CCL-BTree: A Crash-Consistent Locality-Aware B$^+$-Tree for Reducing XPBuffer-Induced Write Amplification in Persistent Memory

Zhenxin Li
Zhejiang University
Hangzhou, Zhejiang, China
zhenxin@zju.edu.cn

Shuibing He*
Zhejiang University
Hangzhou, Zhejiang, China
heshuibing@zju.edu.cn

Zheng Dang
Zhejiang University
Hangzhou, Zhejiang, China
dangzheng@zju.edu.cn

Peiyi Hong
Zhejiang University
Hangzhou, Zhejiang, China
hongpeiyi@zju.edu.cn

Xuechen Zhang
Washington State University
Vancouver, Washington, USA
xuechen.zhang@wsu.edu

Rui Wang
Zhejiang University
Hangzhou, Zhejiang, China
rwang21@zju.edu.cn

Fei Wu
Zhejiang University
Hangzhou, Zhejiang, China
wufei@zju.edu.cn

Abstract

In persistent B$^+$-Tree, random updates of small key-value (KV) pairs will cause severe XPBuffer-induced write amplification (XBI-amplification) because CPU cacheline size is smaller than media access granularity in persistent memory (PM). We observe that XBI-amplification directly determines the application performance when the PM bandwidth is exhausted in multi-thread scenarios. However, none of the existing work can efficiently address the XBI-amplification issue while maintaining superior range query performance.

In this paper, we design a novel crash-consistent locality-aware B$^+$-Tree (CCL-BTree). It preserves the key order between adjacent leaf nodes for efficient range query and proposes a *leaf-node centric buffering* strategy that merges writes and then flushes them together to reduce the number of flushes to the PM media. For crash-consistency, all the buffered KVs are recorded in write-ahead logs. CCL-BTree further devises *write-conservative logging* to skip unnecessary log operations, and *locality-aware garbage collection* to avoid random PM writes in reclaiming log data. Our experiments show that CCL-BTree reduces the XBI-amplification by up to 81%, improves the insert throughput by up to 9.35×, and achieves good range query performance compared to state-of-the-art persistent B$^+$-Trees.

CCS Concepts: • **Information systems** → **Data structures**; **Storage class memory**.

Keywords: Persistent Memory, Index Structures, B$^+$-Tree

ACM Reference Format:
Zhenxin Li, Shuibing He, Zheng Dang, Peiyi Hong, Xuechen Zhang, Rui Wang, and Fei Wu. 2024. CCL-BTree: A Crash-Consistent Locality-Aware B$^+$-Tree for Reducing XPBuffer-Induced Write Amplification in Persistent Memory. In *European Conference on Computer Systems (EuroSys '24), April 22–25, 2024, Athens, Greece.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3627703.3629582

1 Introduction

B$^+$-Tree is a widely used index structure in file systems and database systems, because of its better performance than other index structures, e.g., higher range query performance than hash table [34, 52] and radix tree [28, 32], and better data locality than skip list [10, 30]. Persistent B$^+$-Trees [7, 17, 31, 36, 51] that are built on the emerging byte-addressable persistent memory (PM) attracted wide attention in in-memory databases recently, for its low latency, large capacity, and data persistence.

Like DRAM, PM is packaged on the memory bus and handles requests at the 64 B cacheline granularity. For the convenience of illustration, Figure 1 uses the only commercially available PM, Intel Optane DCPMM [19], as an example to illustrate its architecture. Memory accesses to PM

*Shuibing He is the corresponding author.

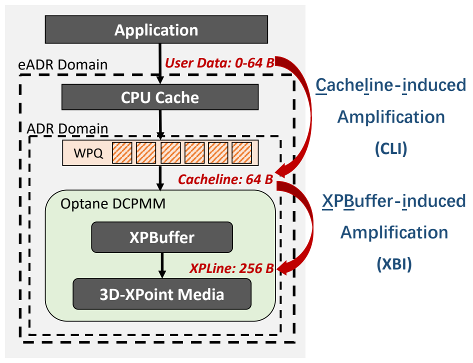

Figure 1. Architecture of PM systems using Optane DCPMM.

are first served by a small on-DIMM write-combining buffer (XPBuffer). They are then transformed to a 256 B data unit (XPLine) before physically accessing the media [44, 47]. As a result, persistent B$^+$-Tree may suffer from severe write amplification in two hardware layers: CPU cache and XPBuffer. Specifically, (1) when randomly writing KVs and their associated metadata, which are smaller than the size of a CPU cacheline, cacheline-induced write amplification (CLI-amplification) may happen. (2) When the cacheline does not fall into the previously accessed XPLine, the cacheline flush may further trigger the XPLine-induced write amplification (XBI-amplification).

To improve the write performance of persistent B$^+$-Trees, most existing work [6, 17, 31, 36, 48, 51] focuses on reducing the number of cacheline flushes to address CLI-amplification. For example, wB$^+$-Tree [6] keeps leaf nodes unsorted to decrease the cost of entry shifting. LB$^+$-Tree [31] packets metadata and data in one CPU cacheline so that they can be updated in one flush rather than two separate flushes. However, we observe that XBI-amplification rather than CLI-amplification directly impacts the write performance of PMs (§2.2). Therefore, these persistent indexes may exhibit suboptimal write performance.

Recent research has introduced a novel approach to address XBI-amplification, called FlatStore [8], which resolves small random writes by organizing them sequentially using a log-structured data layout in PM. This achieves better data locality in XPBuffers and allows for multiple contiguous data writes to be merged into a single XPLine write to PM. However, FlatStore suffers from poor range query performance as the sequentially requested entries are randomly stored in log files. Range query operation is crucial for database systems that require inequality comparisons [13, 35]. Our research shows that FlatStore degrades the performance of range queries by up to 82.1% (§2.3).

None of the existing B$^+$-Trees can efficiently reduce XBI-amplification while maintaining superior range query performance. In this paper, we propose a new persistent B$^+$-Tree variant, named CCL-BTree, which devises three techniques to solve this problem. First, we introduce a *leaf-node centric buffering* strategy (§3.2) that leverages in-DRAM buffers to reduce the total number of XPLine flushes. Specifically, we add a layer of buffer nodes between the last-level inner nodes and their corresponding leaf nodes in the traditional B$^+$-Tree. We store the leaf nodes in PM and keep the inner nodes and buffer nodes in DRAM. Buffer nodes serve two purposes: (1) they will merge write requests and then flush them together to leaf nodes to reduce the number of XPLine flushes to PM media; and (2) they can be used as cache to reduce the number of reads to the leaf nodes. Note that we keep the KVs unsorted within a buffer node or leaf node for lightweight KV insertions, while maintaining the order between adjacent leaf nodes to preserve high range query performance.

Second, given that power failure can result in the loss of KV entries stored in DRAM buffer nodes, we develop a *write-conservative logging* scheme (§3.3) to enforce crash-consistency. The scheme appends a log entry to the write-ahead log (WAL) in PM, and then inserts a new KV into the empty slot in the buffer node. Additionally, it omits logging KVs for writes that trigger buffer nodes to flush to reduce unnecessary log operations and alleviate the XBI-amplification caused by writing WALs.

Third, to avoid random PM writes in reclaiming the log data, we propose a *locality-aware garbage collection* technique (§3.4) that never flushes a KV to a random PM location in garbage collection (GC). To serve this purpose, before writing KVs to the buffer nodes, we record KVs to a primary log, which is used for failure recovery. During the GC, we use a secondary log to store KVs which have not been written to PM and incoming KVs added to the index. Once all buffer nodes are scanned, the secondary log becomes the primary log and the old primary log is marked as deleted and recycled.

By combining these techniques, CCL-BTree alleviates XBI-amplification in all code paths and boosts write performance. In summary, we make the following contributions:

- We conduct an in-depth analysis of write amplification problem for persistent B$^+$-Trees. We find that most of them suffer from high XBI-amplification, which directly impacts write performance of PMs.
- We propose CCL-BTree, which leverages the leaf-node centric buffering, write-conservative logging, and locality-aware GC, to alleviate the XBI-amplification in persistent B$^+$-Trees while maintaining crash-consistency and superior range query performance.
- We implement CCL-BTree and extend it to NUMA scenarios. Our evaluation results show that CCL-BTree

reduces XBI-amplification by up to 81% and 76% under uniform and Zipfian workloads, respectively. It outperforms other indexes by 1.97× to 9.35× for insertions while obtaining similar search performance. The source code of CCL-BTree is is publicly available at https://github.com/ISCS-ZJU/CCL-BTree.

2 Background and Motivation

2.1 Persistent Memory System

PM architecture. Byte-addressable PM sits on the memory bus and includes persistent media and on-chip buffers, exemplified by Intel Optane DCPMM [19] and Samsung's Memory-Semantic SSD [38, 39]. The on-chip buffers are utilized to bridge the performance gap between the fast CPU cache and the relatively slow PM media. As Intel Optane DCPMM is the only commercially available PM device, we use it as an example of PM in this paper and Figure 1 shows its architecture. Memory accesses to DCPMM first arrive at a 16 KB on-DIMM write-combining buffer (XPBuffer), and then are translated to 256 B XPLine before physically accessing the 3D-XPoint media. DCPMM supports two persistent mechanisms: Asynchronous DRAM Refresh (ADR) or extended ADR (eADR) [37]. ADR ensures that, during a power loss, all data in the write pending queues (WPQs) of the integrated memory controller is flushed to PM, whereas data in CPU caches is lost. To achieve crash consistency, programs must explicitly flush CPU cachelines using cacheline flush instructions (e.g., `clflush`, `clflushopt`, or `clwb`) and enforce ordering using memory fence instructions (e.g., `mfence` and `sfence`). The eADR domain includes CPU caches, so that programs no longer need to explicitly flush cachelines for data consistency.

Cacheline-induced amplification. Applications use flush operations to write data from CPU cache to XPBuffer in PM. As the CPU cacheline size is 64 B, an application's small write that is less than 64 B will trigger an amplified cacheline write. We name this phenomenon as the cacheline-induced write amplification (CLI-amplification).

XPBuffer-induced amplification. The data access granularity between the XPBuffer and the 3D-XPoint media is a 256 B unit [47], which is referred to as an XPLine. However, as the cacheline size is 64 B, which is less than the XPLine size, a single cacheline flush to PM may trigger a large-sized XPLine read-modify-write operation to the media. We call this XPBuffer-induced amplification (XBI-amplification).

In this paper, we define the CLI-amplification/XBI-amplification as the ratio between the total amount of data written to XPBuffer/3D-XPoint media and the amount of users' data to write. Accordingly, we use the *ipmctl* tool [18] to collect the hardware counter metrics (i.e., real media read/write bytes and XPBuffer read/write bytes) and calculate the two types of amplification in the following sections.

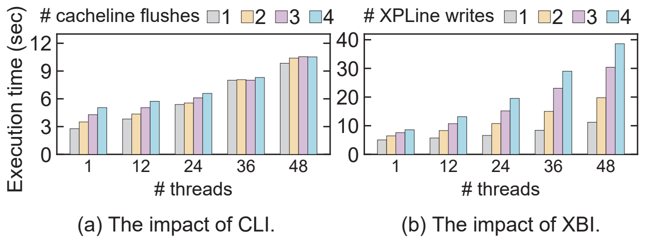

Figure 2. Impact of CLI-amplification and XBI-amplification. (a) Each thread writes and flushes N different cachelines in a randomly selected XPLine 5 million times. (b) Each thread writes and flushes 4 cachelines to N different XPLines 5 million times.

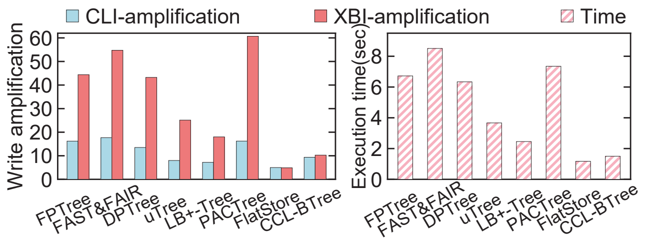

Figure 3. Write amplification and execution time under **uniform distributions**.

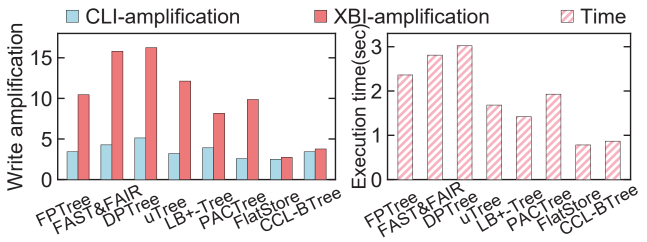

Figure 4. Write amplification and execution time under **Zipfian distributions**.

2.2 Needs of Reducing XBI-Amplification

CLI-amplification and XBI-amplification are two major problems in PM write performance. Since physical media access is often slow and forms a bottleneck in the critical path [44, 47], XBI-amplification critically determines the performance of in-memory applications when the PM bandwidth is exhausted in multi-threaded scenarios.

We design two experiments to verify this. First, we fix the number of XPLine flushes while increasing the number of cacheline flushes per write request. Second, we fix the number of cacheline flushes while increasing the number of XPLine flushes per write request. Figure 2(a) shows that, as the number of threads increases, the execution times for different numbers of cacheline flushes are getting closer. When the number of threads is greater than 36, the execution time is virtually unchanged as the total number of cacheline

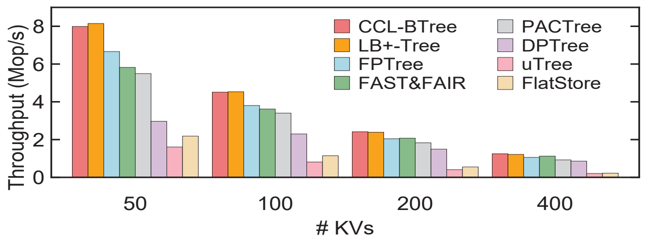

Figure 5. Range query performance with 48 threads.

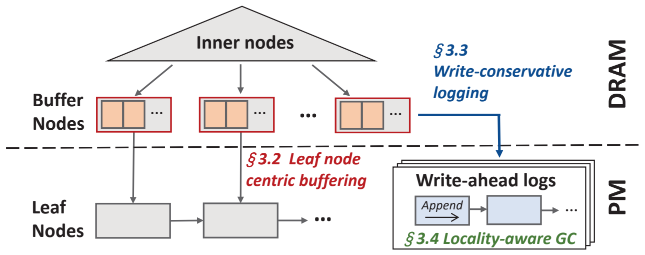

Figure 6. Overview of the CCL-BTree.

flushes is increased by 4×. In contrast, Figure 2(b) shows the execution time of the workload is linearly increased as the number of XPLine flushes is increased. These results show the greater impact of XBI-amplification.

2.3 XBI-Amplification in Persistent B$^+$-Trees

Most of existing persistent B$^+$-Trees are designed to reduce cacheline flushes, thereby alleviating the CLI-amplification. However, they may not help address XBI-amplification as they do not necessarily reduce the number of XPLine flushes.

To verify this, we measure the CLI- and XBI-amplification in existing B$^+$-Trees (FPTree [36], FAST&FAIR [17], DPTree [51], μTree [7], and LB$^+$-Tree [31]), the trie-based PACTree [25], the log-structured FlatStore [8], and our proposed CCL-BTree. We evaluate the indexes in both uniform and Zipfian distributions (coefficient value = 0.9) with 48 threads. We first warm up the indexes using 50 million 16 B KVs and then upsert the remaining 50 million KVs.

Figure 3 and 4 show two observations. (1) Existing persistent B$^+$-Trees are effective at reducing CLI-amplification but inefficient at reducing XBI-amplification. For example, the XBI-amplification is 37.1 and 12.4 on average for the uniform and Zipfian distributions, respectively, while CCL-BTree can reduce it to 10.2 and 3.7; (2) Compared to LB$^+$-Tree and μTree, CCL-BTree has a higher CLI-amplification but lower XBI-amplification under the uniform distribution, because we induce extra cacheline flushes to trade for a locality-friendly access pattern with fewer XPLine flushes.

In contrast to B$^+$-Trees, FlatStore provides both small XBI-amplification and CLI-amplification because the log-structured data layout forms sequential writes in PM. However, it stores KVs in chronological order rather than key order, resulting in significant performance degradation for range queries. To verify this, we execute range query operations on 100 million KVs while varying the query size from 50 to 400 KVs. As shown in Figure 5, FlatStore exhibits up to 5.59× lower throughput than persistent B$^+$-Trees when the scan size is 400. This slow range query performance is unaccepable for many applications.

Summary. We deem that XBI-amplification is a serious and common problem in all existing persistent B$^+$-Tree variants. The straightforward way to reduce XBI-amplification via out-of-place logging breaks the sorted nature of B$^+$-Tree, which degrades the range query performance. We need to directly address XBI-amplification while guaranteeing a good range query performance.

3 Design of CCL-BTree

This section first presents an overview of CCL-BTree and then introduces its three key techniques. CCL-BTree aims to minimize XBI-amplification by reducing random writes in PM, meanwhile maintaining crash consistency and high range query performance.

3.1 Overview

We achieve these goals by introducing a crash-consistent and locality-aware B$^+$-Tree as shown in Figure 6. We first propose a *leaf-node centric buffering* strategy to reduce the XBI-amplification. To ensure the crash consistency of data stored in DRAM buffer nodes, we then propose a *write-conservative logging* scheme that records the data updates in a sequential PM log before writing them to DRAM buffers. Finally, we present a *locality-aware garbage collection* technique to speed up the log reclaiming process without any interference to the frontend workloads.

3.2 Leaf-Node Centric Buffering

A traditional B$^+$-Tree comprises two primary components: (1) inner nodes that store the indexes of items and (2) leaf nodes that store the KV pairs. When a new update arrives, the B$^+$-tree searches the internal nodes for the corresponding leaf node. Then, the update is directly written to the leaf node that stored in PM, leading to the random write and thus causing the XBI-amplification issue.

Limitations of the global buffering approach. To reduce XBI-amplification in PM, an intuitive approach is to utilize a large global buffer pool in DRAM to cache and merge incoming write requests for leaf nodes, as in DPTree [51]. This technique initially stores all KVs in the buffer pool, and then merges them to PM leaf nodes when the pool is full. However, during the merging process, the KVs are randomly inserted into different leaf nodes stored in different PM XPLines, resulting in random writes in PM and the severe XBI-amplification issue. Additionally, this method can lead to decreased insert performance, as foreground threads

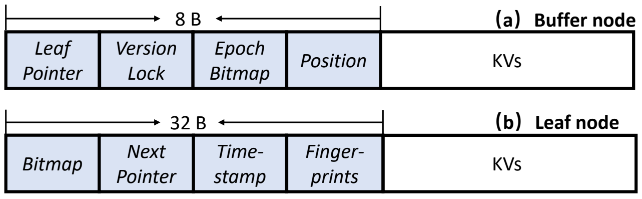

Figure 7. The layout of buffer node and leaf node.

can be stalled by the costly merging operation. Furthermore, search and scan performance can also suffer, as the extra buffer lookup cost introduced by the large global buffer size is non-trivial. We will validate these issues by studying the performance of DPTree in §5.2.

Leaf-node centric buffering approach. To address the above-mentioned issues, we propose a *leaf-node centric buffering* strategy, by introducing a new layer of buffer nodes between the last-level inner nodes and leaf nodes, as shown in Figure 6. Specifically, we store the leaf nodes in PM and keep the inner nodes and buffer nodes in DRAM. It allocates a fine-grained buffer node for each leaf node to buffer multiple writes and flush them to the same XPLine timely to improve the XPLine access locality. Figure 7(a) shows the layout of buffer nodes, which consists of five parts: a pointer to the corresponding leaf node, a version lock for concurrency control, an epoch bitmap for GC, a position counter to store the number of KVs that are not flushed to leaf nodes, and N_{batch} slots to store KV items. To limit the space cost, we store the first four parts in a compressed 8 B header. In addition, we maintain a global epoch, which is initialized as "0". It is then altered between "0" and "1" before the execution of each GC. Each bit of the epoch bitmap indicates whether the corresponding KV is inserted before the current round of GC or during the current GC. The epoch bit of a newly inserted KV is set with the current global epoch.

Insert flow in buffer nodes. Insertions are performed in a log-structured manner in buffer nodes. For a new insertion, we first search the key in inner nodes, just like in a traditional B$^+$-Tree. After traversing down to the target buffer node, we use the position counter to find the first empty slot. If an empty slot is found, we append a log entry to the WAL for crash consistency (§3.3), and then insert the KV to this slot and increase the position counter. If no empty slots are available, we reset the position counter and then flush all cached entries and the new incoming KV in batch into the leaf node together in one XPLine write, thus relieving the XBI-amplification problem. Note that while KVs are not sorted in each buffer node and leaf node to speed up insert performance, we preserve the key order between adjacent leaf nodes to retain high range query performance.

Benefit for query operations. The buffered entries are always up to date and can serve the incoming search requests directly. Motivated by this simple revelation, even when the buffered KVs are flushed to the leaf nodes, they are still reserved in the buffer nodes until overwritten. In this way, the number of slow PM reads to the leaf nodes can be reduced.

Extra DRAM consumption. One concern is that the buffer nodes incur extra DRAM space consumption. Generally, more DRAM space used to buffer will bring more performance improvement. CCL-BTree allows users to control the balance between the two parts by adjusting the value of N_{batch}. In our implementation, we empirically set N_{batch} to 2 by default to achieve a good trade-off (§5.4). We will evaluate the memory consumption in §5.5.

3.3 Write-Conservative Logging

Since data stored in DRAM buffer nodes may be lost on a power failure, we use the ubiquitous WAL technique applied in PM to ensure crash consistency. However, a naive WAL method may incur many additional PM writes, impacting the PM access performance. To address this issue, we propose a *write-conservative logging* strategy to reduce the number of log operations.

Write ahead logging. Each thread has an individual WAL for scalability under high concurrency. Each WAL consists of multiple 4 MB log chunks. CCL-BTree maintains a free log list to manage the recycled log chunks. When a new log chunk is needed, it is first retrieved from the free list. If the free list is empty, a new log chunk is allocated. Each log entry is 24 B storing a KV pair of 16 B and a timestamp of 8 B. We use *_rdtsc*() instruction to obtain the hardware clock and generate timestamps. To avoid the constant skew of the hardware clock across sockets, we use the ORDO primitive [24] to ensure correct ordering of timestamps, as in prior work [27, 46].

Write-conservative logging. As shown in Figure 8(a), a naive logging method will first write the log for each new KV and then flush the cacheline of the log to PM for data consistency. This may yield unnecessary logging overhead. We design a write-conservative logging approach to reduce the number of log operations, and further reduce the PM writes. As shown in Figure 8(b), the idea is to skip the logging operations for KV writes that trigger the flushes when the buffer nodes are full. We name such KV writes as *trigger writes*. As these KVs will be immediately flushed together with the buffered KVs to PM, this write-conservative method can still guarantee their data consistency.

Avoiding logging for *trigger writes* reduces the number of log operations from K to $K * \frac{N_{batch}}{N_{batch}+1}$ where K is the number of insertions. Since N_{batch} is typically set to a small integer (e.g., $N_{batch} = 2$ by default), this strategy can yield considerable performance improvement. Please see §5.3 for a more detailed discussion.

A key challenge of the write-conservative logging is that we may lose track of the latest version of KVs because they may appear in both WALs and leaf nodes. The location of

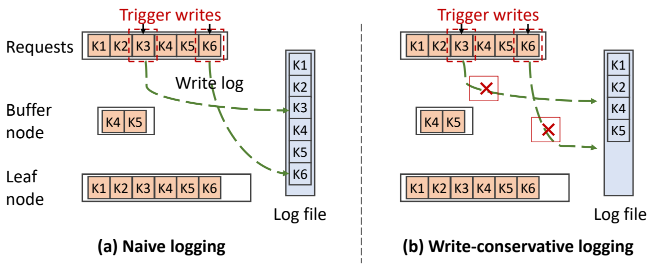

Figure 8. Illustration of logging approaches. (a) The naive logging approach logs all KVs. (b) The write-conservative logging doesn't log KVs (i.e., K3 and K6) that trigger buffer flushing.

the latest version is determined by whether the last insertion is a trigger write. Specifically, in the case of a non-trigger write, the latest version will be stored in the WAL when a system crash occurs. In comparison, for a trigger write, the KV will be directly written to the leaf node without being logged. Therefore, the latest version will be available in the leaf node.

To solve this issue, in addition to tracking the timestamp in the log for each KV insertion, we also introduce a new *timestamp* field in leaf nodes, as shown in Figure 7(b). We update the timestamp field after inserting all buffered KVs and the KV of trigger write into the leaf node. In this way, if there are multiple versions of data of the same key in leaf nodes and WALs during recovery, we can get the latest version of this key by just comparing their timestamps. If the timestamp update of a leaf node is not completed when a failure occurs, the buffered KVs will be overwritten by replaying WALs and the system will resume with the KVs in the WALs after failure recovery. The KV of the corresponding trigger write will be lost.

Failure recovery. With the write-conservative logging strategy described above, we can easily achieve the crash consistent recovery. First, we can reconstruct the CCL-BTree by rebuilding their inner nodes in DRAM while traversing the linked list of leaf nodes in PM. Second, by replaying log entries, we recover the KVs to leaf nodes for those stored in buffer nodes but have not been flushed to PM when failures happened. Finally, we reset timestamps in the leaf nodes. After that, CCL-BTree recovers all KV items and can serve new requests. The empty buffer node will be lazily created for each leaf node when new writes arrive.

3.4 Locality-Aware Garbage Collection

The size of WALs will keep increasing with the write-conservative logging strategy. Therefore, we need a GC mechanism to reclaim the PM space for logging. However, a naive GC strategy would cause additional XBI-amplification problems. We propose a *locality-aware garbage collection* technique to alleviate the XBI-amplification during GC and minimize the interference to the frontend workloads.

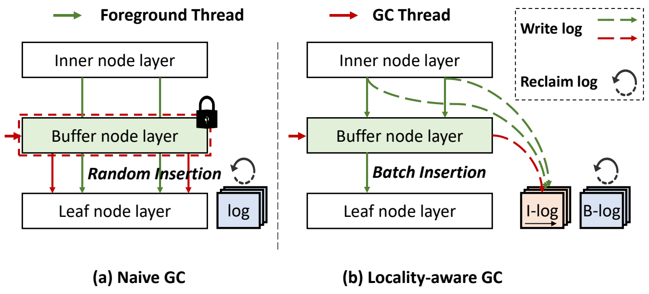

Figure 9. Illustration of GC approaches.

Naive GC. As shown in Figure 9(a), the naive GC process is started by (1) stopping the foreground buffering and logging via a global lock, and meanwhile (2) invoking a background garbage collecting thread to sequentially traverse all buffer nodes (via the pointers in the last-level inner nodes), and flushes the KVs that have not been flushed to leaf nodes. After that, all log chunks in the WALs are moved to the free log list for reclaiming, and we can unlock the buffer node layer to serve the incoming requests.

From the process above, we can see that the naive GC strategy suffers from severe XBI-amplification problem for two reasons: (1) the KVs flushed from the buffer nodes may be located in random leaf nodes in PM; (2) the buffer stops serving incoming requests during the naive GC process, so the requests cannot be batched in the buffer nodes. Our experiments show that the naive GC strategy reduces the throughput of insertion by 37.5% (§5.3).

Locality-aware GC. To alleviate the XBI-amplification during GC, we should keep the buffer function in effect all the time and avoid random flushes by the GC thread. We design a novel locality-aware GC to meet these goals. The main idea is to convert random leaf node access into sequential log writing. The naive GC flushes all buffered KVs from buffer nodes to leaf nodes, resulting in numerous random accesses to different leaf nodes. In contrast, locality-aware GC only copies them to new logs in an append-only manner, thereby eliminating random accesses.

Specifically, we maintain two logs in locality-aware GC, including *B-logs* and *I-logs*. B-logs store all the log entries written before entering GC. And I-logs store all the log entries written during GC. As shown in Figure 9(b), our locality-aware GC process starts by flipping the global epoch number which indicates the pointers of current B-log and I-log. After that, the background GC thread scans all buffer nodes and copies the KVs that have not been written to leaf nodes to I-logs. In the meantime, the new coming KVs can also be buffered to the corresponding buffer nodes and logged to I-logs. The GC thread skips KVs that have been flushed to the leaf nodes and the new coming KVs whose epoch bit is equal to the global epoch. At the end of the GC process, the I-logs contain both a subset of unflushed log entries in the original B-log and new coming KVs inserted during the current GC process. Then, we mark the I-logs as new B-logs to receive the forthcoming log entries, and reclaim the PM space of the original B-logs. When both the foreground threads and the background thread access the same buffer node, we use the version lock in the buffer node for concurrency control.

In our locality-aware GC, we can keep buffer nodes working all the time during the whole GC process, thus maintaining high throughput of insertion. When foreground threads flush leaf nodes through batch insertion, they also accelerate the GC process by reducing the number of KVs to be transferred between B-logs and I-logs. The GC process is triggered when the ratio of the log file size and the size of all the leaf nodes is larger than a memory usage threshold TH_{log}. By default, TH_{log} is empirically set to 20%. We discuss it in §5.4.

3.5 Theoretical Performance Analysis

Quantification of decreased XBI-amplification. Existing researches are ineffective in reducing XBI-amplification because their cacheline flushes are too random to be cached in XPBuffer and each 64 B cacheline flush may trigger a 256 B XPLine flushes to PM media. In contrast, CCL-BTree uses fine-grained buffer node to merge multiple KV writes and flushes them to the same XPLine timely to improve the XPLine access locality. We use an example of K new incoming KV updates to illustrate this. For a traditional B$^+$-Tree, we may need at most K XPLine flushes for these K updates because they may randomly write to different leaf nodes. For CCL-BTree, we can reduce the XPLine flushes for leaf nodes from K to $\frac{K}{N_{batch}+1}$, thus decreasing the XBI-amplification.

Quantification of additional XBI-amplification caused by logging. To ensure data consistency, CCL-BTree introduces extra PM writes by logging, as we need to write the PM log for each KV. However, logs are sequentially appended to the per-thread WAL, and multiple (e.g., 256B / 24B = 10.7) log entries would be merged to one XPLine flush. Thus, for the K KV updates, the logging strategy will incur additional $\frac{24}{256} * K * \frac{N_{batch}}{N_{batch}+1}$ XPLine flushes for WALs, slightly increasing the XBI-amplification.

Overall performance. Based on the above benefit-cost analysis, we find that CCL-BTree can reduce the number of XPLine flushes from K to $\frac{256 + 24 * N_{batch}}{256 * (N_{batch} + 1)} * K$. When N_{batch} is equal to 2 (the default value), it reduces 60.4% XPLine flushes compared to the traditional B$^+$-Tree, thus improving the KV write performance. We will evaluate the overall XBI-amplification and verify this in §5.3.

4 Implementation

Based on the proposed designs above, we implement a prototype system CCL-BTree in C++. In this section, we introduce the key implementation decisions of CCL-BTree to improve its effectiveness and scalability.

4.1 CCL-BTree Structure

Inner nodes. They store the query indexes of inserted keys, and have the same structure as classical B$^+$-Trees. We place them in DRAM and follow the inner nodes implementation of FAST&FAIR [17], by default, and it can be easily replaced by other existing index structure implementations.

Buffer nodes. They store the DRAM buffered KV pairs of the corresponding leaf nodes as stated in §3.2. The detailed implementations refer to Figure 7(a).

Leaf nodes. They store the inserted KV pairs in PM, and their detailed implementations are shown in Figure 7(b). Each leaf node is set as 256 B to make full use of an XPLine access. A leaf node starts with a 32 B header, which stores the metadata of this leaf node, i.e., (1) a 14-bit bitmap to identify invalid keys, (2) a 48-bit pointer to the next leaf node for efficient range scan, (3) an 8 B timestamp for failure recovery, (4) a 14 B fingerprint array for efficient search, and a 2 B padding. The remaining space is used to store KV entries, and it can hold up $(256B - 32B)/16B = 14$ KV entries at most for the default setting of 8 B key and 8 B value. Each fingerprint is a 1 B hash value of the corresponding key in the leaf node. We can compare the fingerprint of a search key with the 14 fingerprints of a leaf node to filter the unmatched keys and quickly locate the slots that contain potential matches, which reduces the number of PM reads [36].

Write-ahead logs. They store the KV log entries that are used for backing up the KV entries buffered in DRAM node buffers. The detailed structure of a log entry is stated in §3.3.

4.2 Insertion Flow

Insertion to leaf nodes. Upon an insertion, the KV is inserted to the buffer node according to the description in §3.2. If the buffer node is full, we flush the cached entries and the new incoming KV in batch in one XPLine flush. The batch insertion is executed in three steps: (1) insert the entries into the empty slots in the data region one by one and record the modified cachelines (the keys are unsorted in the leaf node); (2) persist the modified cachelines using *clwb* and call one *sfence* to ensure the order; (3) update the *fingerprints*, *timestamp*, and *bitmap*, and persist the metadata region using single *clwb* and *sfence*. As any modifications to the leaf node are invisible before the *bitmap* is persisted, the insertion ensures the crash-consistency of the leaf node.

Logless split of leaf nodes. When a leaf node is full during batch insertion, CCL-BTree splits the leaf node without writing logs to reduce XBI-amplification. The first step is to allocate a new leaf node (`newLeaf`) and copy half of the entries from the current leaf node (`oldLeaf`) to the new leaf node. Next, it inserts the KVs whose keys are larger than `splitKey` in the remaining KVs to be inserted into the `newLeaf`. And then it updates the metadata region of `newLeaf` and flushes the entire `newLeaf`. After that, it flushes the modified cacheline in the `oldLeaf`. A single `sfence` is called now to ensure all flushed data has been persistent. Note that we only update the data region of `oldLeaf`, and its metadata region is updated and persisted after this `sfence` instruction. Finally, the remaining KVs (i.e., KVs whose keys are smaller than `splitKey`) are inserted into the `oldLeaf` through normal batch insertion.

Since the *bitmap* and the *nextpointer* fields in the metadata region are compressed to 8 B, we can atomically insert the `newLeaf` into the linked list of leaf nodes and update the status of the splitting leaf node. If a crash happens before that, any modifications to the splitting leaf node are invisible to users and CCL-BTree can restart correctly by scanning the linked list. Besides, we adopt the chunk-based allocation strategy [7] to avoid the potential PM leak for the newly created leaf node.

Update, delete, and merge operations. Update and insertion are implemented as upsert operations. Upsert will insert a new entry if the key does not exist. Otherwise, it performs an update. For a deletion, the tombstone KV (i.e., value is set to zero) is inserted into the buffer node. When the tombstone KV is written to the leaf node, we clear its corresponding bit in the bitmap so that its slot can be used for serving a new KV. In this way, both updates and deletions can be regarded as insertions and benefit from our designs.

If the deletion causes a leaf node to be underutilized (i.e., < 50%) and its left sibling node has enough space, CCL-BTree merges the node to the left sibling node. It first acquires the locks of the two nodes and then moves the KV pairs in the underutilized node to its sibling. After that, the metadata in the sibling node is updated in order. Because the bitmap and the next pointer are 8 B, we can atomically detach the underutilized node from the linked list of leaf nodes and validate the newly inserted KVs in the sibling node. We use the same method as in the split operation to enforce crash consistency and avoid PM leaks in the merge process.

4.3 Query Flow

Point query. For a given key, we first search the key in the inner node layer to find the corresponding target buffer node. Then, we scan the whole buffer node to check if any cached KVs whose key is equal to the target key. For correctness, we should scan the buffer nodes from the left-most slots to avoid getting the stale values. If there is a match, the value of the cached KV in the buffer node is returned directly without accessing PM. Otherwise, we continue to search from the corresponding leaf node. We can check the bitmap and fingerprints compressed in the same cache line to minimize the number of PM reads [36].

Range query. CCL-BTree maintains the key order between adjacent leaf nodes but releases the order within a buffer node or a leaf node. This partially ordered design is widely used to reduce the high PM write overhead caused by item shiftings [6, 25, 31, 36, 51]. Although this incurs overhead for range queries to sort the keys within a leaf node, the overhead is not significant because the sorting process is executed in DRAM and the performance bottleneck is caused by the slow PM read, especially in multi-threaded scenarios. For a range query, we first search the starting key in the inner node layer to find the target buffer node. Then we use the *next* pointers in the last-level inner nodes to traverse the successive buffer nodes and leaf nodes to obtain all KV entries in this range. If entries with the same key exist in both leaf nodes and buffer nodes, we retain the entries stored in the buffer nodes since the buffer nodes always store the latest versions of KVs.

4.4 System Optimizations

Besides the techniques proposed in §3, we further adopt several optimizations to improve the effectiveness and scalability of CCL-BTree.

Optimization #1: NUMA-friendly PM accesses. Remote PM accesses across NUMA may induce extra PM latency and degrade the system performance due to the multi-socket cache coherence [25, 42]. Most of the prior persistent indexes [6, 7, 17, 31, 51] only consider the situation with a single socket and cannot scale well for multiple NUMA nodes. In contrast, CCL-BTree scales well in multi-socket NUMA systems for three reasons. First, CCL-BTree introduces buffer nodes in DRAM to batch the incoming KVs which reduces the number of remote PM accesses. Second, CCL-BTree binds the log file per thread to the local PM to avoid remote logging. Third, the locality-aware GC only scans the buffer nodes in DRAM and copies log entries into the local log file, avoiding remote PM access during GC.

Optimization #2: Concurrency control. In the inner node layer, we adopt the *lock-free search algorithm* used in FAST&FAIR [17], which uses node-level locking for insert operations, for high-performance concurrency control. After traversing down from the root and reaching the target buffer node safely, we switch to the typical version lock protocol to ensure the correctness of the concurrency control. Specifically, each buffer node maintains a version number to detect conflict (i.e., odd number means this node is locked). A writer tries to increment the version number atomically using compare-and-swap (CAS) instruction when the version is even. If the version is odd or the CAS instruction is unsuccessful, the operation retries from the inner node layer. Otherwise, the writer enters the critical section and increases the version number after completing its operations. A reader first checks whether the buffer node is being held by a concurrent writer (i.e., odd version number). If not, it optimistically read data without holding any lock. Upon finishing its operations, it checks the version again and retries if the version number changes. Note that the leaf nodes share the version number of their corresponding buffer nodes, thus no extra locking is needed for leaf nodes. Besides, the version number stored in DRAM can also help avoid excessive PM accesses for conflict detection.

Optimization #3: Variable-size KVs. CCL-BTree adopts an *indirection pointer strategy* for variable-size KVs that are widely used in real-world scenarios [3, 5]. Specifically, CCL-BTree reserves additional PM areas beyond the tree to store the actual keys and values which are larger than 8 B. At the same time, it replaces the keys and values in the tree (as well as in the logs) with 8 B indirection pointers that point to the locations of the actual data. We utilize the most significant bit to indicate whether the 8 B word is an indirection pointer or actual data. This strategy is commonly used in existing persistent indexes [6, 8, 16, 36, 48] and popular database systems [13, 35]. However, updating indirection pointers still suffer from significant XBI-amplification, as an 8 B indirection pointer write can potentially result in a 256 B write on real PM media. Therefore, our techniques remain effective in reducing the XBI-amplification caused by indirection pointers. We will further evaluate the impact of variable-size KVs in §5.4.

5 Evaluation

5.1 Experimental Setup

Platform. We run the experiments on a Linux server with two Intel Xeon Gold 5318Y CPUs. Each CPU has 24 physical/48 logical cores, 64 GB DRAM, and four 128 GB Intel Optane DCPMMs 200 series. The DCPMMs attached to a CPU are mounted with the Ext4-DAX file system and configured in App Direct Mode. We bind every thread to one core using the function $pthread\_setaffinity\_np()$ to avoid the thread switching overhead. All source codes are compiled with g++7.5 with -O3 optimization.

Workloads. We use the micro-benchmarks and the YCSB macro-benchmarks [11] to generate different workloads. For each workload, we first warm up the index with 50 million KVs, so that the data size exceeds the size of the L3-cache and the tested performance reflects the performance of PM [32]. After warming up, we run each test with 50 million operations three times and report the average performance. For the scan operation, we retrieve 100 entries from a random start key. By default, we use 8 B keys and 8 B values.

Target comparisons. We compare CCL-BTree with state-of-the-art persistent structures, including FPTree [36], FAST&FAIR [17], DPTree [51], μTree [7], LB$^+$-Tree [31], PACTree [25], FlatStore [8], and RocksDB deployed on PM [20]. We use their public code except FlatStore, which is not open-source. We reimplement FlatStore as faithfully as possible according to the description in the paper. We do not plot the delete performance of PACTree because we cannot run this function correctly.

To make the comparison fair, we use the same 256 B tree node size for each index except for μTree and DPTree. This configuration yields the best performance for all indexes, except μTree and DPTree, due to the alignment between XPLine size and leaf node. Specifically, each node contains 14 KVs ($14 * 16 = 224$ B) and a 32 B metadata region. The μTree index stores only one KV in each leaf node to avoid expensive structural refinement operations in PM, whereas DPTree requires large leaf nodes containing 256 KVs to amortize persistence overhead. Additionally, we use pre-allocated PM pools from the local socket for all indexes to minimize the allocation overhead since memory allocation can significantly affect index structure performance [12, 25, 31, 32]. We set the default number of GC threads for CCL-BTree to 1.

5.2 Overall Performance

Micro-benchmarks. Figure 10 shows the system performance of each index with uniform key distribution. For insertion, CCL-BTree outperforms other B$^+$-Tree variants by 1.97× to 9.35× with 96 threads. CCL-BTree also has the best scalability. It delivers a continuously increasing throughput until 96 threads while the others achieve nearly saturated throughputs at around 36 threads. There are three reasons for the improvements. First, as we have discussed in §2.2, the XBI-amplification becomes more dominant in insertion performance when the number of threads increases. CCL-BTree can efficiently alleviate it. Second, when the index spans multiple NUMA nodes, the leaf-node centric buffering and NUMA-aware logging can significantly reduce the remote PM accesses. Third, the inherent limitations of existing indexes also degrade their performance. For example, FAST&FAIR and PACTree place the whole tree structure in PM, leading to more slow PM accesses. DPTree adopts the global buffer pool strategy, incurring high XBI-amplification and merge overhead (§3.2). For instance, DPTree has a much higher XBI-amplification (43.2) than CCL-BTree (10.2) with 48 threads. LB$^+$-Tree uses HTM for concurrency control, which might not scale well to multi-threaded and cross-NUMA scenarios as transaction aborts increase. To analyze this further, we show the detailed improvement of each optimization of CCL-BTree in §5.3.

The update and delete operations bring similar performance trends as insertion operations because they also benefit from the optimizations of CCL-BTree. For the search operation, CCL-BTree outperforms other indexes by 1.07× to 2.95× because the cached KVs can be directly returned without accessing the slow PM leaf nodes. For the scan operation, CCL-BTree performs 10% worse than LB$^+$-Tree. This is because CCL-BTree needs to search and merge results from both buffer nodes and leaf nodes to get the latest entries. Except LB$^+$-Tree, it outperforms other indexes by 1.09× to 4.93×. μTree exhibits the worst scan performance (1.53 Mop/s) because it stores only one KV pair in each leaf node, leading to a large number of random PM reads to the linked list.

YCSB macro-benchmarks. We use YCSB benchmark to produce five realistic uniform workloads with different read/write ratios. They are insert-only, insert-intensive (75% insert and 25% read), read-intensive (75% read and 25% insert), read-only, and scan-insert (95% scan and 5% insert).

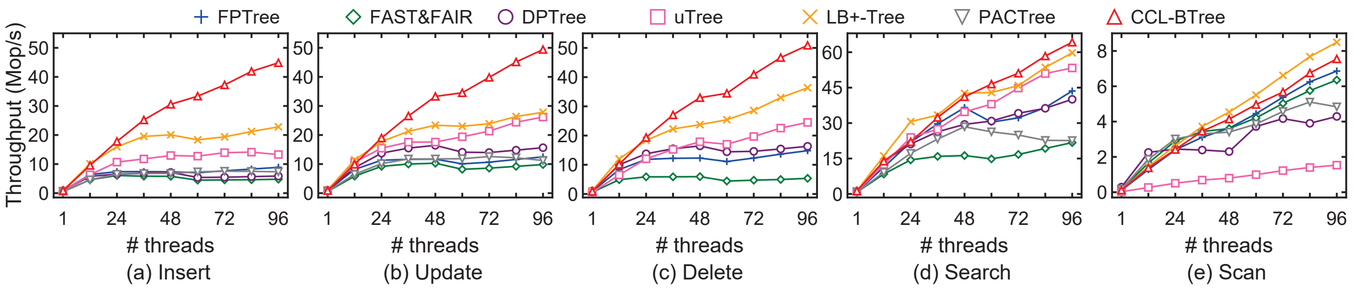

Figure 10. Performance of persistent indexes for the micro-benchmark.

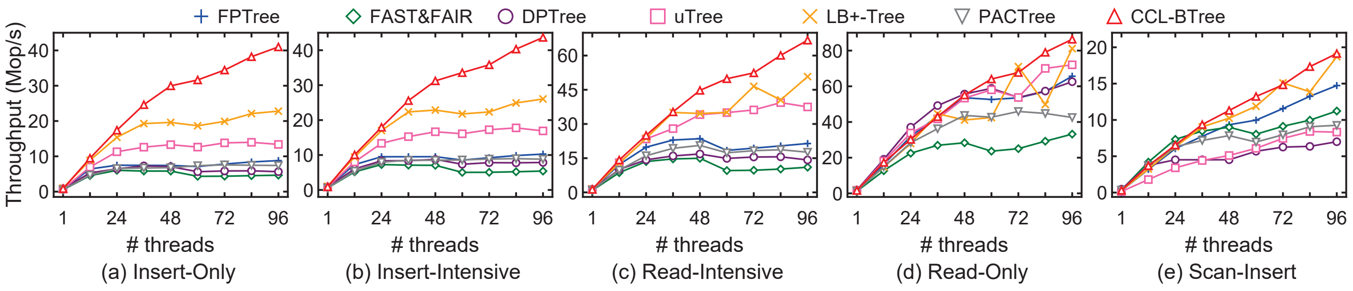

Figure 11. Performance of persistent indexes for the YCSB benchmark.

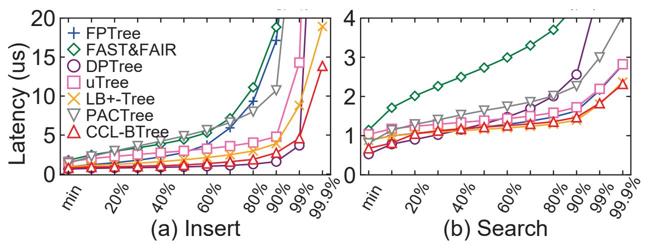

Figure 12. Latency analysis for the micro-benchmarks.

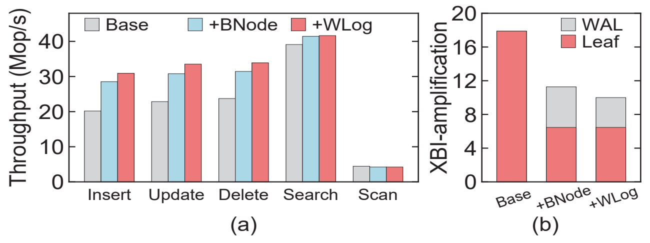

Figure 13. Performance and XBI-amplification analysis of each optimization with 48 threads.

Figure 11 shows the performance trends of YCSB benchmark are similar to the micro-benchmarks. For the insert-only and insert-intensive workloads, CCL-BTree can improve throughputs by at least 1.67× with 96 threads. Besides, CCL-BTree maintains the best performance even within the read-only and scan-insert workloads. These results indicate CCL-BTree also scales well under various real-world access patterns.

Latency analysis. Figure 12 shows the latency distribution of all indexes for the insert and search workloads of the micro-benchmark with 48 threads. As shown in Figure 12(a), CCL-BTree shows a 1.37× to 6.83× lower 99.9th percentile insert latency than other persistent indexes. This is mainly attributed to the less XBI-amplification which alleviates the access blocking time when PM bandwidth is exhausted. Although DPTree shows the lowest insertion latency with the 60 – 99th percentiles, its average latency is longer than CCL-BTree. This is because the foreground requests of DPTree may be stalled by the merging process. When the merging happens, the insertion latency after the 99.9th percentile can reach up to 300 ∼ 400 milliseconds.

For search operations, Figure 12(b) shows that CCL-BTree exhibits much lower latency before the 20th percentile because the target KVs are obtained and returned from the buffer nodes without accessing the slow PM. Starting from the 20th percentile, most indexes show similar search latencies due to the same searching path (i.e., searching from the DRAM indexing layer and then getting the target KVs through one PM access). At the 99th percentile, DPTree shows a higher tail latency (8.2 us) because it needs to search both the global buffer pool and the base tree.

5.3 Improvement of Each Optimization

Buffering and write-conservative logging. Figure 13(a) shows the performance improvement of each optimization in CCL-BTree. *Base* is the implementation of CCL-BTree without any optimizations discussed in §3. *+BNode* denotes

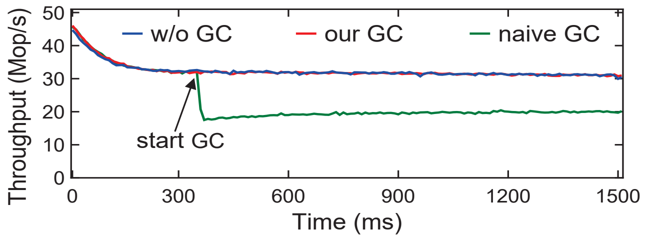

Figure 14. Performance of different GC strategies.

the version with the *leaf-node centric buffering* and the naive WAL logging. *+WLog* denotes the version with the *leaf-node centric buffering* and the *write-conservative logging* enabled.

As shown in Figure 13(a), *+BNode* improves the throughput of *Base* by an average of 35.9% for PM write operations (i.e., insertion, deletion, and update). This is due to the leaf-node centric buffering, which drastically reduces the XBI-amplification as discussed in §3.5. *+BNode* also enhances search performance by 6.5% because some read requests can be served in the buffer nodes without resorting to the slow PM. After enabling the write-conservative logging, *+WLog* further improves the average throughput by 8.3% compared to *+BNode*, thanks to the reduced number of log operations.

For a better demonstration, Figure 13(b) shows the XBI-amplification, which is measured by dividing the actual PM media write size by the user write size, during the execution of *Base*, *+BNode* and *+WLog*. We count the XBI-amplification caused by modifying leaf nodes and WALs separately. The results show that, *+BNode* reduces the XBI-amplification by 63.9% on leaf node modifications compared to *Base*. Although *+BNode* introduces extra PM writes for WALs to ensure data consistency, it still reduces the overall XBI-amplification by 36.9%, resulting in the write performance improvement shown in Figure 13(a). Additionally, by enabling write-conservative logging, *+WLog* further reduces the XBI-amplification on WALs by 26.5% compared to *+BNode*. In summary, with these two techniques, CCL-BTree reduces the total XBI-amplification by 44.1%, which is lower than the ideal value of 60.4% as discussed in §3.5. This is because the ideal value does not take into account the effect of the node splitting.

Locality-aware GC. Figure 14 shows the system throughput with different GC strategies. We first populate the CCL-BTree with 50 million KVs and clean all buffer nodes (i.e., all KVs are flushed into the leaf nodes). After that, we continue inserting KVs and record the throughput periodically. When the GC is triggered, the throughput of naive GC decreases by 37.5% (from 32 Mop/s to 20 Mop/s) because of the random flushes to the leaf nodes. In contrast, we find that our locality-aware GC strategy has a marginal impact on the system performance compared to the case without GC. The overhead of our locality-aware GC is trivial for three reasons: (1) locality-aware GC copies the KVs to a new log instead of flushing them to a random location in PM, thus reducing the number of XPLine flushes and the bandwidth consumption; (2) The foreground threads are executed normally by inserting the incoming KVs to the buffer nodes and I-logs; (3) locality-aware GC skips the KVs that have been flushed to the leaf nodes and inserted during the current GC, which reduces the number of KVs to be copied.

Table 1. Sensitivity of N_{batch}. We use the workloads in micro-benchmarks with 48 threads. *TP* means throughput.

Nbatch		1	2	3	4	5
Insert	TP (Mops/s)	27.4	30.6	31.6	32.9	33.3
	media write (GB)	9.37	7.52	6.63	6.17	5.88
Search	TP (Mops/s)	39.9	41.1	42.9	43.7	44.4
	DRAM hit (M)	4.9	9.4	14.2	17.6	21.7
Usage	DRAM (GB)	0.47	0.59	0.74	0.82	0.93
	PM (GB)	2.71	2.77	2.96	2.83	2.81

Table 2. Sensitivity of TH_{log}. We use the insert workload in micro-benchmarks with 48 threads.

TH_{log}	10%	15%	20%	25%	30%	35%
Throughput (Mop/s)	31.5	31.7	31.6	31.7	31.5	31.9
Peak log size (MB)	825.3	764.6	700.8	857.8	930.1	965.5

5.4 Sensitivity Analysis

Batch size. The number of buffered KVs (i.e., N_{batch}) in each buffer node affects the performance and memory usage of CCL-BTree. Table 1 shows, when N_{batch} increases from 1 to 5, the insert throughput is increased by 21.5% and the search throughput is increased by 11.3%. The improved performance benefits from fewer media write (from 9.37 GB to 5.88 GB) and higher DRAM hits (from 4.9M to 21.7M). However, the DRAM consumption is increased by 97.8%. Therefore, there is a trade-off between system performance and memory usage. Given the decent performance improvement and the modest space overhead, we set N_{batch} to 2 in our design.

GC trigger threshold. Table 2 shows the impact of TH_{log}. We observe that (1) TH_{log} has a marginal influence on the insert throughput because of the effectiveness of our locality-aware technique; and (2) it affects the peak log size. Based on these results, we empirically set TH_{log} to 20% to limit the PM consumption.

Access skewness. Figure 15(a) shows the insert performance with skewed key distributions. We warm up using 50 million KVs and then perform 50% lookup and 50% upsert in 50 million operations with 48 threads. CCL-BTree achieves the best performance in all cases. As the skewness increases, the improvement becomes more prominent because more operations can be completed in the buffer nodes, thus avoiding slow PM accesses. When the coefficient is 0.99, LB$^+$-Tree's performance significantly drops. This is because the highly

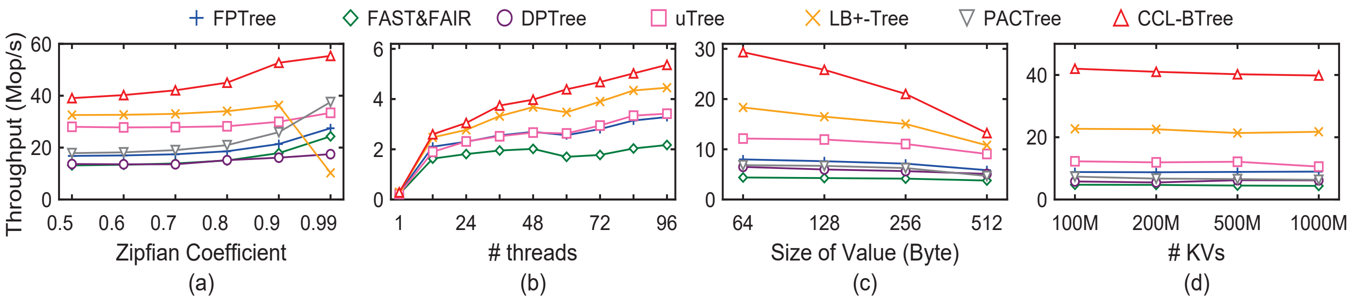

Figure 15. Sensitivity analysis. (a) Skew test with various skewness. (b) Variable-size KV test. The sizes of keys and values are randomly generated from 8 to 128 B. (c) Large value test. The key size is fixed to 8 B. (d) Various dataset sizes.

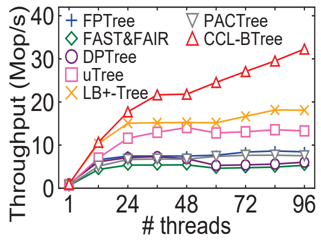

Figure 16. eADR test.

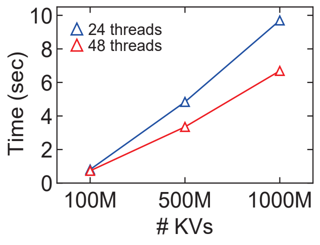

Figure 17. Recovery.

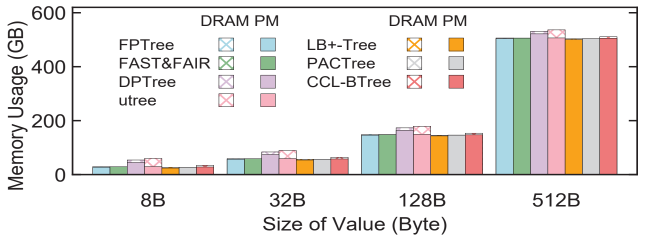

Figure 18. Space consumption.

skewed workload incurs frequent HTM transaction aborts, leading to severe performance degradation.

Variable-size KV. Figure 15(b) shows the insert performance of variable-size KVs with different threads. The sizes of keys and values are randomly generated from 8 to 128 B. We exclude the results of DPTree and PACTree because we are unable to run their code in the test. As expected, all indexes have lower throughput than their counterparts with 8 B fixed-size keys and values due to the pointer chasing and string comparison during the traversal. Nevertheless, CCL-BTree still outperforms other indexes by up to 2.47×.

Large value. Figure 15(c) shows the insert performance of large data with 96 threads. The value size varies from 64 to 512 B and the key size is fixed to 8 B. We adopt indirection pointers to store large values in an out-of-band area. The improvement decreases because the whole XBI-amplification becomes alleviated as the value size increases. However, CCL-BTree still outperforms other indexes by 1.2× to 3.5× for the 512 B value sizes, because the flushes of indirection pointers still benefit from our designs. A large portion of real-world applications generate values smaller than 512 B. For example, Facebook reports the average value size is smaller than 128 B in all its workloads [5]. CCL-BTree benefits more from this kind of workloads.

Dataset size. Figure 15(d) shows the insert throughput of all indexes under 96 threads with various dataset sizes. CCL-BTree's throughput is stable at around 40 Mop/s as the dataset size increases. Even for a very large dataset (i.e., 1000 million operations), CCL-BTree still outperforms other indexes by at least 1.83×.

5.5 Other Tests

The eADR mode. Figure 16 shows the index performance when PM works in the eADR mode. As the eADR platform is still not available, we use the ADR platform to emulate it by removing the flush instructions in the programs, similar to prior work [14, 29, 43, 49]. We only test the insert operation in the micro-benchmarks because search operations show similar trends as in the ADR mode. We can observe that CCL-BTree still outperforms other indexes by 1.78× to 6.07× with 96 threads. Whether the cache line is persistent or not, our designs are capable of improving XPLine locality, which greatly reduces the number of slow accesses to the PM media. Therefore, we expect that CCL-BTree is also efficient on the real eADR platform.

Another interesting observation is that CCL-BTree has a higher throughput if explicit flush operations are maintained (e.g., 44.8 Mop/s with 96 threads in the ADR mode in Figure 10(a) versus 32.3 Mop/s in the eADR mode). This is because the CPU cache size is much smaller than the total dataset size. To make space for new requests, dirty cachelines will be automatically evicted from the CPU cache and flushed to PM. These implicit flushes are oblivious to the XPLine locality and may convert sequential cacheline writes into random XPLine flushes to PM media [14], leading to severe XBI-amplification. This observation further inspires us that, even for the eADR platform, reasonable use of explicit flush operations may improve application performance.

Recovery. Figure 17 shows the recovery performance of CCL-BTree with different KV numbers. The recovery time

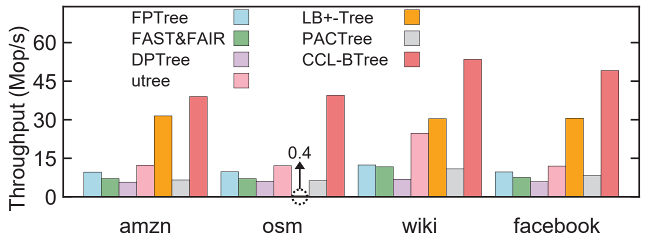

Figure 19. Performance of four realistic datasets.

increases linearly with the total data size and scales well for parallel recovery. With 1000 million KVs (about 16 GB in total size) and 48 threads, the recovery time is 6.2 seconds.

Memory consumption. Figure 18 shows the memory consumption of DRAM and PM with different indexes after inserting one billion KVs. We fix the key size as 8 B and vary the value size from 8 B to 512 B. We can see that the DRAM consumption of CCL-BTree is always much lower than the PM consumption, and occupies 17.5%, 9.3%, 3.9%, 1.1% of the total memory space for different value sizes. These ratios are comparable to other DRAM-PM hybrid indexes. For example, for the case of 8 B value size, CCL-BTree consumes 28.8 GB of PM and 6 GB of DRAM, while FPTree, LB$^+$-Tree, DPTree and μTree consume 26.9 GB, 24.2 GB, 44.9 GB, 29.8 GB of PM space, and 2.5 GB, 2.5 GB, 9.2 GB, 29.8 GB of DRAM space, respectively. FAST&FAIR and PACTree only consume 29.1 GB and 27.1 GB of PM space, respectively, because they are pure PM indexes. Additionally, as the value size increases, the PM consumption of CCL-BTree also increases linearly, while the DRAM consumption remains almost constant by using the *indirect pointer strategy*. As a result, the DRAM consumption ratio decreases linearly. We claim that such DRAM consumption is generally acceptable in most production workloads [5], as the DRAM consumption typically takes up a small fraction of the total data values.

Realistic datasets. To further showcase the performance of CCL-BTree, we evaluate it using four realistic datasets obtained from SOSD [33]. These datasets include: (1) *amzn* containing book popularity data from Amazon, (2) *osm* containing cell IDs from Open Street Map, (3) *wiki* containing timestamps of edits from Wikipedia, and (4) *facebook* containing randomly sampled Facebook user IDs. Both *amzn* and *osm* contain 800 million keys respectively, whereas *wiki* and *facebook* contain 200 million keys. All the datasets use 8 B keys and 8 B values. Figure 19 shows the insert throughputs of all indexes with 96 theads. We observe that CCL-BTree outperforms others by at lease 1.24×, demonstrating its efficiency with various real-world datasets.

Comparison with persistent log structures. We compare CCL-BTree with FlatStore and RocksDB deployed on PM. Table 3 shows the results with 48 threads. While the insert throughput of CCL-BTree is 16% lower than FlatStore, its range query is 3.72× higher. FlatStore has poor range query performance because all KVs are unsorted. RocksDB is always an order of magnitude slower than other indexes and we omit its scan throughput since it takes more than 2 hours. RocksDB's poor insert performance is caused by its expensive compaction operations and PM-oblivious write policy that occurs severe XBI-amplification. The search performance is poor because of the multiple-level searching. The scan operation must seek and sort-merge entries from multiple levels in PM, thus also delivering low throughput.

Table 3. Compare with log-structured indexes.

Throughput (Mops/s)	RocksDB-PM	FlatStore	CCL-BTree
Insert	1.2	36.4	30.5
Search	1.1	38.2	41.1
Scan	x	1.1	4.1

6 Generality Discussion

Applicability to other PM devices. Our CCL-BTree design is based on Intel Optane DCPMMs [19]. Despite Intel having ceased its Optane memory business in August 2022, they are scheduled to unveil a new family of PM products for the 4th and 5th Generations Xeon Scalable processors (Sapphire Rapids and Emerald Rapids) [40]. Our proposed techniques are still useful for these devices as they continue to use 3D-XPoint as internal media. Additionally, CCL-BTree can also be applied to future PM devices facing access granularity mismatches between cacheline and device's internal media. Several compelling examples are the CXL-based PM products such as Samsung's Memory-Semantic SSD [38] and KIOXIA's XL-FLASH [26], and byte-addressable SSD [1]. All these promising alternatives to Optane DCPMMs have an internal buffer, whose block-level media access granularity (e.g., 4 KB flash page) is significantly larger than the cacheline size. As a result, these devices also suffer from severe XBI-like write amplification. In the future, other types of PMs based on different media (e.g., NRAM [2], PCM [45], MRAM [41], etc.) may appear, which could also have a limited buffer amount with high media access granularity to match the speeds of fast cache and slow memory media, making our techniques still applicable to them.

Applicability to other indexes. Besides B$^+$-Trees, the ideas of CCL-BTree can also apply to other types of index structures, such as radix trees and hash tables. For example, in the persistent hash tables (e.g., CCEH [34], CLevel [9]), we can introduce a buffer node for one or multiple buckets to batch the updates to them, and use the write-conservative logging and locality-aware GC to ensure crash consistency with reduced write amplification.

7 Related Work

Tree-based indexes. A large number of researches [4, 6, 7, 15, 17, 21, 25, 31, 36, 48, 51] are proposed to build efficient

B^+-Tree or its variants for PM and storage. Most of them [6, 17, 31, 36, 48, 51] focus on reducing the number of cacheline flushes to improve the write performance in PM. μTree [7] aims to reduce the tail latency by moving the expensive structural refinement operations out of PM. PACTree [25] is the first persistent index that mitigates NUMA effects by using separate pools for data in each NUMA node. B^{ε}-tree [4] also uses inner nodes to buffer writes but it is a write-optimized data structure for disk storage. None of them has considered the mismatch between the cacheline size and the media access granularity of PM, leading to severe XBI-amplification. In contrast, CCL-BTree considers both the random access pattern of B^+-Tree and the media access granularity of PM, alleviating the XBI-amplification.

Log-structured indexes. Many efforts [22, 23, 50] have redesigned the LSM-Tree based indexes for PM. However, range queries in LSM-Trees must seek and sort-merge entries from multiple levels, which leads to lower scan performance compared to B^+-Tree. FlatStore [8] incorporates a volatile index for fast indexing and a persistent log structure for storage. It enables sequential writes in PM to make full use of the PM bandwidth. But it still suffers from poor scan performance because the KVs are not sorted in logs. Different from these work, CCL-BTree achieves efficient range queries by maintaining the order between adjacent leaf nodes.

8 Conclusions

In this paper, we design and implement a crash-consistent locality-aware B^+-Tree named CCL-BTree, which tackles the XBI-amplification issue in PM while providing high range query performance. CCL-BTree consists of three techniques including *leaf-node centric buffering*, *write-conservative logging*, and *locality-aware garbage collection* to alleviate the XBI-amplification in all the code paths. Our results show that CCL-BTree reduces the XBI-amplification by up to 81% and 76% under uniform and Zipfian workloads respectively, and improves the insert throughput by up to 9.35× while maintaining superior range query performance compared to state-of-the-art indexes.

Acknowledgments

We sincerely thank our shepherd Tianzheng Wang and the anonymous reviewers for their constructive suggestions. This work was supported in part by the National Key Research and Development Program of China under Grant 2021ZD0110700, the National Science Foundation of China under Grant 62172361, the Program of Zhejiang Province Science and Technology under Grant 2022C01044, and the US National Science Foundation under Grant CNS 1906541.

References

[1] Ahmed Abulila, Vikram Sharma Mailthody, Zaid Qureshi, Jian Huang, Nam Sung Kim, Jinjun Xiong, and Wen-mei Hwu. 2019. Flatflash: Exploiting the Byte-Accessibility of SSDs within A Unified Memory-Storage Hierarchy. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 971–985.

[2] Gilles Amblard. 2011. Development and Characterization of Carbon Nanotube Processes for NRAM Technology. In *Alternative Lithographic Technologies III*, Vol. 7970. International Society for Optics and Photonics, 797017.

[3] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. 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*. 53–64.

[4] Michael A Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Bradley C Kuszmaul, Donald E Porter, Jun Yuan, and Yang Zhan. 2015. An Introduction to B-trees and Write-Optimization. *login; magazine* 40, 5 (2015).

[5] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. 2020. Characterizing, Modeling, and Benchmarking Rocksdb Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST)*. 209–223.

[6] Shimin Chen and Qin Jin. 2015. Persistent B+-Trees in Non-Volatile Main Memory. *Proceedings of the VLDB Endowment (VLDB)* 8, 7 (2015), 786–797.

[7] Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, and Jiwu Shu. 2020. UTree: A Persistent B+-Tree with Low Tail Latency. *Proceedings of the VLDB Endowment (VLDB)* 13, 12 (2020), 2634–2648.

[8] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. 2020. FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 1077–1091.

[9] Zhangyu Chen, Yu Huang, Bo Ding, and Pengfei Zuo. 2020. Lock-Free Concurrent Level Hashing for Persistent Memory. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*. 799–812.

[10] Sakib Chowdhury and Wojciech Golab. 2021. A Scalable Recoverable Skip List for Persistent Memory. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 426–428.

[11] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SOCC)*. 143–154.

[12] Zheng Dang, Shuibing He, Peiyi Hong, Zhenxin Li, Xuechen Zhang, Xian-He Sun, and Gang Chen. 2022. NVAlloc: Rethinking Heap Metadata Management in Persistent Memory Allocators. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 115–127.

[13] PostgreSQL Global Development Group. 1996. PostgreSQL. https://www.postgresql.org/. Last accessed on Sep-2023.

[14] Shashank Gugnani, Arjun Kashyap, and Xiaoyi Lu. 2020. Understanding the Idiosyncrasies of Real Persistent Memory. *Proceedings of the VLDB Endowment (VLDB)* 14, 4 (2020), 626–639.

[15] Chenchen Huang, Huiqi Hu, and Aoying Zhou. 2021. BPTree: An Optimized Index with Batch Persistence on Optane DC PM. In *Database Systems for Advanced Applications: 26th International Conference, DASFAA 2021, Taipei, Taiwan, April 11–14, 2021, Proceedings, Part III 26*. Springer, 478–486.

[16] Yihe Huang, Matej Pavlovic, Virendra Marathe, Margo Seltzer, Tim Harris, and Steve Byan. 2018. Closing the Performance Gap between Volatile and Persistent Key-Value Stores Using Cross-Referencing Logs. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 967–979.

[17] 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)*. 187–200.

[18] Intel. 2018. A Utility for Configuring and Managing Intel Optane DC Persistent Memory Modules. https://github.com/intel/ipmctl. Last accessed on Sep-2023.

[19] Intel. 2019. Intel® Optane^{TM} DC Persistent Memory. https://www.intel.com/content/www/us/en/products/memory-storage/optane-dc-persistent-memory.html. Last accessed on Sep-2023.

[20] Intel. 2019. PMEM-RocksDB: A Persistent Key-Value Store for Flash and RAM Storage. https://github.com/pmem/pmem-rocksdb. Last accessed on Sep-2023.

[21] Yizheng Jiao, Simon Bertron, Sagar Patel, Luke Zeller, Rory Bennett, Nirjhar Mukherjee, Michael A Bender, Michael Condict, Alex Conway, Martín Farach-Colton, et al. 2022. BetrFS: A Compleat File System for Commodity SSDs. In *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys)*. 610–627.

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

[23] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *2018 USENIX Annual Technical Conference (ATC)*. 993–1005.

[24] Sanidhya Kashyap, Changwoo Min, Kangnyeon Kim, and Taesoo Kim. 2018. A Scalable Ordering Primitive for Multicore Machines. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys)*. 1–15.

[25] 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 (SOSP)*. 424–439.

[26] KIOXIA. 2022. Kioxia Launches Second Generation of High-Performance, Cost-Effective XL-FLASH Storage Class Memory Solution. https://www.kioxia.com/en-jp/business/news/2022/20220802-1.html. Last accessed on Sep-2023.

[27] R Madhava Krishnan, Wook-Hee Kim, Xinwei Fu, Sumit Kumar Monga, Hee Won Lee, Minsung Jang, Ajit Mathew, and Changwoo Min. 2021. TIPS: Making Volatile Index Structures Persistent with DRAM-NVMM Tiering. In *2021 USENIX Annual Technical Conference (ATC)*. 773–787.

[28] Se Kwon Lee, K Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H Noh. 2017. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*. 257–270.

[29] Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. 2019. Evaluating Persistent Memory Range Indexes. *Proceedings of the VLDB Endowment (VLDB)* 13, 4 (2019), 574–587.

[30] Zhenxin Li, Bing Jiao, Shuibing He, and Weikuan Yu. 2022. PHAST: Hierarchical Concurrent Log-Free Skip List for Persistent Memory. *IEEE Transactions on Parallel and Distributed Systems* (2022).

[31] Jihang Liu, Shimin Chen, and Lujun Wang. 2020. LB+Trees: Optimizing Persistent Index Performance on 3DXPoint Memory. *Proc. VLDB Endow.* 13, 7 (2020), 1078–1090.

[32] Shaonan Ma, Kang Chen, Shimin Chen, Mengxing Liu, Jianglang Zhu, Hongbo Kang, and Yongwei Wu. 2021. ROART: Range-Query Optimized Persistent ART. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST)*. 1–16.

[33] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. 2020. Benchmarking Learned Indexes. *Proceedings of the VLDB Endowment (VLDB)* 14, 1 (2020), 1–13.

[34] 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)*. 31–44.

[35] Oracle. 1995. MySQL. https://www.mysql.com/. Last accessed on Sep-2023.

[36] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. 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)*. 371–386.

[37] Andy Rudoff. 2020. Persistent Memory Programming without All That Cache Flushing. *SDC* (2020).

[38] Samsung. 2022. Memory-Semantic SSD^{TM}: Industry 1st CXL-Based Storage Optimized for AI/ML. https://samsungmsl.com/ms-ssd/. Last accessed on Sep-2023.

[39] Samsung. 2022. Why We Built the Industry First CXL-Based NAND Flash SSD? https://www.youtube.com/watch?v=Ol0Ct_WMZuE. Last accessed on Sep-2023.

[40] Anton Shilov. 2023. Optane's Last Gasp: Intel's Final Persistent Memory Roadmap Leaks. https://www.tomshardware.com/news/intel-optane-last-gasp. Last accessed on Sep-2023.

[41] Said Tehrani, JM Slaughter, E Chen, M Durlam, J Shi, and M DeHerren. 1999. Progress and Outlook for MRAM Technology. *IEEE Transactions on Magnetics* 35, 5 (1999), 2814–2819.

[42] Qing Wang, Youyou Lu, Junru Li, and Jiwu Shu. 2021. Nap: A Black-Box Approach to NUMA-Aware Persistent Memory Indexes. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 93–111.

[43] Rui Wang, Shuibing He, Weixu Zong, Yongkun Li, and Yinlong Xu. 2022. XPGraph: XPline-Friendly Persistent Memory Graph Stores for Large-Scale Evolving Graphs. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1308–1325.

[44] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, and Jishen Zhao. 2020. Characterizing and Modeling Non-Volatile Memory Systems. In *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 496–508.

[45] H-S Philip Wong, Simone Raoux, SangBum Kim, Jiale Liang, John P Reifenberg, Bipin Rajendran, Mehdi Asheghi, and Kenneth E Goodson. 2010. Phase Change Memory. *Proc. IEEE* 98, 12 (2010), 2201–2227.

[46] Kai Wu, Jie Ren, Ivy Peng, and Dong Li. 2021. ArchTM: Architecture-Aware, High Performance Transaction for Persistent Memory. In *19th USENIX Conference on File and Storage Technologies (FAST)*. 141–153.

[47] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST)*. 169–182.

[48] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: Reducing Consistency Cost for NVM-Based Single Level Systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*. 167–181.

[49] Bowen Zhang, Shengan Zheng, Zhenlin Qi, and Linpeng Huang. 2022. NBTree: A Lock-Free PM-Friendly Persistent B+-Tree for eADR-Enabled PM Systems. *Proceedings of the VLDB Endowment (VLDB)* 15, 6 (2022), 1187–1200.

[50] Wenhui Zhang, Xingsheng Zhao, Song Jiang, and Hong Jiang. 2021. ChameleonDB: A Key-Value Store for Optane Persistent Memory. In *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys)*. 194–209.

[51] Xinjing Zhou, Lidan Shou, Ke Chen, Wei Hu, and Gang Chen. 2019. DPTree: Differential Indexing for Persistent Memory. *Proceedings of the VLDB Endowment (VLDB)* 13, 4 (2019), 421–434.

[52] 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)*. 461–476.