStore-M), and about 50% on FlatStore-H.

② ***Separation.*** Hot-cold data separation improves the system performance about 30%. The improvement of system throughput is more than the compaction bandwidth. This is because that hot-cold data separation alleviates the mixture of stale and valid objects in a segment, thus decreasing the amount of valid objects moving and compaction time.

③ ***Shortcut.*** Though Pacman trades some available log space for storing shortcuts, shortcuts still boost the system performance by about 70% for FlatStore-FF and about 45% for FlatStore-M (not shown in the figure) even at high capacity utilization. Moreover, since we do not store shortcuts for hot objects, the rate of successfully using shortcuts to update references is 55% in FlatStore-FF in this experiment. The effect is not obvious for systems with a hash table-based index, since the benefit brought by shortcuts is overshadowed by the additional overhead of storing shortcuts. Since Pacman only stores shortcuts for inserted cold objects, and we assume that about half of inserted objects are cold, the space overhead is less than 4% for 64B objects. Note that this space overhead has been paid by Pacman.

④ ***Batching.*** The batching compaction pipeline and corresponding optimizations bring another 40% improvement on FlatStore-M and FlatStore-H. However, batching has smaller effect on FlatStore with a persistent index, which is because the main overhead of compaction comes from operations on the persistent index.

Put them together, Pacman increases the compaction bandwidth, and improves the system throughput by about 3× for

Figure 9: Facebook ETC throughput and latency. *(Y axis in P99 Put Lat. is in log scale.)*

FlatStore-H and 4.6× for FlatStore-FF.

The results of Pacman on Viper are similar to FlatStore-H, except for ***reducing*** and ***batching***. As Viper uses deleted flags in objects, ***reducing*** on Viper has smaller effect than FlatStore. However, due to the inefficiency of Viper's original compaction, the batching compaction algorithm significantly improves Viper's performance.

4.4 Comparison with Other KV Stores

In this section, we compare FlatStore with Pacman (denoted by suffix *-Pac* in Figure 9) against other three representative KV stores on PM, *ChameleonDB* [50], *pmemkv* [6], and *PMem-RocksDB* [3, 5]. ChameleonDB adopts a LSM-based hash index tracking KV objects in the log. We implement ChameleonDB since it is not open-source. Our implementation can achieve approximate performance in its paper when not considering garbage collection. As ChameleonDB does not provide their garbage collection approach, we implement its garbage collection like WiscKey [33] for their similar structures, but with a hot-cold data separation (§3.4). pmemkv internally leverages PMDK [4] for object allocation, which is a non-copying allocator. We set pmemkv's storage engine to *cmap*, a persistent concurrent hash map. PMem-RocksDB is based on RocksDB [16], a LSM-tree-based key-value store. PMem-RocksDB locates SSTables and write ahead log (WAL) on PM. We follow the recommended configurations [5] except for enabling the key-value separation of PMem-RocksDB, since it offloads the object management to PMDK, which is similar to pmemkv. Note that we do not restrict the capacity of pmemkv and PMem-RocksDB.

We compare these systems with a production workload from Facebook ETC memcached pool [7]. Specifically, the workload has trimodal object size distribution, where the size of an object can be small (1-13 bytes), median (14-300 bytes) and large (larger than 300 bytes). This distribution is representative in real-world productions, as it also resembles the workloads of UP2X at Facebook [9]. We use skewed distribution (Zipfian parameter 0.99) for small and median objects, and uniform distribution for large objects as prior work [12, 14]. After random prefilling each system with 200 million objects, each thread performs 20 million operations of write-intensive workload (50% Get and 50% Put).

For a fair comparison, 1) we only include FlatStore-PH and FlatStore-FF as all compared systems have a persistent index; 2) we disable the log batching in FlatStore as a normal embedded KV store. All systems use 24 service threads and 4 background threads (for compaction and PMem-RocskDB's flush) except for pmemkv. As pmemkv does not need garbage collection, we set pmemkv with 28 service threads. We report their throughput and latency in Figure 9.

Throughput. From the throughput results, we have the following observations.

The log-structured approach has great performance advantages. Even without Pacman, FlatStore-PH outperforms pmemkv as unordered KV stores, and FlatStore-FF outperforms PMem-RocksDB as ordered KV stores. However, the garbage collection overhead overshadowed this advantage. Due to the efficient compaction, Pacman improves the performance of original systems, especially for FlatStore-FF as it has more severe compaction overheads.

PMem-RocksDB has the lowest throughput even though efforts have been endeavoured to optimize RocksDB with persistent memory. This is because RocksDB was born for SSD, and some design is not suitable for PM. For example, the memtable, WAL, software caching, and file-based management are less effective when the storage device changes from disk or flash to PM.

Though pmemkv does not need garbage collection, it has a low performance. Since the value size varies in this workload, pmemkv needs to allocate new object by PMDK's `pmemobj` allocator if the size exceeds previous allocated size.

ChameleonDB has much lower performance than expectation. We also evaluate ChameleonDB with unlimited log space (i.e., no compaction) that can achieve similar performance to a KV store with a volatile hash table-based index (not shown in the figure). However, the garbage collection of ChameleonDB is much difficult due to its LSM-based index. The LSM-based index inserts a new KV pair to memtable directly without looking for the former entry of the same key. Therefore, cleaners are unaware of the garbage information and have to copy bulk of cold data. Its performance decreases sharply at higher capacity utilization due to the inefficient garbage collection. ChameleonDB requires a more elaborate garbage collection approach which considers both KV separation [10, 41] and PM's peculiarities.

Latency. The median Get latency of each system accords with their index's overhead. ChameleonDB has the lowest latency due to its efficient DRAM-PM-hybird index.

For PMem-RocksDB, the notorious write stalls in LSM-trees [49] result in high Put tail-latencies. The dilatory flush and compaction not only block foreground write operations, but also lead to multiple SSTable levels, which makes Get operations inefficient. Besides, for low-latency devices like PM, bloom filters aimed at reducing storage I/Os can introduce non-negligible latencies in Get operations.

As pmemkv turns to transaction and logging for atomic in-place updates, it has the highest median Put latency and tail latency among all evaluated systems with a high thread count. First, transactions are more likely to be aborted with a high thread count. Second, the low-performance crash consistency mechanism of pmemkv makes index access inefficient.

4.5 Recovery

We evaluate the recovery with FlatStore-H and FlatStore-FF. We randomly prefill 200 million objects of 256B value size with 80% capacity utilization, in which the log space is 63.3 GB. Then we perform 100 million update operations to disorder all segments. We set 8 threads for the recovery.

For FlatStore-H, garbage information and shortcuts are restored with the volatile index in the meantime. It takes 14 seconds to recover all these things in FlatStore-H. For FlatStore-FF, it takes 102 ms to recover the states of segments. Note that the system is ready for service at this moment. Then, the 8 threads recover information for garbage collection (e.g., deleted flag bitmaps) in background, which takes 37 seconds.

5 Related Work

PM-based KV stores. There has been plenty of research on high-performance KV indexes [11, 20, 26, 27, 32, 37, 38, 43, 50, 51] and KV storage systems [8, 12, 22, 24, 30, 42, 49]. In this paper, we focus on log-structured KV stores on PM.

Log-structured memory storage systems. The log-structured design has been widely adopted in storage systems. RAMCloud [39] is a distributed KV store that uses log-structured memory to achieve high memory utilization. It uses memory to serve requests from clients and disk to store backup copies of data. FASTER [20] designs a hybrid log that spans main memory and storage. Nibble [35] is a concurrent log-structured in-memory KV store that uses a scalable multi-head log allocator with a concurrent index. It can scale up to hundreds of cores with ultra-large volumes of memory. MICA [31] and Segcache [47] are in-memory caching systems using the log-structured approach.

The log-structured approach is also embraced by many persistent memory systems. FlatStore [12], RStore [30], and Viper [8] are all DRAM-PM hybrid KV stores that leverage a volatile index on DRAM and log-structured storage on PM. To reduce small writes on PM, FlatStore proposes pipelined horizontal batching to batch small-sized requests from multiple cores, achieving high throughput without sacrificing low latency. Viper assigns threads to different PM regions to minimize the thread-to-DIMM ratio, and stores data in DIMM-aligned storage segments. NOVA [45] is a scalable persistent memory file system that maintains separate logs for each inode. LSNVMM [19] is a log-structured transactional memory system that takes advantage of copy-on-write to avoid redo/undo logging.

Optimizations on garbage collection. The main overhead of log-structured storage systems comes from garbage collection. RAMCloud designs an elaborate garbage collection approach to enable high memory utilization. RAMCloud decouples the garbage collection on the memory logs and backups. Hence, memory can have higher utilization and backup disks bear less garbage collection work. In-place updates can reduce the pressure of garbage collection [20, 31]. However, the in-place updates could lead to internal memory fragmentation. Furthermore, for persistent memory, in-place updates without expensive logging cannot guarantee crash consistency. Hot-cold separation is beneficial to garbage collection. Log-structured file systems such as F2FS [28] separate data by their types. HashKV [10] partitions KV objects by hashing. They separate hot and cold objects to some extent. Yang et al. [48] propose a PM-aware garbage collector in JVM. They separate the read-mostly phase and the write-only phase to fully utilize PM bandwidth. However, they adopt PM for merely increasing memory capacity and do not provide persistence guarantee.

The peculiarities of persistent memory make the problem of memory compaction more severe. To the best of our knowledge, PACMAN is the first work that optimizes compaction for log-structured key-value store on PM.

6 Conclusion

Garbage collection overhead in log-structured KV stores becomes more severe on PM. We summarize that the culprit is that existing approaches are unawareness of PM's characteristics. In this paper, we analyzed the inefficiencies of existing compaction approaches in log-structured KV stores on PM. According to the analysis, we design, implement, and evaluate PACMAN, an efficient PM-aware compaction approach for log-structured KV store on PM. PACMAN introduces several techniques to streamline garbage collection with the consideration of PM idiosyncrasies. PACMAN significantly boosts the compaction bandwidth and improves the performance of state-of-the-art systems at high capacity utilization. Our implementation of PACMAN is publicly available at https://github.com/thustorage/pacman.

Acknowledgements

We sincerely thank our shepherd Changwoo Min and the anonymous reviewers for their valuable feedback. We also thank Junru Li and Zhe Yang for their help on this work. This work is funded by the National Natural Science Foundation of China (Grant No. 62022051, 61832011), and Huawei.

References

[1] Aligned memory blocks (the gnu c library). https://www.gnu.org/software/libc/manual/html_node/Aligned-Memory-Blocks.html, 2021.

[2] 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.

[3] How Intel Optimized RocksDB Code for Persistent Memory with PMDK. https://software.intel.com/content/www/us/en/develop/articles/how-intel-optimized-rocksdb-code-for-persistent-memory-with-pmdk.html, 2021.

[4] Persistent Memory Development Kit. https://pmem.io/pmdk/, 2021.

[5] PMem-RocksDB, A version of RocksDB that uses persistent memory. https://github.com/pmem/pmem-rocksdb, 2021.

[6] pmemkv: Key/value datastore for persistent memory. https://pmem.io/pmemkv/, 2021.

[7] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, page 53–64, New York, NY, USA, 2012. Association for Computing Machinery.

[8] Lawrence Benson, Hendrik Makait, and Tilmann Rabl. Viper: An efficient hybrid pmem-dram key-value store. *Proceedings of the VLDB Endowment*, 14(9):1544–1556, 2021.

[9] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, Santa Clara, CA, February 2020. USENIX Association.

[10] Helen H. W. Chan, Yongkun Li, Patrick P. C. Lee, and Yinlong Xu. Hashkv: Enabling efficient updates in KV storage via hashing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 1007–1019, Boston, MA, July 2018. USENIX Association.

[11] Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, and Jiwu Shu. uTree: A Persistent B+-Tree with Low Tail Latency. *Proc. VLDB Endow.*, 13(12):2634–2648, July 2020.

[12] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. Flatstore: An efficient log-structured key-value storage engine for persistent memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1077–1091, New York, NY, USA, 2020. Association for Computing Machinery.

[13] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[14] Diego Didona and Willy Zwaenepoel. Size-aware sharding for improving tail latencies in in-memory key-value stores. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 79–94, Boston, MA, February 2019. USENIX Association.

[15] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Evolution of development priorities in key-value stores serving large-scale applications: The RocksDB experience. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 33–49. USENIX Association, February 2021.

[16] Facebook. Rocksdb. https://rocksdb.org.

[17] Swapnil Haria, Mark D. Hill, and Michael M. Swift. Mod: Minimally ordered durable datastructures for persistent memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 775–788, New York, NY, USA, 2020. Association for Computing Machinery.

[18] Jen-Wei Hsieh, Tei-Wei Kuo, and Li-Pin Chang. Efficient identification of hot data for flash memory storage systems. *ACM Trans. Storage*, 2(1):22–40, February 2006.

[19] Qingda Hu, Jinglei Ren, Anirudh Badam, Jiwu Shu, and Thomas Moscibroda. Log-structured non-volatile main memory. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 703–717, Santa Clara, CA, July 2017. USENIX Association.

[20] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 187–200, Oakland, CA, February 2018. USENIX Association.

[21] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 121–136, New York, NY, USA, 2017. Association for Computing Machinery.

[22] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young ri Choi. SLM-DB: Single-Level Key-Value Store with Persistent Memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 191–205, Boston, MA, February 2019. USENIX Association.

[23] Anuj Kalia, David Andersen, and Michael Kaminsky. Challenges and solutions for fast remote persistent memory access. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 105–119, New York, NY, USA, 2020. Association for Computing Machinery.

[24] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning LSMs for nonvolatile memory with NoveLSM. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 993–1005, Boston, MA, July 2018. USENIX Association.

[25] Jonghyeon Kim, Wonkyo Choe, and Jeongseob Ahn. Exploring the design space of page management for multi-tiered memory systems. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 715–728. USENIX Association, July 2021.

[26] Wook-Hee Kim, R. Madhava Krishnan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. PACTree: A High Performance Persistent Range Index Using PAC Guidelines. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 424–439, New York, NY, USA, 2021. Association for Computing Machinery.

[27] R. Madhava Krishnan, Wook-Hee Kim, Xinwei Fu, Sumit Kumar Monga, Hee Won Lee, Minsung Jang, Ajit Mathew, and Changwoo Min. TIPS: Making volatile index structures persistent with dram-nvmm tiering. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 773–787. USENIX Association, July 2021.

[28] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2FS: A New File System for Flash Storage. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 273–286, Santa Clara, CA, February 2015. USENIX Association.

[29] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 38–49, 2013.

[30] Lucas Lersch, Ivan Schreter, Ismail Oukid, and Wolfgang Lehner. Enabling low tail latency on multicore key-value stores. *Proc. VLDB Endow.*, 13(7):1091–1104, March 2020.

[31] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, Seattle, WA, April 2014. USENIX Association.

[32] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. Dash: Scalable Hashing on Persistent Memory. *Proc. VLDB Endow.*, 13(10):1147–1161, April 2020.

[33] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 133–148, Santa Clara, CA, February 2016. USENIX Association.

[34] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, page 183–196, New York, NY, USA, 2012. Association for Computing Machinery.

[35] Alexander Merritt, Ada Gavrilovska, Yuan Chen, and Dejan Milojicic. Concurrent log-structured memory for many-core key-value stores. *Proc. VLDB Endow.*, 11(4):458–471, December 2017.

[36] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. SFS: Random write considered harmful in solid state drives. In *10th USENIX Conference on File and Storage Technologies (FAST 12)*, San Jose, CA, February 2012. USENIX Association.

[37] Moohyeon Nam, Hokeun Cha, Young ri Choi, Sam H. Noh, and Beomseok Nam. Write-Optimized Dynamic Hashing for Persistent Memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 31–44, Boston, MA, February 2019. USENIX Association.

[38] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 371–386, New York, NY, USA, 2016. Association for Computing Machinery.

[39] Stephen M. Rumble, Ankita Kejriwal, and John Ousterhout. Log-structured memory for dram-based storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST'14, page 1–16, USA, 2014. USENIX Association.

[40] Denis Serenyi. Cluster-level storage at google. In *Keynote at the 2nd Joint International Workshop on Parallel Data Storage and Data Intensive Scalable Intensive Computing Systems*, 2017.

[41] Chen Shen, Youyou Lu, Fei Li, Weidong Liu, and Jiwu Shu. Novkv: Efficient garbage collection for key-value separated lsm-stores. 2020.

[42] Jiwu Shu, Youmin Chen, Qing Wang, Bohong Zhu, Junru Li, and Youyou Lu. TH-DPMS: Design and Implementation of an RDMA-Enabled Distributed Persistent Memory Storage System. *ACM Trans. Storage*, 16(4), oct 2020.

[43] Qing Wang, Youyou Lu, Junru Li, and Jiwu Shu. Nap: A Black-Box approach to NUMA-Aware persistent memory indexes. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 93–111. USENIX Association, July 2021.

[44] Xingbo Wu, Fan Ni, Li Zhang, Yandong Wang, Yufei Ren, Michel Hack, Zili Shao, and Song Jiang. Nvmcached: An nvm-based key-value cache. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[45] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, Santa Clara, CA, February 2016. USENIX Association.

[46] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 169–182, Santa Clara, CA, February 2020. USENIX Association.

[47] Juncheng Yang, Yao Yue, and Rashmi Vinayak. Segcache: a memory-efficient and scalable in-memory key-value cache for small objects. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 503–518. USENIX Association, April 2021.

[48] Yanfei Yang, Mingyu Wu, Haibo Chen, and Binyu Zang. *Bridging the Performance Gap for Copy-Based Garbage Collectors atop Non-Volatile Memory*, page 343–358. Association for Computing Machinery, New York, NY, USA, 2021.

[49] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. MatrixKV: Reducing write stalls and write amplification in LSM-tree based KV stores with matrix container in NVM. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 17–31. USENIX Association, July 2020.

[50] Wenhui Zhang, Xingsheng Zhao, Song Jiang, and Hong Jiang. Chameleondb: A key-value store for optane persistent memory. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 194–209, New York, NY, USA, 2021. Association for Computing Machinery.

[51] Pengfei Zuo, Yu Hua, and Jie Wu. Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 461–476, Carlsbad, CA, October 2018. USENIX Association.