Developing Index Structures in Persistent Memory Using Spot-on Optimizations with DRAM

Xingsheng Zhao[*]
xingsheng.zhao@mavs.uta.edu
University of Texas at Arlington
Arlington, Texas, USA

Prajwal Challa
vxc5208@mavs.uta.edu
University of Texas at Arlington
Arlington, Texas, USA

Chen Zhong
chen.zhong@mavs.uta.edu
University of Texas at Arlington
Arlington, Texas, USA

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

ABSTRACT

The emergence of persistent memory (PMem) is greatly impacting the design of commonly used data structures to obtain the full benefit from the new technology. Compared to the DRAM, PMem's larger capacity and lower cost make it an attractive alternative for hosting large data structures, such as indexes of in-memory databases, especially for those that require data persistency. However, simply using existing index structures in the PMem can be unexpectedly inefficient for three reasons. (1) Index accesses are composed of small writes and reads. (2) Each small write is required to come with expensive fence and flush operations. And (3) PMems usually prefer large accesses for high performance with their internal block-like access designs despite being byte-addressable. For example, Intel Optane DC PMem has a 256-byte access unit (XPLine), leading to significant read/write amplification for small accesses.

In this work we systematically study a series of techniques, including application-managed write-buffering, read-caching, and out-of-place updates and their synergistic effect on performance of some representative indexes (hash table, B+ tree, and skip list) designed for PMems. We then apply the knowledge obtained from this investigation into the design of a high-performance PMem index, named Spot-on tree (SPTree), that facilitates applications to selectively cache read-intensive components of an index and to buffer written data to index structure, while providing crash consistency and quick recovery upon crash. Compared to the state-of-art indexes, SPTree provides up to 2X and 4X higher write and read throughput, respectively.

CCS CONCEPTS

• **Information systems** → **Data access methods**; • **Theory of computation** → **Concurrent algorithms**; • **Hardware** → **Non-volatile memory**.

[*]now at Google

KEYWORDS

B-tree, hash table, non-volatile memory

ACM Reference Format:
Xingsheng Zhao, Prajwal Challa, Chen Zhong, and Song Jiang. 2024. Developing Index Structures in Persistent Memory Using Spot-on Optimizations with DRAM. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE '24), May 7–11, 2024, London, United Kingdom.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3629526.3645032

1 INTRODUCTION

The memory/storage hierarchy, which consists of multiple levels including CPU cache, DRAM, and block devices, such as SSDs and HDDs, has been stabilized for decades. Accordingly, the principal management designs for data across its levels, such as set-associative CPU caches, page-based virtual memory, and block-based read cache and write-back buffer, are well established by carefully considering individual devices' performance characteristics to maximize the hierarchy's performance. However, with the emergence of byte-addressable persistent memory (PMem), such as Intel Optane DC persistent memory [1] and NVDIMM (with flash storage) [4, 5], a new level/tier was introduced into the hierarchy. We contend that it is necessary for the DRAM to serve as a cache level for the PMem to boost its effective performance. And it is a challenging task for accessing of data structures on the PMem to be accelerated with an efficient use of DRAM.

1.1 DRAM as a Cache of PMem

Like DRAM, the PMem is a byte-addressable memory device that can be directly accessed via load and store instructions. However, its performance gap with the DRAM can be still substantial. As the Optane PMem has a more consistent performance behavior, we use it as a representative of PMem hereafter. The performance of the Optane PMem is lower than that of DRAM by 2-3X or more in terms of its latency and throughput [12]. In the meantime, the PMem can have a 5-10x increase of per-module capacity over the DDR4 DRAM while its per-GB price is 2X-5X cheaper than DRAM. Therefore, placing the PMem underneath the DRAM in the hierarchy has the potential of taking advantage of both DRAM's high performance and PMem's larger capacity and lower cost. Indeed, the Optane PMem has a memory mode in which DRAM acts as a cache for data accessed on the Pmem. Though few details are known on how Intel

CPU's IMC (Integrated Memory Controller) enables this transparent caching, a constraint in the use of the mode highlights the challenge in the management of this memory level. The constraint is that the persistent memory has to be treated as a volatile memory, or data in the memory cannot survive a system restart, though the capability of retaining data on the PMem is one of its major features that are attractive to many potential users.

1.2 DRAM as a Write Buffer of PMem

When DRAM is used as PMem's cache, it not only should be used as a read cache, but also must be used as a write buffer to enable the write-back policy for three reasons. First, DRAM is faster than the PMem by around 2-3X for reads and is around 4-6X for writes. Second, recent studies have shown that the Optane PMem has an access unit of 256 bytes to the memory's media (in comparison, NVDIMM with flash has a 0.5∼4KB page-size access unit). Any write smaller than the size leads to a write amplification and reduction of effective throughput [3]. For example, with 64-byte random writes the PMem's throughput is reduced to about 1/4 of its peak one [24]. Third, to ensure crash consistency for written data, an application may have to frequently use expensive fence and flush instructions between writes to the PMem, which may significantly degrade write efficiency. DRAM can be used as a write buffer to coalesce writes and then flush data in the buffer to the PMem. By doing so multiple random writes to the PMem may be transformed into one big sequential write that aligns with PMem's access granularity so as to receive high throughput from the PMem. It is tempting to use the large DRAM space to make up for the PMem's shortcoming.

However, it is difficult to retain the PMem's persistence feature by simply using a DRAM as its cache/buffer. Some fundamental challenges exist due to some unique characteristics of the DRAM-PMem layers. Unlike CPU-caches/DRAM layers, the DRAM as a cache for the PMem presents unique challenges. This is primarily due to the fact that the CPU cache is much smaller than the DRAM and can be managed with an affordable cost with hardware supports. Instead, the gap between sizes of DRAM and the PMem is much less significant. Furthermore, the CPU cache can be battery/capacitor protected to keep dirty data in it from being lost upon a power failure. But the large DRAM cache/buffer is unlikely to have such a support. This presents a dilemma about the use of DRAM as a cache/buffer for the PMem. Using DRAM as the PMem's buffer enables a large write-back space while simultaneously presents a risk of losing data due to DRAM's non-volatile nature.

Long before emergence of the commercially available PMems, the advantages of non-volatile memory (NVM) have been recognized. Significant efforts have been made to migrate popular in-DRAM data structures to the NVM with optimizations of their crash-consistent implementations by efficiently using fence and flush operations [10, 13, 18, 25]. Among the data structures, indexes, such as hash table and B+ tree, are the most performance-critical due to their frequent and on-the-critical-path accesses. In the meantime, they are vulnerable to performance loss due to their frequent use of pointers and accessing of small data. One of their major use cases is the development of key-value (KV) stores in an NVM with the objectives of high write throughput, low read latency, low DRAM footprint, and rapid recovery and restart after a system crash. An unexploited opportunity in the efforts is to leverage DRAM to conduct spot-on caching and buffering for individual components of an index where the performance gap is large. Unlike system-level caching and buffering services that are indiscriminately covering the entire DRAM and NVM levels with a fixed space unit (e.g., disk block), this proposed spot-on approach is customized to the structure of an index. Therefore, performance-limiting operations, such as pointer chasing and random writes, in the PMem can be made efficient with dedicated caches/buffers for individual components such as inner nodes and leaf nodes in a tree structure.

In this paper, we made several important contributions on the improvement of major data structures' performance in the PMem.

- We identify unique issues and opportunities in the efforts of bridging the performance gap between DRAM and the PMem in the memory hierarchy.
- We propose and evaluate a comprehensive and complementary set of techniques to enable DRAM's spot-on caching and buffering for the PMem.
- Using the techniques we design and implement a new index structure, Spot-on Tree (SPTree), that can efficiently leverage limited DRAM space to reduce read and write amplifications and the cost for maintaining crash-consistency in the PMem.
- We introduce a unique design of the write-ahead-logging (WAL) technique to enable instant service resumptions.
- Experiment results show that SPTree can achieve up to more than 4X throughput and less than 1/5 latency over some of the state-of-the-art indexes, such as Fast&Fair and PACTree.

2 BACKGROUND AND RELATED WORK

In this section, we describe the background of Intel Optane Persistent memory, which is selected in this study as the representative of the PMem technology, and some related works on persistent indexes to motivate this study.

2.1 The PMem

The Optane Persistent memory can be configured in two different modes. The first one is named `Memory Mode`, in which the CPU considers the Optane as its main memory and uses the entire DRAM as its cache. While the cache is so large and the caching unit is page, it would be too expensive and thus infeasible to keep dirty pages in the DRAM persistent. Therefore, in this mode the Optane PMem does not provide persistency at all. And the PMem becomes essentially a larger but slower DRAM. The second mode is the App-Direct mode. In this mode, the PMem works as a persistent device. A file system supporting Direct Access (DAX) provides direct access to the persistent memory, and bypasses the file system block I/O. In this mode, a program is exposed to the aggregate space of the DRAM and the PMem as well as their distinct performance characteristics.

Though the PMem is byte-addressable, the physical media access granularity is 256 bytes (XPLine) [3, 23, 24]. Any non-contiguous writes of data smaller than the XPLine size requires a read-modify-write operation, leading to write amplification and reduced effective memory bandwidth. To reduce write amplification, the PMem employs a write-combining buffer to merge adjacent small writes. However, its size is only 16KB [3, 22]. Writes to memory addresses with a coverage larger than this scope cannot benefit from the feature. It is unknown how to flexibly and dynamically set up buffers in the DRAM to overcome this PMem's limitation on write performance. Similarly, the CPU cache is often not sufficient to improve its read performance on par with that of DRAM because of the PMem's large size and programs' weak access locality.

2.2 Persistent Indexes

There are mainly two kinds of persistent indexes. One is persistent hash table, such as CCEH[18], Level hashing[25], and Dash[16]. And the other is persistent range indexes whose keys are sorted and support range search, including FastFair[10], FP-Tree[9], and PACTree [13].

Persistent Hash Table. Being aware of higher write cost in a persistent memory, existing works on development of persistent hash tables mostly focus on reducing number of writes in the memory, such as PFHT [6], level hashing [25], and CCEH [18]. In particular, PFHT is a cuckoo hashing variant that limits number of displacements to only one to reduce memory writes during service of a write request. Level hashing adopts a two-level hash scheme so that each key can have three buckets as the candidates for its insertion, which helps reduce key relocations in the table and improve the load factor. CCEH is an extendable hashing to minimize performance impact of rehashing of the entire table. It organizes buckets into 64KB segments to leverage fast sequential access in a persistent memory. It then uses the linear probing strategy in its search for either a key or a bucket with empty slot(s). A successful insertion requires only one memory write. While these works reduce number of writes in the memory, each write takes place directly at the memory's location determined by the hash table design. While the locations are spread out in the memory, the performance loss due to write amplification in the PMem can be significant.

Persistent Range Indexes. There have been some works on designs of B-trees for persistent memory. FastFair[10] is a lock-free-read B+-tree that avoids expensive copy-on-write and logging to tolerate transient inconsistency. BzTree[2] relies on the Persistent Multi-word Compare-And-Swap(PMwCAS) primitive to implement a lock-free tree. FP-Tree[9] stores inner nodes of the tree in the DRAM to achieve high performance. However, it has to scan all nodes on the persistent memory to reconstruct the inner nodes after a reboot or a crash before resuming its service. PACTree [13] employs a persistent trie index as its internal nodes and asynchronously updates the internal nodes using a structural-modification-operation (SMO) log. In these works, the DRAM is not leveraged to buffer writes to accommodate the PMem's block access unit. Even if reads are accelerated by partitioning an index structure between the DRAM and the PMem, access of in-PMem leaf nodes, where a majority of data is stored, is not accelerated. Furthermore, the in-DRAM sub-structure has to be entirely rebuilt during a recovery process before new accesses can be resumed. This compromises a promise made for the persistent memory, which is the instant service resumption.

3 THE THREE SPOT-ON TECHNIQUES

In this section, we analyze three representative persistent index data structures and propose a series of techniques leveraging DRAM to optimize the PMem performance in a spot-on fashion. Three data structure are CCEH[18] (a persistent hash table), FastFair [10] (a persistent B-tree) and P-Skiplist (a persistent skiplist) [20], as shown in Figure 1. The set of techniques are *Buffering*, *Out-of-place-update*, and *Caching*. They are intended to be applied in a sequence, which is the *Buffered, Out-of-place merging, and then Caching (BOC)* design.

While each of the techniques is a well known one and has been extensively practiced, this work focuses on their customized use on specific index structures in a spot-on manner. Rather than keeping and managing the cached/buffered data at one centralized space in the faster memory, the BOC approach distributes the space to the carefully selected index components to maximize its utilization. It doesn't rely on a replacement algorithm to determine where the cache space should be allocated. While effectiveness of a replacement algorithm is often limited and the algorithm's time and space overhead for managing small pieces of data can be significant, the BOC approach effectively addresses this issue with its customized cache space distribution design.

The key BOC takeaways are that, the read-modify-write in a 256-byte XPLine and long latency of random access in the PMem are the fundamental performance bottleneck for persistent index data structures. It suggests that a persistent index should (1) buffer small writes in the DRAM then update them to the PMEM in a batch manner, and use (2) out-of-place merging to reduce flush&fence, which is a well-understood performance bottleneck. When there are frequent pointer chasing accesses in the indexes, one should (3) cache the search path in the DRAM to minimize the search latency. This represents a holistic design approach in recognition that any of the individual techniques could not adequately recoup PMem's performance loss due to its performance idiosyncrasies.

3.1 Technique One: Buffering

The PMem differs from the DRAM in several ways. One of them is that there is a mismatch between CPU cache-line access granularity (64 bytes) and the 3D-Xpoint media's access granularity (256 bytes) [23]. To overcome this mismatch, it has a write-combining buffer (16KB) to merge small writes and reduce write amplification [23]. Given the small size, it is hard to exploit the locality to frequently hit the buffer.

For writes in the persistent index data structures, most of them are small writes, such as insertion of a new record (e.g., a 16-byte key-value pair), structural modification operations (SMO) in a B+-tree, which lacks access locality considering the small write buffer in the PMem. Consequently, most of the small writes result in the read-modify-write operations within the PMem and leads to high write amplifications and reduced effective memory bandwidth.

In this study of the spot-on buffering technique, we allocate small write buffers in the DRAM and assign them individually to selected components in the index structures (see Figure 1). Specifically, for CCEH each write buffer is as large as half of the segment size. For FastFair each leaf node has a write buffer of half of the leaf node size. P-Skiplist only creates write buffers for nodes whose height is higher than two in the skip list. And each buffer can store at most 16 KV pairs. The operations in the indexes are not changed except that writes into the selected components are first admitted to their respective write buffers. When a buffer becomes full, all key-value

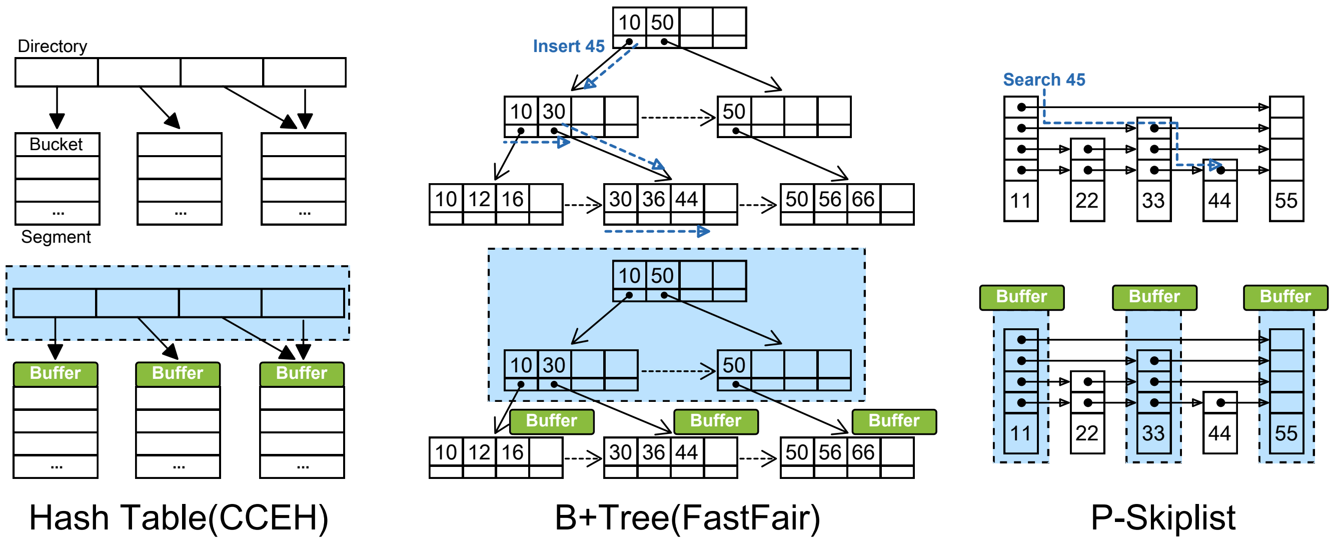


Figure 1: Adding write buffers to the three selected index data structures

(KV) pairs in it are written to its corresponding index component in a batch. This is actually a merging of KV pairs in the buffer for a batched write.

We then run a benchmark to insert 120 million KV pairs (8-byte key and 8-byte value) into each of the indexes with uniformly distributed and non-redundant keys. In the experiment we measure amount of raw data written to the PMem's media using the ipmctl tool [11]. Figure 2 shows the amount of the raw data with or without the write buffers for each of the indexes. It shows that the total amount of the data written to the PMem is reduced by up to four times. The reductions are especially higher for CCEH and FastFair where writes to the last-level nodes are buffered. This observation demonstrates that spot-on buffering can effectively enable batched writes and much reduced write amplification by greatly improving spatial access locality.

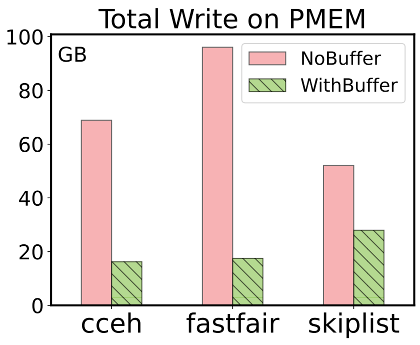


Figure 2: Total amount of raw data written to the PMem's media after insertion of 120 million 16-byte key-value pairs to the three indexes with and without the buffers (with one thread and without using the WAL log).

3.2 Technique Two: Out-of-place Update

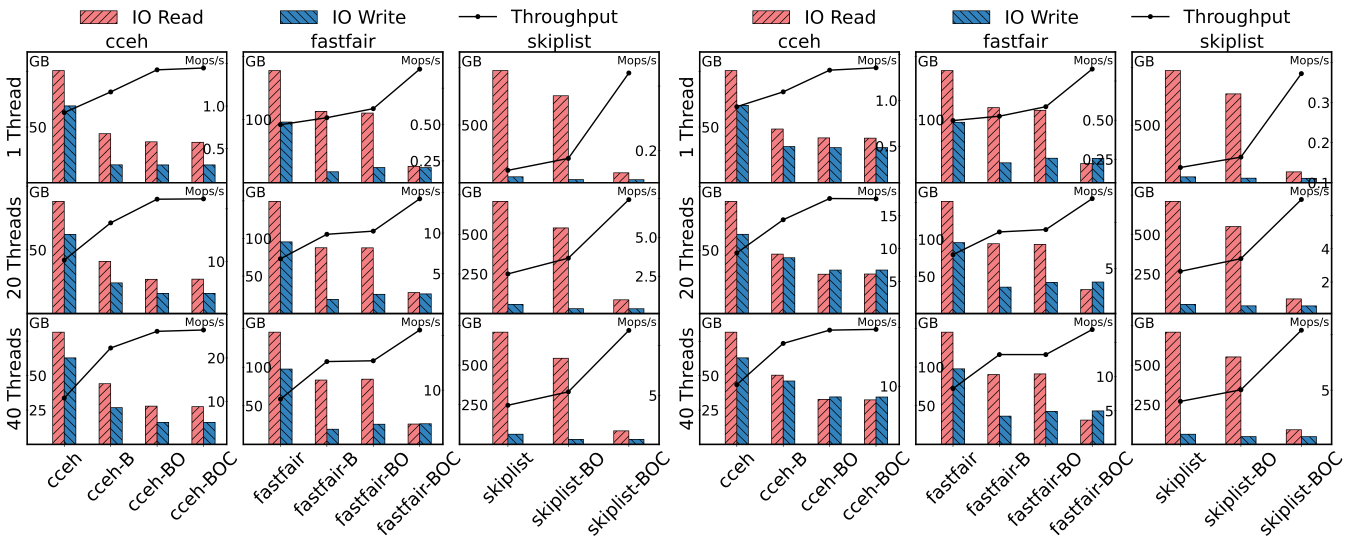
There is a potential issue with adding write buffers to the persistent indexes. When we write the KV pairs in a buffer to the PMem in an in-place manner (i.e., updating the index component in place), we may need to repeatedly add flush&fence within a small contiguous range of the PMem space (such as a segment in CCEH, or a leaf node in FastFair) to enforce its crash consistency in case of power failure. It has been reported that reading a recently flushed cacheline after fence instructions could experience much higher latency as the read has to wait the flush to complete [23].

To analyze the effect of flush&fence on the buffer merging, we implement an out-of-place buffer merge in CCEH and FastFair. Since P-Skiplist's KV pairs are stored separately in the linked list nodes, the insertion has been in the out-of-place manner. So we do not consider P-Skiplist in this experiment.

In CCEH, when a write buffer of a segment is full, we first copy the segment from the PMem to the DRAM. We then merge the KV pairs in the write buffer into the DRAM copy and write it back to the PMem in a newly allocated space. Finally, we atomically change the directory pointer to this new segment. For FastFair, we employ a similar approach for its leaf nodes. During the merging, a new leaf node is created to hold all of the KV pairs from both the old leaf node and write buffer. Then the new leaf replaces the old leaf by atomically changing the parent pointer and sibling's left 'next' pointer. An out-of-place merge is essentially conducted in the background. Thus, flush&fence operations for individual KV pairs are avoided. Another benefit of this technique is that the current buffer and the index component are still available for serving read requests during the merge.

Figure 3 shows the amount of raw data read from and raw data written to the PMem media with each of the three indexes when 120 million KV pairs are inserted. It also shows the insert throughput. Different number of threads are used (1, 20, and 40 threads at the top, middle, and bottom, respectively, of the figure). For a particular index, the three optimization techniques are incrementally applied.

The figure shows that even when the read and write amounts remain unchanged, the throughput of CCEH and FastFair with only



(a) Without WAL **(b) With WAL**

Figure 3: Amount of raw media read/write (denoted as "I/O Read/Write") and throughput for inserting 120 million key-value pairs (8-byte key and 8-byte value) with and without using a Write-Ahead-Log (WAL) log. For each index, the three techniques are incrementally added ('B' for "Buffering", 'O' for "Out-of-place Update", and 'C' for "Caching"). For example, "cceh", "cceh-B", "cceh-BO", and "cceh-BOC" refer to the CCEH index without any optimizations, with Technique 1, with Techniques 1 and 2, and with Techniques 1, 2, and 3, respectively.

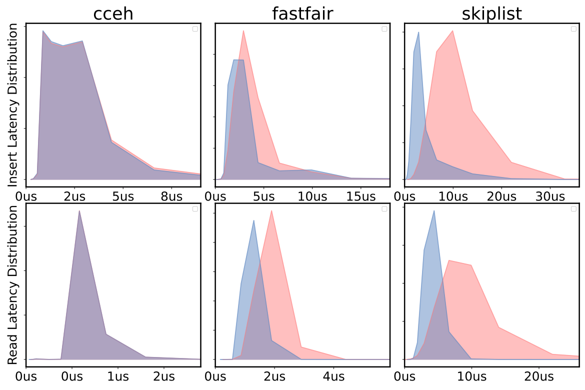


Figure 4: Distributions of insert/read latency for an index without caching (the red area) and with caching (the purple area).

Buffering ("cceh-B" and "fastfair-B") becomes higher after applying "Out-of-place Update" ("cceh-BO" and "fastfair-BO"). This improvement is attributed to the removal of the flush&fence overheads.

3.3 Technique Three: Caching

For index data structures whose operations contain many random reads (in the form of pointer chasing), the performance could be bottlenecked by slow random reads in the PMem. Read latency on the Optane PMem is considerably higher than DRAM (about 2X-3X) because reads need to fetch data from the 3D-Xpoint media, which has longer media latency [3]. Meanwhile, most of the pointer chasing operations happen in the internal nodes whose total size may account for only a small portion of the entire data structure size. So it is worth enabling caching individually for the internal nodes in the DRAM to boost the lookup performance.

In this experiment study, we cache internal nodes in all three indexes (directory in CCEH, inner nodes of FastFair, nodes with height higher than two in P-Skiplist) and measure their insert/read latency for 120 million insert requests followed by 120 million read requests.

As shown in Figure 4, caching internal nodes have little effect on CCEH's latency as it has only one pointer chasing operation for each access, which is for locating the segment from the directory. However, we observe significant latency improvement for FastFair and P-Skiplist as they need to do intensive pointer chasing (random reads) across the internal nodes before finding a target node, indicating that caching is a necessary optimization for many-level indexes.

3.4 Put them Together

With all these three techniques available (*buffering, out-of-placed Update, and then caching(BOC)*), we apply them to three indexes one by one and measure the performance improvement after insertion of 120 million key-value pairs. In the experiment, we experiment with two cases: one without the WAL log, and another one with the WAL log. With the log, all data written to the leaf nodes will also be added to the log in the PMem. Note that writes to the log are fully sequential and incur little amplification. As shown in Figure 3, the write buffer not only reduces write amplification but also read amplification. This is not a surprise since during the buffer merging, the PMem data to be merged has been cached in the DRAM. The cost of reads is amortized via batching. Meanwhile, the benefits of out-of-place update (merging) are consistent from the single thread execution to the multi-thread execution. The throughput with the addition of "Out-of-place Update" ("cceh-BO" and "fastfair-BO") is always higher than that without the optimization ("cceh-B" and "fastfair-B", respectively). Finally, caching the internal nodes not only reduces latency of pointer chasing, but also removes a large amount of the PMem's media read ("Read I/O") during search ("fastfair-BOC", "skiplist-BOC"), significantly increasing the PMem's effective memory bandwidth. Furthermore, the performance advantages remain with the increase of thread count and with addition of the WAL log.

4 THE DESIGN OF SPTREE

Understanding the benefits of the three spot-on optimization techniques, we propose SPTree (SPot-on Tree), a DRAM-PMem hybrid persistent tree index that uses spot-on caching and buffering to address a sequence of issues challenging performance of in-PMem indexes, including long latency of pointer chasing, write amplification due to mismatch between KV pair size and PMem's media access unit, and quick recovery after a crash. As shown in Figure 5, SPTree consists of three layers (the top, middle, and bottom layers).

4.1 The Three Layers in SPTree

The Top Layer. As shown in Figure 5, the top layer caches the internal nodes of the tree in the DRAM, which is an in-DRAM index that walks a key to its corresponding middle level node and leaf node. SPTree uses the DRAM index to address the high latency issue of pointer chasing in the PMem. That is, all search operations in the SPTree take place in the DRAM top layer before a target leaf node in the bottom layer is reached. We modify the ARTree [14] as the DRAM top-layer. Meanwhile, the index has a PMem backup, organized as a FastFair B+-tree, for a quick recovery after a reboot or a system crash. It is updated asynchronously by the background threads to move the slow updates on the PMem off the critical path. Each time when a leaf node splits or merges, the new leaf node's indexing information is synchronously updated in the DRAM index, and sent to the background threads and then asynchronously propagated to the in-PMem index. If some of the updates have not yet been reflected in the PMem index when a system crash happens, we can still recover the missing information by checking the possible smallest key(the low ley or 'lkey') and the largest key(high key or 'hkey') in the leaf nodes. More details on the recovery are in Section 4.6. Note that updates to the in-PMem index are much less frequent than those to the leaf nodes.

The Middle Layer. The middle layer consists of Mnodes (middle-layer nodes) in the DRAM. Each leaf node in the PMem has its corresponding Mnode in the DRAM. The Mnode records the range of keys in the leaf node (the smallest key (lkey) ... largest key(hkey)). This key range serves the purpose similar to that of the version number that allows non-blocking reads during a write. When a search reaches to an Mnode, the search key is checked against its key range. If the search key does not belong to this Mnode, it indicates that concurrent insertion or deletion operations have caused splitting or merging of Mnodes (and also leaf nodes), and we need to go back to the top layer for a retry.

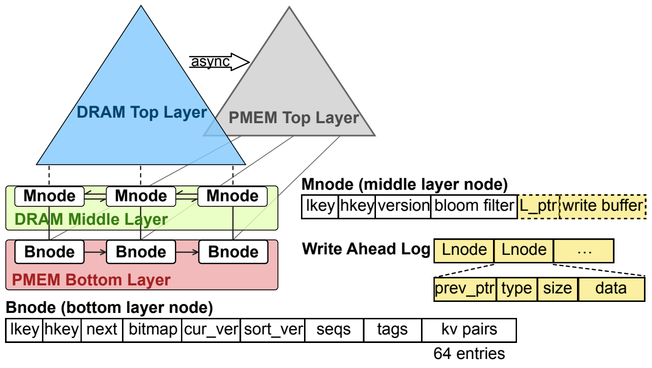

async
DRAM Top Layer
PMEM Top Layer
Mnode
Mnode
Mnode
DRAM Middle Layer
Bnode
Bnode
Bnode
PMEM Bottom Layer
Mnode (middle layer node)
lkey | hkey | version | bloom filter | L_ptr | write buffer
Write Ahead Log
Lnode | Lnode | …
prev_ptr | type | size | data
Bnode (bottom layer node)
lkey | hkey | next | bitmap | cur_ver | sort_ver | seqs | tags | kv pairs
64 entries

Figure 5: The SPTree tree structure

An Mnode also stores a Bloom filter which remembers all the keys in its leaf node. Each insert updates the Bloom filter in its corresponding Mnode. The Bloom filter is used to filter out most of the point queries for non-existing keys before they reach the in-PMem leaf nodes. An Mnode can also be configured with a write buffer, which buffers all the newly inserted KV pairs (up to 14) to reduce write amplification. Once a buffer is full, all the KV pairs in the buffer will be merged to the leaf node.

The Bottom Layer. The bottom layer stores the leaf nodes, or Bnodes (bottom layer nodes), in the PMem. All Bnodes are organized as a singly linked-list. Each Bnode can store up to 64 KV pairs. The leaf node groups fingerprints (hash values) of its keys as key tags in an array for a quick preliminary search by using SIMD instruction (`_mm_cmpeq_epi8_mask`). Meanwhile, it applies a two-phase insertion approach to reduce the use of flush and fence instructions. In the first phase, KV pairs and their tags are batch written to the Bnode followed by a flush&fence. Only in the second phase will they be validated by setting the bitmap in the node followed with another flush&fence. In this way, only two flush&fence operations are needed for multiple insertions. In this way, SPTree minimizes the use of flush&fence to improve efficiency.

4.2 Concurrency Control

SPTree relies on `Optimistic Lock Coupling` [15] for concurrency control. An optimistic lock consists of a lock and a version counter (packed into 8 bytes). For writers, the optimistic lock provides exclusive access that allows only one writer at a time. Upon an unlock, the lock is released and the version number is incremented by one atomically. For readers, they do not acquire the lock. Instead, they wait until the write lock is freed. Then they compare the version number before and after reading the value. If the reading changes, they will retry until a consistent reading is reached. Both of the DRAM top layer and the middle layer use the optimistic lock.

4.3 Search Operation

Search for a given key is the most frequently used operation on the index. It is not only used to service a read request, but also has to be employed at the beginning of the execution of every insert/update/delete/lookup/scan operation. A search operation first traverses the DRAM top layer to look for the largest key that is smaller or equal to the search key. This largest key is the lkey in an Mnode. Then the search key is compared with the hkey in the Mnode to ensure the search key is in the key range of this Mnode. If it is in the range, the search continues to the Bnode in the PMem. Otherwise, the search travels in the middle layer, which is a doubly linked list, to locate the target Mnode. During the search operation, the PMem top layer is not accessed because it is only used for recovery.

4.4 Insert/Update/Delete/Lookup/Scan

Insert. The thread servicing an insert request first searches the DRAM top layer to locate an Mnode whose key range includes the inserted key. Then the write lock is acquired on the Mnode. If the Mnode has a write buffer, the KV pair is inserted into the write buffer, and also appended to a write ahead log (WAL) for an after-crash recovery. Otherwise, the thread inserts the KV pair directly into the corresponding Bnode on the PMem. In the first case, if the write buffer is full, SPTree writes all the KV pairs in a batch to the Bnode, and clears the write buffer. SPTree uses a two-phase insertion in the batch write operation. (1) All the KV pairs and their tags are written to the Bnode followed by a flush&fence. (2) The bitmap indicating valid pairs is set and the version (*cur_ver*) advances by one. Meanwhile, a split is conducted if necessary. Finally, the Bloom filter in the Mnode is updated for the inserted key.

Update. Update is similar to insert during its search for the target Mnode. After a target Mnode is found, the Bloom filter is checked to see if the key exists. If not, it returns immediately. Otherwise, it acquires the write lock. If there is a write buffer with the target Mnode, it tries to update the KV in the buffer, and write the new value to the WAL if the update is a success. Otherwise, we check the corresponding Bnode for the update.

Delete. For delete, after the thread reaches a target Mnode, it checks the Bloom filter first to avoid unnecessary access to the PMem. If the Bloom filter returns true, and the target Mnode has a write buffer, it removes the delete key in it if it exists in the buffer. Additionally, it adds a DELETE tombstone record in the write buffer. The DELETE tombstone is also written to the WAL log to record this deletion. In this case, the key will be physically deleted later during a compaction when a KV pairs in the write buffer are written back. Note that this tombstone is necessary to prevent follow-up reads for this key from mistakenly returning the deleted KV pair from the corresponding Bnode instead of a "none". If the Mnode doesn't have a write buffer, it will immediately try to remove the key from the Bnode.

Lookup. Lookup operation in SPTree is a lock-free read. By applying the optimistic lock, the thread checks the version in the target Mnode before and after the read. If the two versions match, it means no inserts take place during the read and it can return the value safely. Otherwise, the lookup will start over.

Scan. Scan operation first looks for the target Mnode whose 'lkey' is equal to or smaller than the lower bound of the scan range and whose 'hkey' is larger than the lower bound. It then checks whether a reconstruction of the sequence array (`'seqs'`) is required by comparing `'cur_ver'` and `'sort_ver'`. If `'cur_ver'` is larger than `'sort_ver'`, it means new inserts happen after the last reconstruction of the `'seqs'`, it then acquires the write lock and sorts the keys in the Bnode and stores them in the ascending order in the `'seqs'` array. Meanwhile, it sets `'sort_ver'` to `'cur_ver'` to indicate this Bnode is ready for scan. When there are keys stored in the Mnode write buffer, a batch write of the KV pairs is triggered before scanning. During the scan of a Bnode, the keys that fall in the scan range are collected in a buffer. Then the values of `'cur_ver'` before and after the scan are compared. If they are equal, those buffered KV pairs are appended in the output array. Otherwise, it retries the scan in this Bnode. Scan continues to the sibling Bnode until the output array is full or the key is out of scan range. The operation's pseudocode description is at Algorithm 1.

Algorithm 1: Scan

```
Function Scan(start_key, result_size, results):
    /* Find mnode that has lkey <= start_key <= hkey */
    mnode = LocateMNode(start_key)
    bnode = mnode.bnode
    while Output results array not full do
        ReadLockGuard(bnode)
        if mnode write buffer is not empty then
            WriteLockGuard(bnode)
            Flush write buffer;
        if bnode.cur_ver > bnode.sort_ver then
            WriteLockGuard(bnode)
            Sort(bnode)
            bnode.sort_ver = bnode.cur_ver
        Collect keys falling in scan range
        bnode = bnode.next
```

4.5 Split and Merge

When a Bnode (leaf node) is full, SPTree conducts a split, which creates a new Bnode along with its new Mnode. Then a [`lkey, Mnode pointer`] mapping is inserted to the DRAM top layer while a [`lkey, Bnode pointer`] is sent to the background thread to asynchronously update the PMem top layer. A split operation involves four steps. (1) The writer first acquires the lock for the current Bnode and its next sibling. (2) It then allocates a new Bnode, moves the right half of the KV pairs from the full node to the new Bnode, and sets the `'next'` pointer in the full node pointing to the new node. These three operations are atomically conducted using the leak-free PMem allocator (such as Intel PMDK's `pmemobj_alloc()`) to prevent memory leak. (3) Then `'bitmap'` and `'hkey'` in the full Bnode are modified to remove the split-out keys. And (4) finally, the mapping information is updated in the DRAM top layer and propagated to the PMem top layer.

When a delete operation detects that keys in two adjacent nodes become fewer than half of a node's capacity, a merge operation is triggered. (1) The thread first acquires the two nodes' write locks. (2) Then it shifts the KV pairs in the right Bnode to the left Bnode using the two-phase insertion (set the KV pairs first, then the bitmap). (3) Next it modifies the left Bnode's hkey as the right Bnode's hkey, marks the right Bnode as deleted in its `'cur_ver'` and drops the Bnode. (4) Finally, it updates the mapping in the top layer.

4.6 Recovery and Crash Consistency

Since SPTree uses the DRAM top layer and the DRAM middle layer to provide service, these two layers need to be reconstructed after a reboot or a power failure. In a normal reboot, all the updates in the

DRAM top layer have been propagated to the PMem top layer. And the PMem top layer stores the pointers to the Bnodes in the PMem. During a recovery, SPTree can quickly collect all the pointers to the Bnodes by scanning the PMem top layer, which is only around 2% of entire index size. Then it rebuilds the Bnode's Mnode and the DRAM top layer simultaneously.

If this recovery is after an unexpected crash, we need to fix the incomplete state. If a crash happens before a split's Step (2) completes, SPTree can recover to the state before the split (guaranteed by the leak-free allocator). If the crash happens after Step 2 and before Step 3, then a dummy leaf node is linked to the bottom layer. An example incomplete status looks like this: [`split node, lkey..10, hkey..100`] -> [`dummy node, lkey..50, hkey..100`] -> [`next node, lkey..100, hkey..200`]... . The dummy node will have a `'lkey'` lower than its previous node's `'hkey'`. If we find a dummy node during the recovery, we fix it by dropping the dummy node. If the crash happens during Step (3) (`'hkey'` is set and the `'bitmap'` has not been set), then the split node will have keys that do not belong to it. This can be fixed by ignoring those out-of-range keys during the next split. If the crash happens before Step (4) completes, we only need to fix the missing mappings in the top layer. We can identify the missing mappings by comparing the adjacent Mnode's `'lkey'` and `'hkey'` after a non-crash recovery. An Mnode's `'hkey'` is designed to be the same as its next sibling's `'lkey'`. If not, we scan the linked list from the current Mnode's Bnode to recover the missing Mnodes.

If a crash happens before a merge's Step (2) competes, SPTree can recover to the state before the merge because the shifted KV pairs have not been exposed by the bitmap. If the crash happens before Step (3), the shifted KV pairs can be filtered out using the `'hkey'`. If the crash is after setting the `'hkey'` and before dropping the right node in Step (3), an example incomplete status looks like this: [`merge-left node, lkey..10, hkey..100`] -> [`'merge-right node, lkey..50, hkey..100`] -> [`'next node, lkey..100, hkey..200`]. The merge-right node has an `'lkey'` lower than merge-left node's `'hkey'`. We fix this by dropping the merge-right node during the recovery. If the crash happens before Step (4), the dummy mapping will point to a deleted Bnode. We then delete this mapping during the recovery.

Regarding data consistency in Bnodes, all the KV pairs that have not completed the second phase (setting the bitmap) will not be exposed. So any partially updated pairs will not be visible.

If an Mnode has its write buffer, the KV pairs in the buffer can be recovered by replaying the WAL log. However, any KV pairs in its corresponding Bnode cannot be used to serve look requests until all of the lost pairs in the write buffer are recovered from the log. These pairs are scattered in the log. It would be too slow to resume the service of requests if one had to wait for the scan of the entire log to be completed.

4.7 Instant Service Resumption with a Structured WAL

The WAL approach has been widely used for preventing data loss in systems such as LevelDB [8, 19], RocksDB [7, 21], and Kangaroo [17]. Existing use of the technique is simplistic: data in new inserts, updates, and deletes that are sent to the in-DRAM data structures are also appended at the tail of a log in the persistent storage. This approach enables efficient I/O by writing data sequentially. However, the side effect is that the data are not well organized and a particular data cannot be efficiently located for quick access. Maintenance of a separate index for the log is way too expensive to be practical. This is especially problematic with the persistent memory as an instant restart is often expected.

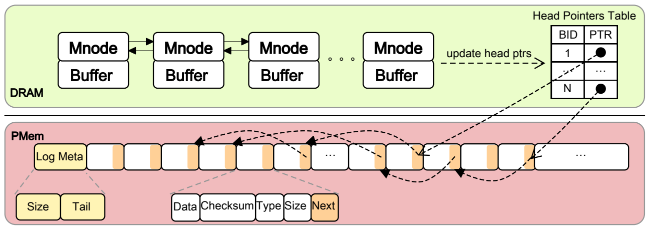


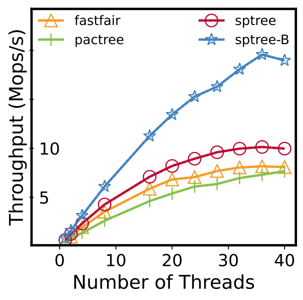
Figure 6: The structured WAL log.

To this end, SPTree proposes a structured WAL log that allows keys to be searched and retrieved from the log without a time-consuming sequential scan. This technique enables (almost) instant resumption access to the SPTree index before a full recovery is completed.

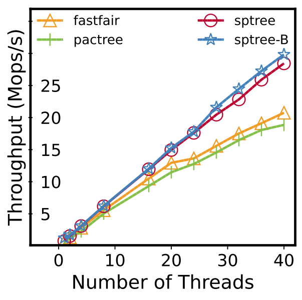
In the design, the conventional WAL log is enhanced by organizing linked lists, each consisting of KV pairs that belong to the same write buffer, in the log. As shown in Figure 6, each record in the log contains a `'next'` pointer, which is the offset of the record in the log about the KV pair that belongs to the same the buffer and has been appended to the log immediately prior to this one. SPTree maintains a table of head pointers in the DRAM, each pointing to the most recently appended record in the log that belongs to a buffer. Each time a buffer's KV pair is appended, the buffer's current head pointer becomes its `'next'` pointer and the `'head'` pointer is updated to the new pair. In this way, without introducing additional writes to the log, the WAL log becomes a structured one containing multiple linked lists. After a crash, SPTree can immediately service a read request by following the corresponding buffer's head pointer and searching on the linked list.

To maintain the structured log for high access efficiency, there are two issues to address. The first one is about placement of head pointers. For efficiency, the table of head pointers must be in the DRAM. In the meantime, it must survive a system crash. To this end, SPTree periodically checkpoints the table to the PMem along with the offset of the WAL's current tail. This offset represents the checkpoint position, indicating that the checkpointed table is up to date until this position. After a crash, the table is reloaded into the DRAM. And SPTree only needs to scan the log from its tail to the checkpoint position to find out all newer header pointers and update the in-DRAM table. Each record contains a 8-byte checksum to determine if it is a valid one. Accordingly, we can identify the log's tail.

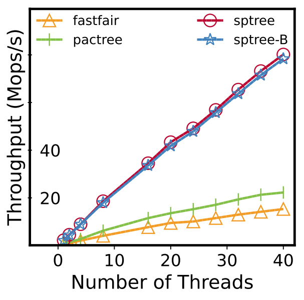
The second issue is about size of the log. Once KV pairs in a buffer have been merged into the in-Pmem Bnode, their corresponding records in the log become obsolete and can be removed from the log as well as the linked lists. Otherwise, they would make the log and search on the log unnecessarily long. Therefore, SPTree needs to identify the true end record in a linked list and flag it. To this end,



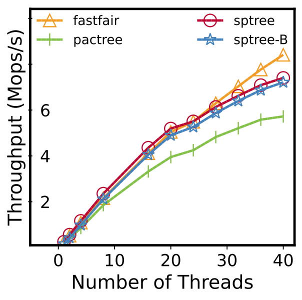
(a) Insert



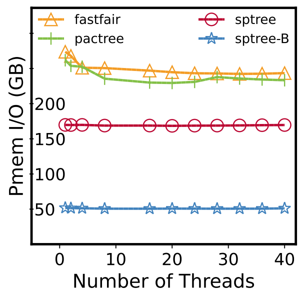
(b) Positive Read



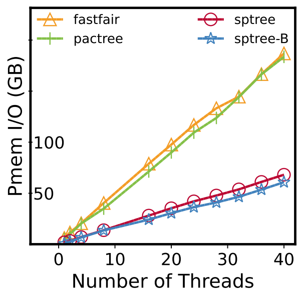
(c) Negative Read



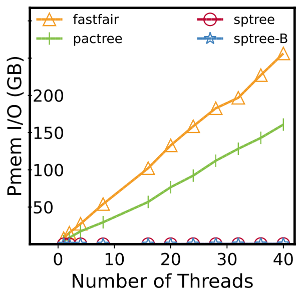
(d) Scan



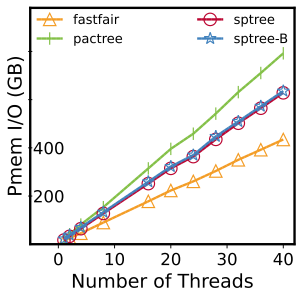
(e) Insert



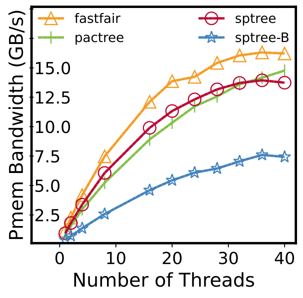
(f) Positive Read



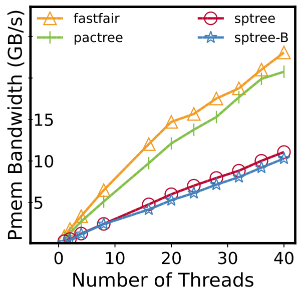
(g) Negative Read



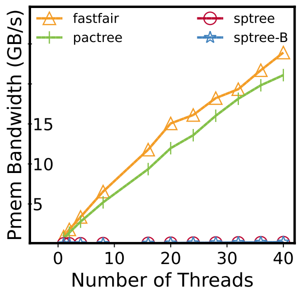
(h) Scan



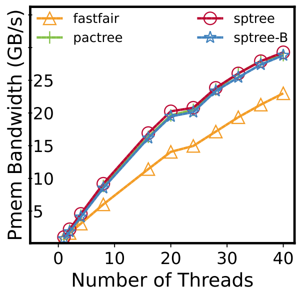
(i) Insert



(j) Positive Read



(k) negative read



(l) Scan

Figure 7: Throughput, PMem I/O volume, and PMem bandwidth with different types of requests.

after persisting a buffer's KV pairs to the BNode, SPTree writes a special record to the WAL flagged (at the `'Type'` field shown in Figure 6) as the end of the buffer's linked list. A key search in the list will stop at the end record. Furthermore, to know the offset in the WAL log beyond which all records in the log can be collected as garbage, SPTree keeps track of the oldest end record, whose up-to-date offset is maintained in the PMem. When a space reclamation via on-log garbage collection is required, all records from this offset to the head of the log can be removed.

Thanks to the linked lists in the WAL, access to the SPTree index can instantly become available. In the meantime, a background thread scans the log to recover the write buffers. When the buffers are fully restored, access to the log for reading KV pairs is no longer required.

5 EVALUATION

In this section, we experimentally evaluate SPTree by comparing it with several state-of-the-art B+-Tree for persistent memory, including FastFair [10] and PACTree [13]. As a sorted index that supports range search, we do not compare SPTree with the hash-based indexes, such as CCEH.

5.1 Experiment Setup

In the experiments, we use 16-byte KV pairs. All the threads in an experiment are pinned to one socket using `numactl`. SPTree's DRAM footprint is about 12% of the total size of the PMem bottom layer when the write buffers are not used. If all of the Mnodes in the SPTree have been assigned with write buffers (denoted *sptree-B*), the DRAM footprint is about 33% of the PMem bottom layer. All the experiments are run on a server with an Intel Xeon Gold 6230 20-core processor, 64GB DRAM and 6 × 128GB Intel Optane DC.

5.2 The Throughput

To evaluate the performance of the trees, we conduct extensive experiments, including insertions of new KV pairs (Insert), reading keys in the indexes (Positive Read), reading keys not in the indexes (Negative Read), and range queries (Scan). Experiment results are shown in Figure 7. In each experiment, different number of threads

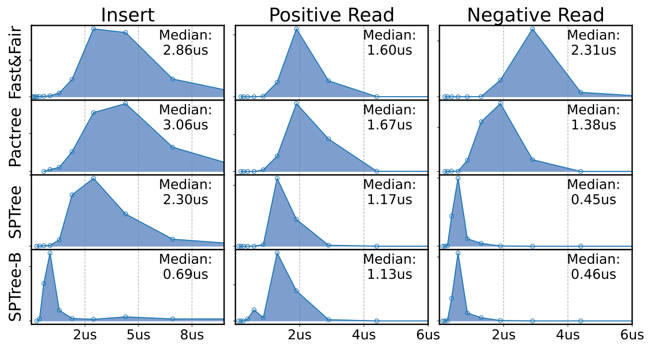


Figure 8: Latency comparison

(from 1 to 40 threads) are used. For *Insert* each thread sends 120 millions/number-of-threads requests. For *Positive Read* and *Negative Read*, and *Scan* each thread sends 10 million requests. Figure 7 reports throughput of the trees (number of requests serviced per second) and the corresponding raw PMem media access amount (I/O volume) . The I/O volume represents all read/write data amount on the Optane PMem's media, including amplified I/O due to existence of its 256B access unit. It is measured with `ipmwatch`, available in the Intel VTune Amplifier tool. It also shows the PMem's bandwidth, which is the raw I/O volume per second.

Insert. As shown in Figure 7a, SPTree consistently outperforms the others for insert performance. This is mainly because in the other two trees the pointer chasing in the internal nodes causes large read amplifications, resulting in reduced effective memory bandwidth. As shown in Figures 7e and 7i, though the bandwidth of FastFair and PACTree during the insertion is equal to or even higher than that of SPTree, their high I/O bandwidth actually results in the lower throughput. When the write buffers are enabled for SPTree (sptree-B), the total I/O volume is reduced by 3X. This is mainly because of the reduced write amplification as well as the read amplification as explained in Section 3.4.

Positive Read. We see up to 25% throughput improvement for SPTree over the others. This advantage mainly comes from the reduced I/O during search in the internal nodes. As shown in Figure 7f, SPTree's I/O volume is only about 1/3 of the others. When write buffers are used, we see slightly performance improvements over the one without buffers. This is because the write buffers also function as read caches for `Positive Read`. Hence, a small portion of the request does not reach the Bnodes in the PMem. That's why I/O volume of sptree-B is smaller, as shown in Figure 7f.

Negative Read. SPTree has up to 4X throughput improvement compared with the other two trees, as shown in Figure 7c. This is because it caches the existing keys in the Mnodes' Bloom filter, which filters out most of the unnecessary PMem accesses during the negative read. As shown in Figures 7g and 7k, there is almost no PMem I/O and we barely read from the PMem. All the saved bandwidth can be used to service other requests.

Scan. Both SPTree and PACTree use an indirection array for sorting keys in the leaf node. This strategy comes with a cost of higher read amplification compared with the physically sorted KV pairs in FastFair. As shown in Figure 7h, SPTree and PACTree have higher I/O volume during the `Scan`. However, thanks to the low overhead of the DRAM top layer, SPTree's scan performance is only 10% lower than FastFair, while PACTree is 30% lower.

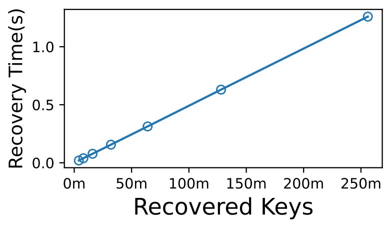
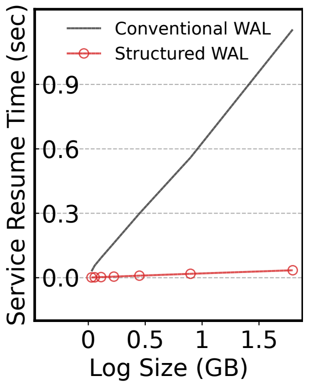
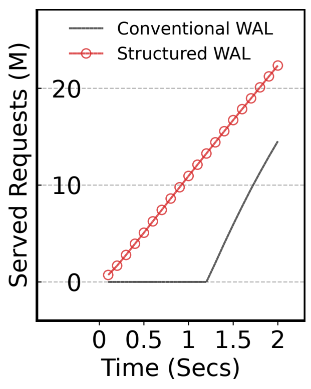


Figure 9: Recovery time to rebuild the DRAM top and middle layers.



(a)



(b)

Figure 10: (a) Service resumption time for varying size of records in the log. (b) Number of lookup requests served within a certain time period right after a restart with a log of 1.8GB.

5.3 The Latency

In this section, we evaluate the read/write latency of the three trees. We use 20 threads to write 120 million KV pairs. Each thread sends 4 million read requests.

As shown in Figure 8, SPTree always has the lowest latency among the trees in all of the workloads (`Insert, Positive Read, and Negative Read`). By caching the internal nodes in the DRAM, the random pointer chasing cost in the PMem is greatly reduced. Meanwhile, using the Bloom filters also helps SPTree to avoid the access to the PMem for non-existing keys. In the experiment, after 120 million KV pairs are inserted, the false positive rate is around 5%. It is worth noting that when write buffers are used, the insert latency for SPTree (SPTree-B) reduces by 3X compared with the one without using write buffer, which is at the cost of DRAM for only around the 30% of the PMem footprint.

5.4 Recovery

If the write buffers for SPTree are not used, SPTree can start servicing the requests without the DRAM top and middle layers as the PMem top layer can be used. In the meantime, it rebuilds the DRAM layers in the background and then puts it into service. As shown in Figure 9, the time to rebuild the DRAM top and middle layers are negligible (1 second for 250 million keys) because it simultaneously rebuilds the DRAM layers based on the PMem top layer.

If the write buffers are used and an unexpected crash takes place, the structured WAL log is employed to quickly resume request service. Figure 10a shows the time it takes to resume its service (for the first lookup request to be served after a crash) with different size of the log (from the oldest end record to the log tail). As shown, using the structural log, SPTree reduces the resumption time to almost 0, much smaller than that using the regular WAL log. Figure 10b shows the number of lookup requests that can be served within a given period of time right after a restart on a log of 1.8GB. As shown, without the structured WAL log no requests can be served until after about 1.4 seconds from the restart. While instant-on resumption is expected for any persistent memory, only SPTree makes it possible with its unique structured WAL design.

6 CONCLUSIONS

In this paper, we propose to use the spot-on DRAM caching and buffering techniques to efficiently address the performance issues at the critical places in an index structure where performance is compromised due to the PMem's performance characteristics. We systematically studied the benefits of the techniques to understand their individual and combined impacts on optimization of in-PMem index structures. Empowered by this understanding, we further introduce SPTree, a persistent ordered tree designed for high-performance systems. Adopting a holistic approach, SPTree leverages the techniques in its design supported by a novel structured WAL log to deliver the instant-on user experience. Experiments show that SPTree minimizes the PMem I/O traffic and achieves 2X to 4X improvement of access performance over the state-of-the-art PMem tree index designs in terms of both throughput and latency.

The source code of the SPTree is available at *https://github.com/hansonzhao007/buflog*.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. This work was partially supported by the US National Science Foundation under grant CCF-2313146.

REFERENCES

[1] [n. d.]. Intel Optane Persistent Memory. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/overview.html.
[2] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. 2018. Bztree: A High-Performance Latch-Free Range Index for Non-Volatile Memory. *Proc. VLDB Endow.* 11, 5 (jan 2018), 553–565. https://doi.org/10.1145/3164135.3164147
[3] Hanyeoreum Bae, Miryeong Kwon, Donghyun Gouk, Sanghyun Han, Sungjoon Koh, Changrim Lee, Dongchul Park, and Myoungsoo Jung. 2021. Empirical Guide to Use of Persistent Memory for Large-Scale In-Memory Graph Analysis. In *39th IEEE International Conference on Computer Design, ICCD 2021, Storrs, CT, USA, October 24-27, 2021.* IEEE, 316–320. https://doi.org/10.1109/ICCD53106.2021.00057
[4] Renhai Chen, Zili Shao, and Tao Li. 2016. Bridging the I/O performance gap for big data workloads: A new NVDIMM-based approach. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).* 1–12. https://doi.org/10.1109/MICRO.2016.7783712
[5] Renhai Chen, Zili Shao, Duo Liu, Zhiyong Feng, and Tao Li. 2019. Towards Efficient NVDIMM-Based Heterogeneous Storage Hierarchy Management for Big Data Workloads. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52).* Association for Computing Machinery, New York, NY, USA, 849–860. https://doi.org/10.1145/3352460.3358266
[6] 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* (Monterey, California) *(INFLOW '15).* Association for Computing Machinery, New York, NY, USA, Article 1, 9 pages. https://doi.org/10.1145/2819001.2819002
[7] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. 2017. Optimizing Space Amplification in RocksDB. In *8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings.* www.cidrdb.org. http://cidrdb.org/cidr2017/papers/p82-dong-cidr17.pdf
[8] S. Ghemawat and J. Dean. 2011. LevelDB. https://github.com/google/leveldb
[9] Jiawei Han, Jian Pei, and Yiwen Yin. 2000. Mining frequent patterns without candidate generation. *ACM sigmod record* 29, 2 (2000), 1–12.
[10] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree. In *16th USENIX Conference on File and Storage Technologies (FAST 18).* USENIX Association, Oakland, CA, 187–200. https://www.usenix.org/conference/fast18/presentation/hwang
[11] intel. [n. d.]. ipmctl. https://github.com/intel/ipmctlïż¿.
[12] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. 2019. Basic performance measurements of the intel optane DC persistent memory module. *arXiv preprint arXiv:1903.05714* (2019).
[13] Wook-Hee Kim, R Madhava Krishnan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. 2021. PACTree: A high performance persistent range index using PAC guidelines. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles.* 424–439.
[14] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, Christian S. Jensen, Christopher M. Jermaine, and Xiaofang Zhou (Eds.). IEEE Computer Society, 38–49. https://doi.org/10.1109/ICDE.2013.6544812
[15] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of Practical Synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware* (San Francisco, California) *(DaMoN '16).* Association for Computing Machinery, New York, NY, USA, Article 3, 8 pages. https://doi.org/10.1145/2933349.2933352
[16] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2020. Dash: Scalable hashing on persistent memory. *arXiv preprint arXiv:2003.07302* (2020).
[17] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger. 2021. Kangaroo: Caching Billions of Tiny Objects on Flash. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, Robbert van Renesse and Nickolai Zeldovich (Eds.). ACM, 243–262. https://doi.org/10.1145/3477132.3483568
[18] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H Noh, and Beomseok Nam. 2019. {Write-Optimized} Dynamic Hashing for Persistent Memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19).* 31–44.
[19] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica* 33, 4 (1996), 351–385. https://doi.org/10.1007/s002360050048
[20] William W. Pugh. 1990. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Commun. ACM* 33, 6 (1990), 668–676. https://doi.org/10.1145/78973.78977
[21] Facebook RocksDB Team. 2021. A Persistent Key-value Store for Fast Storage Environments. http://rocksdb.org
[22] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, and Jishen Zhao. 2021. Characterizing and Modeling Nonvolatile Memory Systems. *IEEE Micro* 41, 3 (2021), 63–70. https://doi.org/10.1109/MM.2021.3065305
[23] Lingfeng Xiang, Xingsheng Zhao, Jia Rao, Song Jiang, and Hong Jiang. 2022. Characterizing the Performance of Intel Optane Persistent Memory: A Close Look at Its on-DIMM Buffering. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22).* Association for Computing Machinery, New York, NY, USA, 488–505. https://doi.org/10.1145/3492321.3519556
[24] Wenhui Zhang, Xingsheng Zhao, Song Jiang, and Hong Jiang. 2021. ChameleonDB: a key-value store for optane persistent memory. In *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*, Antonio Barbalace, Pramod Bhatotia, Lorenzo Alvisi, and Cristian Cadar (Eds.). ACM, 194–209. https://doi.org/10.1145/3447786.3456237
[25] Pengfei Zuo, Yu Hua, and Jie Wu. 2018. Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory. In *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