








Revisiting PM-Based B⁺-Tree With Persistent CPU Cache

Bowen Zhang , Shengan Zheng , Liangxu Nie , Zhenlin Qi , Hongyi Chen ,
Linpeng Huang , *Senior Member, IEEE*, and Hong Mei , *Fellow, IEEE*

Abstract—Persistent memory (PM) promises near-DRAM performance as well as data persistence. Recently, a new feature called eADR is available for PM-equipped platforms to guarantee the persistence of CPU cache. The emergence of eADR presents unique opportunities to build lock-free data structures and unleash the full potential of PM. In this paper, we propose NBTree, a lock-free PM-friendly B⁺-Tree, to deliver high scalability and low PM overhead. To our knowledge, NBTree is the first persistent index designed for PM systems with persistent CPU cache. To achieve lock-free, NBTree uses atomic primitives to serialize index operations. Moreover, NBTree proposes five novel techniques to enable lock-free accesses during structural modification operations (SMO), including *three-phase SMO*, *sync-on-write*, *sync-on-read*, *cooperative SMO*, and *shift-aware search*. To reduce PM access overhead, NBTree employs a decoupled leaf node design to absorb the metadata accesses in DRAM. Moreover, NBTree devises a cache-crafty persistent allocator and adopts *log-structured insert* and *in-place update/delete* to enhance the access locality of write operations, absorbing a substantial amount of PM writes in persistent CPU cache. Our evaluation shows that NBTree achieves up to 11× higher throughput and 43× lower 99% tail latency than state-of-the-art persistent B⁺-Trees under YCSB workloads.

Index Terms—B⁺-Tree, EADR, lock-free, persistent CPU cache, persistent memory.

I. INTRODUCTION

BYTE-ADDRESSABLE persistent memory (PM), such as Intel Optane DC persistent memory module (DCPMM) [1], offers DRAM-comparable performance as well as disk-like durability. In general, PM-equipped platforms support the asynchronous DRAM refresh (ADR) feature [2], which ensures that the content of the PM DIMMs, as well as the writes that have reached the memory controller's write pending

queues (WPQ), survives power failures. However, writes within CPU cache remain volatile. Thus, explicit cacheline flush instructions and memory barriers are required to guarantee the data persistence.

Recently, a new platform feature called extended ADR (eADR) is available with the arrival of the 3rd generation Intel Xeon Scalable Processors and the 2nd generation Intel Optane DCPMM [3]. Compared with ADR, eADR further guarantees that data within CPU cache will be flushed back to PM after a crash through the reserved energy. It ensures the persistence of globally visible data in CPU cache and eliminates the need to issue costly synchronous flushes. The emergence of eADR not only facilitates the design of lock-free data structures but reduces PM write overhead.

Building efficient index structures in PM is promising to offer both high performance and data durability for in-memory databases. Most existing persistent indexes [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15] are solely designed for ADR-based platforms. On eADR-enabled platforms, an intuitive transformation approach is to simply remove all cacheline flush instructions [16]. However, this naive approach cannot fully exploit the potential of persistent CPU cache. Those indexes still suffer from two major drawbacks that keep them from achieving high performance in eADR-enabled platforms.

First, existing persistent indexes suffer from inefficient concurrency control. Locks are widely used in persistent indexes because none of the existing primitives can atomically modify and persist data on ADR-based platforms. Atomic CPU hardware primitives, such as Compare-And-Swap (CAS), can atomically modify the data but do not guarantee its persistence because CPU cache is volatile. Therefore, without locking, it's possible that a *store* hasn't been persisted before a dependent read from another thread, leading to a *dirty read* anomaly. Fortunately, eADR closes the gap between the visibility and persistence of the data in CPU cache, making sure that threads always read persistent data. Thus, eADR provides us with opportunities to develop efficient lock-free data structures.

Second, existing persistent indexes still impose high overheads on PM accesses. Prior researchers strive to lower PM overhead by reducing the number of cacheline flush instructions, because data flushing is the primary bottleneck of ADR-based PM systems [7]. With eADR, explicit data flushes to PM are no longer necessary. However, excessive PM accesses in index operations and memory management continue to hinder the effectiveness of persistent indexes since PM has higher read

Manuscript received 18 August 2023; revised 14 January 2024; accepted 27 February 2024. Date of publication 5 March 2024; date of current version 19 March 2024. This work was supported in part by the National Key Research and Development Program of China under Grant 2022YFB4500303, in part by the National Natural Science Foundation of China (NSFC) under Grant 62227809, in part by the Fundamental Research Funds for the Central Universities, Shanghai Municipal Science and Technology Major Project under Grant 2021SHZDZX0102, and in part by the Natural Science Foundation of Shanghai under Grant 22ZR1435400. Recommended for acceptance by D. Li. (Corresponding author: Shengan Zheng.)

Bowen Zhang, Liangxu Nie, Zhenlin Qi, Hongyi Chen, Linpeng Huang, and Hong Mei are with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai 200240, China.

Shengan Zheng is with the MoE Key Lab of Artificial Intelligence, AI Institute, Shanghai Jiao Tong University, Shanghai 200240, China (e-mail: shengan@sjtu.edu.cn).

Digital Object Identifier 10.1109/TPDS.2024.3372621

latency and lower bandwidth than DRAM [17], [18], [19]. Especially for the write operations, although data flushing to PM is off the critical path, dirty cachelines will eventually be written back to PM due to the limited CPU cache capacity, which consumes scarce PM write bandwidth. Therefore, in order to fully exploit the potential of PM systems with persistent CPU cache, it's necessary to further enhance the cache utilization and minimize PM accesses in persistent indexes.

In this paper, we present NBTree, a lock-free PM-friendly B⁺-Tree to deliver high scalability and low PM overhead. To our knowledge, NBTree is the first PM index based on eADR-enabled platforms.

To achieve high scalability, NBTree proposes a fully lock-free concurrency control protocol. For leaf node operations, NBTree adopts *log-structured insert* and *in-place update/delete*, combining with CAS primitives, to support lock-free accesses. When the inserted leaf is full, NBTree replaces the old leaf with new leaves to maintain the balance of nodes via structural modification operations (SMO). NBTree proposes three novel techniques (*three-phase SMO*, *sync-on-write*, and *sync-on-read*) to deal with the potential anomalies during the lock-free accesses to the leaf in SMO: (1) *Lost update* caused by concurrent updates and deletions to the leaf. NBTree addresses this anomaly by utilizing *three-phase SMO* and *sync-on-write*. When an update or deletion operates on the leaf during SMO, it first in-place modifies the old leaf. Then, the modification is either passively migrated to the new leaf by *three-phase SMO* or actively synchronized to the new leaf using *sync-on-write*. (2) *Inconsistent read* caused by concurrent search operations. The lock-free search on the SMO leaf might read uncommitted dirty data or stale data. NBTree uses the *sync-on-read* technique to detect and resolve those anomalies. To further reduce tail latency, we propose *cooperative SMO* to make concurrent insertions to the same SMO leaf work cooperatively. For inner node operations, NBTree applies hardware transactional memory (HTM) [20] to achieve atomic writes. Meanwhile, NBTree designs a *shift-aware search* algorithm to ensure the lock-free inner node search reaches the correct leaf.

To reduce PM overhead, NBTree minimizes PM line accesses and improves the locality of PM writes. First, NBTree employs a decoupled leaf node architecture to reduce PM line accesses in index operations. For leaf nodes in NBTree, the metadata and key-value pairs are decoupled into two layers. The metadata layer is stored in DRAM along with inner nodes. PM only contains the key-value layer so that the number of PM reads and writes is minimized. Second, NBTree proposes a Cache-crafty persistent allocator (Calloc) to enhance the locality of metadata modifications for persistent memory management. Specifically, Calloc employs a global allocation bitmap, reclaimed ring buffers, and recyclable logs, which fully utilize persistent CPU cache to absorb the majority of PM writes occurred during allocation, deallocation, and logging, thereby conserving scarce PM write bandwidth. Meanwhile, NBTree adopts *log-structured insert* and *in-place update* to improve the locality of write operations in NBTree, reducing PM writes in the skewed workloads.

In summary, the contributions of this paper include:

- We provide an in-depth analysis of the benefits of persistent CPU cache. Then, we propose NBTree, the first persistent index designed for PM systems with persistent CPU cache as far as we know.
- We propose lock-free concurrency control for NBTree to achieve high scalability. Our proposed techniques, such as *three-phase SMO*, *sync-on-write*, *sync-on-read*, *cooperative SMO*, and *shift-aware search*, ensure strong consistency for lock-free operations.
- We propose a decoupled leaf node structure for NBTree, which reduces the number of PM line reads and writes in each operation and improves cache utilization.
- We propose a cache-crafty persistent memory allocator for NBTree, which fully exploits persistent CPU cache to reduce PM writes produced in PM management, conserving scarce PM write bandwidth.
- We implement NBTree and our evaluation results show that NBTree achieves up to 11× higher throughput and 43× lower 99% tail latency than state-of-the-art counterparts under YCSB workloads.

II. BACKGROUND AND MOTIVATION

In this section, we first introduce the background of persistent memory and eADR mechanism (Section II-A). Then, we analyze how to minimize the PM access overhead in eADR-enabled platforms (Section II-B). Finally, we describe the challenges of designing lock-free persistent data structures (Section II-C).

A. Persistent Memory and eADR

Persistent memory (PM), which is now commercially available, provides many attractive features, such as byte-addressability and data persistency. However, PM still has higher latency and lower bandwidth than DRAM. To reduce write latency, existing PM-based systems utilize the ADR mechanism [2] to drain the writes sitting on the write pending queues (WPQ) to PM by the reserved energy during a power outage. Therefore, data that reaches the WPQ of ADR-based platforms is considered persistent, whereas data in the CPU cache remains volatile. As a result, an additional pair of the flush instruction (e.g., `clwb`, `clflush`, and `clflushopt`) and memory barrier (e.g., `mfence`, and `sfence`) is necessary for programmers to guarantee data persistency [16].

Fortunately, eADR is supported on the 2nd generation Intel Optane DCPMM with the 3rd generation Intel Xeon Scalable Processors. eADR-enabled platforms reserve more energy that enables them to flush data in CPU cache to PM after a power failure, thereby expanding the persistence domain to include CPU cache [21].

eADR offers the following advantages over ADR. The first one is reducing PM write overhead. With persistent CPU cache, synchronous cacheline flush instructions are no longer necessary, which reduces PM write overhead in two aspects. (1) Reducing the latency in the critical path. Previously, flush instructions and memory barriers result in up to an order of magnitude higher latency in the critical path [7]. (2) Saving PM write bandwidth. Write operations with high locality can hit the

TABLE I
PM OVERHEAD PRODUCED WHEN ACCESSING B⁺-TREE'S LEAF NODES

	Flush	PM line write	PM line read
NVTree [26]	2/2/2	2/2/2	O(n)
WB ⁺ Tree [27]	4/3/3	3/2/2	O(log(n))
FPTree [4]	3/1/3	3/1/3	3
RNTree [28]	2/1/2	3/1/3	O(log(n))
BzTree [15]	15/7/10	11/6/7	O(log(n))
FAST&FAIR [29]	O(n)/O(n)/1	O(n)/O(n)/1	O(n)
uTree [5]	2/1/1	2/2/1	2
NBTree	1/1/1	1/1/1	1

a/b/c indicates the PM overhead of an individual insert/delete/update. *n* indicates the number of key-value pairs in the leaf node.

CPU cache without flushing the modifications to PM, reducing the bandwidth consumption.

The second one is facilitating the lock-free design. Data structures can atomically modify and persist data with eADR, facilitating the lock-free design in PM. Most lock-free data structures [22], [23], [24], [25] rely on atomic CPU hardware primitives, such as CAS. However, in ADR-based platforms, those primitives can atomically modify data but cannot ensure their persistence because the CPU cache is volatile. Threads are likely to read unpersisted data in CPU cache, resulting in the *dirty read* anomaly. With eADR, the globally visible data in CPU cache is ensured to be persisted. Thus, it is possible to modify and persist data atomically.

B. PM Overhead Analysis

The performance gap between PM and DRAM encourages people to design PM-friendly storage systems to reduce I/O overhead. Previous works [30], [31], [32], [33], [34], [35], [36], [37], [38] designed for ADR-based platforms mostly focused on reducing the costly flush instructions. With eADR, flushing is no longer required, which dramatically reduces the latency of PM writes. Meanwhile, PM writes with high locality can hit the CPU cache, which conserves the PM write bandwidth. However, dirty cachelines will eventually be evicted to PM according to the cache replacement policy. Excessive PM writes with low locality still result in high latency due to the poor PM write bandwidth. Besides, PM also has higher read latency than DRAM. Thus, the unique features of eADR require a rethinking of how to reduce PM overhead.

We conclude the following three design goals to reduce PM overhead in eADR-enabled platforms. First, reducing the number of *PM line writes* (the 64-byte aligned PM line written in CPU cache). Reducing *PM line writes* can generate fewer dirty cachelines, thereby saving PM write bandwidth. Second, increasing the access locality of write operations. This allows the write operations to hit the CPU cache more frequently, reducing PM writes in eADR-enabled platforms. ADR-based platforms do not benefit much from it because write operations are required to be synchronously flushed to PM. Third, reducing the number of *PM line reads*. The relatively higher read latency of PM is non-negligible, especially in read-intensive workloads.

Existing persistent indexes fall short of the aforementioned design objectives due to excessive *PM line read/writes* and poor write locality. Table I shows that the base operations in

the state-of-the-art persistent B⁺-Trees produce considerable *PM line read/writes* when accessing the leaf nodes. Excessive *PM line writes* consume scarce PM write bandwidth, while too many *PM line reads* incur high latency. Moreover, due to the assumption that CPU cache is volatile, existing persistent indexes do not focus on increasing write locality. Hence, they can not fully utilize CPU cache to reduce PM writes in eADR-enabled platforms. Meanwhile, we notice that the strategies of reducing the number of flushes in ADR-based platforms sometimes result in fewer *PM line writes*. However, they are not equivalent. For example, RNTree [28] proposes a selective metadata persistence technique, which reduces the number of flushes but cannot diminish *PM line writes*.

In addition to the PM overhead that occurred during index operations, persistent indexes also incur considerable PM writes during the PM allocation/deallocation [8], [39]. To persistently record the allocation state of each PM chunk, persistent allocators often produce a large number of random PM writes to modify various metadata. Despite the fact that flush instructions are no longer required with eADR, these metadata modifications still consume a significant quantity of PM write bandwidth due to the poor access locality.

C. The Opportunities and Challenges of Lock-Free Persistent Data Structures

As we analyzed in Section II-A, eADR facilitates the design of lock-free persistent data structures by preventing the *dirty read* anomaly resulting from concurrent threads reading unpersisted data in CPU cache. In eADR-enabled PM systems, lock-free concurrency protocols not only enhance the scalability but also ensure the crash consistency of persistent data structures. Lock-free data structures always keep in a consistent state through atomic updates or temporarily reside in an inconsistent state that can be fixed by concurrent threads using a cooperative mechanism. Therefore, concurrent operations proceed without the need for lock serialization, improving the scalability in multi-core systems. Meanwhile, the eADR mechanism guarantees instant durability for each visible state transformation in the CPU cache. Consequently, after a system crash, lock-free data structures will be found in either a consistent state or an inconsistent state that can be fixed by the subsequent operations.

However, even with eADR, it's non-trivial to design lock-free data structures (LFD). The major challenge lies in the hardware restrictions on CPU atomic primitives, which can only atomically modify a single word (8-byte). In contrast, a single state transformation in non-trivial data structures often needs to write multiple words. Therefore, data structures are likely to expose intermediate states to concurrent threads. Those problems may result in anomalies such as *lost update* and *inconsistent read*. For example, structural modification operations, such as split, are the most complex state changes in B⁺-Trees. During the split, B⁺-Tree transfers the content of the old node to newly allocated nodes and then replaces the old node with new nodes. As the split cannot be completed atomically, a concurrent update may occur in the old node but be missed in the new nodes. In this situation, new nodes are facing the risk of the *inconsistent read* anomaly since they are stale. Even worse, if the update is not

properly synchronized to new nodes, it will be lost permanently, incurring the *lost update* anomaly.

Furthermore, existing lock-free concurrency protocols fall short of fully optimizing the performance of persistent data structures since they do not consider the unique PM characteristics and the emerging eADR feature. DRAM-based lock-free data structures are not inherently tailored to accommodate PM-specific characteristics discussed in Section II-B, such as high read latency and low write bandwidth. Hence, they tend to exhibit significantly reduced performance when applied to PM, largely attributed to their excessive memory accesses. For example, Bw-tree [23], a DRAM-based B+Tree, achieves lock-free concurrency through a delta update policy. During the updates, Bw-tree atomically links a delta record to the tree node via a pointer. However, this approach leads to multiple small random PM writes for each write operation and dramatically increases the pointer chasing length for read operations. Moreover, for skewed workloads, the delta update policy fails to leverage CPU cache to absorb frequent updates of hot key-value entries, unlike what NBTree accomplishes. Furthermore, Bw-tree stores the logical pointers in the tree node and maintains a mapping table to transform the logical addresses of tree nodes to physical addresses. Such an indirection introduces additional overhead to memory accesses. Consequently, as evaluated by Arulraj [15], directly deploying Bw-tree in PM results in inferior performance. For persistent memory PM-based lock-free data structures intended for ADR-enabled systems, there exists a substantial overhead in ensuring crash consistency. For example, BzTree [15] achieves lock-free concurrency by using PMwCAS [40], an extension of MwCAS [41] that guarantees both atomicity and durability of writes. However, BzTree exhibits inferior performance compared to lock-based B⁺-Trees. This degradation is attributed to the excessive reads and writes on persistent memory induced by the persistence guarantee offered by PMwCAS.

III. PM-FRIENDLY B⁺-TREE

NBTree achieves low latency and high scalability by lowering PM overhead in the following two aspects: (1) Reduce the number of *PM line reads/writes* during each index operation. (2) Increase the access locality of write operations to absorb PM writes in persistent CPU cache.

In this section, we describe the PM-friendly design of NBTree. We first present the overall architecture (Section III-A), and then describe the base operations of NBTree (Section III-B). Finally, we introduce our cache-crafty persistent allocator (Calloc) for NBTree (Section III-C).

A. NBTree Structure

The overall architecture of NBTree is shown in Fig. 1. NBTree employs a decoupled leaf node architecture, which separates the leaf node into a metadata layer and a key-value layer. The metadata layer, as well as the inner nodes of NBTree, is maintained in DRAM. They can be rebuilt from the persistent key-value layer of leaf nodes in PM during recovery. The decoupled leaf node design enables NBTree to absorb metadata operations in DRAM, reducing PM line accesses drastically.

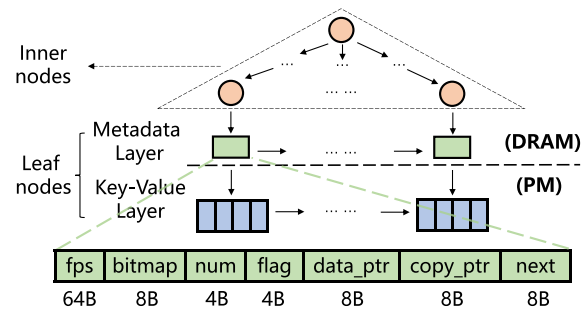


Fig. 1. Overall architecture of NBTree.

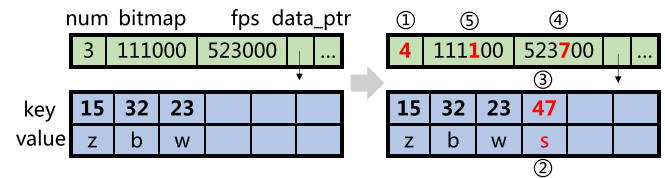


Fig. 2. Procedure of an insertion on an NBTree's leaf. (The fingerprints (fps) of the keys are set to $key\%10$ for brevity).

Specifically, for leaf nodes, both the metadata layer and the key-value layer are linked into a singly linked-list. In the key-value layer, each key-value block is an unsorted array of key-value entries. Each key-value entry stores a 64-bit key and a 64-bit payload. The highest 2 bits (*copy_bit* and *sync_bit*) of the payload are reserved for concurrency control. For variable-sized key-value entries, NBTree stores pointers that indicate the actual keys or values. Each leaf's metadata consists of the following fields: (1) *fps* to store the one-byte fingerprint (hash value) for each key in the leaf, which speeds up key-search on the unsorted array. (2) *num* to store the number of entries occupied by both the committed and in-flight insertions, which handles concurrent insertions. (3) *bitmap* to track the position of the committed insertions in the leaf. (4) *data_ptr* to indicate the address of its key-value block. (5) *copy_ptr* to store the address of newly allocated leaves when the leaf performs SMO. (6) *flag* to track the status of SMO. (7) *next* to indicate the address of the sibling leaf. For inner nodes, NBTree adopts the structure of FAST&FAIR [29].

B. Base Operations

NBTree reduces the overhead of base operations (insert, update, delete, and search) by minimizing PM line accesses and maximizing the cache utilization. For each base operation, NBTree first locates the corresponding leaf by searching the inner nodes in DRAM. Then, it adopts *log-structured insert*, *in-place update/delete*, and *efficient search* to reduce the average number of *PM line read/writes* on the persistent leaf node to 1 and increase the access locality.

Log-structured Insert: Insertions perform in a log-structured manner in NBTree. Fig. 2 illustrates the steps of inserting a new key-value pair in NBTree's leaf. First, NBTree increases the number (①) to occupy the next free slot. Then, NBTree

writes the value (②) and the key (③) to the occupied slot. After writing the key, the new insertion can survive a power failure. Finally, NBTree updates the *fps* (④) and *bitmap* (⑤) to make insertion visible. We observe that the only PM overhead in an insertion is storing a key-value pair. Moreover, with eADR, the *log-structured insert* manner also allows the consecutive insertions on the same leaf to combine in persistent CPU cache, reducing PM writes.

In-place Update/Delete: Conventional log-structured B⁺-Trees, such as NVTree [26] and RNTree [28], update or delete key-value entries by appending new entries. NBTree, on the other hand, performs *in-place update/delete*. To update a key-value entry, NBTree modifies its value in-place. To delete an entry, NBTree invalidates it by resetting its key to 0. Update and delete can survive system crashes by modifying and persisting the 8-byte key or value with eADR support. In-place update manner is not favored in ADR-based platforms, as repeatable flushes to the same cacheline cause extra latency especially running on skewed workload [13]. With eADR, *in-place* updates minimize *PM line writes* and increase the write locality of frequently accessed key-value entries.

Efficient Search: The search range is confined to the valid entries indicated by the *bitmap*. This ensures that the entry found by the search operation is persistent and committed. NBTree further narrows the average number of candidate entries to one by checking the *fingerprints*. Finally, NBTree scans the candidate entries to filter the unmatched keys and the deleted keys. In most cases, the search operation produces only one *PM line read*, since the candidate entry is often unique, and the metadata is stored in DRAM.

Crash Consistency: NBTree can restore its metadata layer and inner nodes using the key-value layer after a crash. During recovery, NBTree scans the list of key-value blocks and labels the slots with non-zero keys as the valid key-value entries. Then, NBTree rebuilds the metadata layer and the inner nodes based on those valid entries.

NBTree maintains consistency even if a crash occurs in the middle of a write operation (insertion/update/deletion). As shown in Fig. 2, during an insertion, writing the key (③) happens after writing the value (②). If the crash happens after writing the key, the intact key-value pair will survive. Otherwise, the in-flight insertion will not leave NBTree in an inconsistent state because the key of the occupied slot is still 0, which is discarded after the recovery. For updates and deletions, they can be completed by atomically modifying 8-byte keys or values, without exposing intermediate state.

C. Cache-Crafty Persistent Allocator

In this section, we propose a cache-crafty persistent allocator (Calloc) to manage PM allocation/deallocations of NBTree. As shown in Fig. 3, Calloc employs a two-layer architecture and offers an allocation *bitmap*, reclaimed ring buffer, and recyclable log to increase the write locality during allocation, deallocation, and logging. Consequently, most of PM writes for the memory management can be absorbed in persistent CPU cache, saving PM write bandwidth in the eADR-enabled platforms.

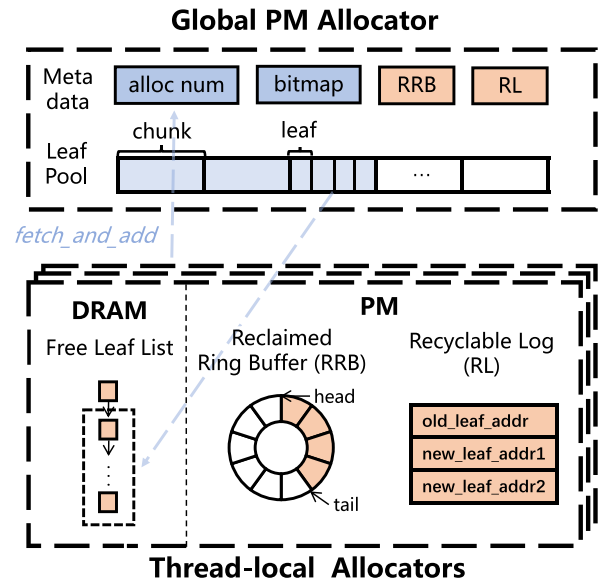


Fig. 3. Overall architecture of Calloc.

Two-layer Architecture: As shown in Fig. 3, Calloc is composed of a coarse-grained global PM allocator and multiple fine-grained thread-local allocators. The global PM allocator contains a leaf pool that is managed in coarse-grained large memory chunks (e.g., 16 MB) and various metadatas for crash consistency. The thread-local leaf allocators locally manage the leaf allocations and deallocations for each worker thread to reduce thread-contentions. Each worker thread asynchronously requests large memory chunks from the global memory pool on demand by using *fetch_and_add* to atomically increase *alloc_num*, which has little persistence overhead since it happens infrequently. Thread-local leaf allocators divide the large memory chunk into multiple leaf nodes and push the continuous leaf addresses into the volatile free leaf lists to serve the leaf allocations.

Allocation Bitmap: To persistently record the allocation state of each leaf node in the free leaf lists with minimal PM writes, Calloc maintains an allocation *bitmap* in the global memory pool. Each bit in the *bitmap* indicates whether a leaf node in the global memory pool is allocated. After allocating a leaf address from the thread-local free leaf list, Calloc sets the corresponding allocation bit of the *bitmap*. Due to the locality of allocations in Calloc, continuous allocations are likely to modify the *bitmap* in the same cacheline. In eADR-enabled platforms, those *write combining* in CPU cache dramatically reduce PM writes of allocations.

Reclaimed Ring Buffer: Calloc proposes a thread-local reclaimed ring buffer design to minimize PM writes during the garbage collection. Although Calloc can ensure the locality of updating the *bitmap* during the allocations, the deallocations tend to modify the *bitmap* randomly, producing expensive random PM writes. Therefore, Calloc employs a persistent reclaimed ring buffer to store the addresses of recently freed leaf nodes. Specifically, during the deallocation, Calloc appends

the freed leaf address to the tail of the ring buffer. As for the allocations, Calloc first consumes the freed leaf nodes in the header of the ring buffer before accessing the free leaf list. In this way, the random PM writes to the `bitmap` during the garbage collection can be avoided when the reclaimed ring buffer is not full. Meanwhile, the small reclaimed ring buffers are frequently accessed during the allocations and deallocations of leaf nodes. As a result, writes on reclaimed ring buffers tend to hit CPU cache, which does not consume PM write bandwidth in eADR-enabled platforms.

Recyclable Log: Calloc introduces the recyclable log design to address the memory leak problem that might happen in the structural modification operations (SMOs, e.g., split) with little persistence overhead. During SMO, NBTree first allocates new leaves, then performs three-phase SMO (Section IV-B) to replace the old leaf with new leaves, and finally reclaims the old leaf. The memory leak will occur if the crash happens when the allocated leaves are not linked into NBTree or the old leaf unlinked from NBTree is not reclaimed.

To resolve this problem in SMO, Calloc records the addresses of both old leaf and new leaves in the thread-local recyclable log before updating the `bitmap` to persist the allocation of new leaves. After completing SMO and the deallocation of old leaf, Calloc clears the log information so that it can be reused for the subsequent SMOs. In this way, Calloc can reclaim the potentially leaked leaf nodes during the recovery procedure by accessing the log entry. Meanwhile, Calloc is able to reuse the same memory space of log entry for SMOs that happened in the same worker threads. Due to frequent accesses, the recyclable log tends to reside in the persistent CPU cache. Hence, PM writes to recyclable logs can be absorbed in the persistent CPU cache with the support of eADR.

Discussion: To make Calloc more adaptable to general scenarios, Calloc can maintain thread-local free block lists with different sizes, similar to buddy systems. Moreover, for multi-socket systems, Calloc creates separate memory pools for each NUMA node to reduce expensive cross-NUMA accesses, which is inspired by PACTree [42].

IV. LOCK-FREE DESIGN

In this section, we introduce the lock-free concurrency control of NBTree, which is based on a precondition guaranteed by the eADR-enabled platforms: **globally visible data is persistent**. We propose different concurrency-control protocols for operations on normal leaf nodes (Section IV-A), leaf nodes during SMO (Section IV-B), and inner nodes (Section IV-C).

A. Leaf Node Operations

We divide the base operations in NBTree into two categories. The first category is `insert`, which appends new data to the free slot. The second category is UDS operations, including update, deletion, and search. The UDS operations always work on the committed insertions. In the following, we discuss how NBTree resolves the `insert-insert`, `UDS-UDS`, and `insert-UDS` conflicts.

TABLE II
APPROACHES EMPLOYED DURING DIFFERENT PHASES OF SMO TO FACILITATE LOCK-FREE LEAF NODE OPERATIONS

SMO Phase	Copy	Sync	Link
Update/Delete	three-phase SMO (sync phase)	sync-on-write	
Search	unnecessary	sync-on-read	unnecessary
Insert	cooperative SMO		

Insert-Insert Conflicts: We use atomic primitives to serialize concurrent insertions. As shown in Fig. 2, to begin an insertion, NBTree uses the `fetch_and_add` to atomically increase the `num`, occupying the next free slot. This ensures that concurrent insertions are placed in separate slots. At the end of an insertion, NBTree uses `CAS` to atomically update the `bitmap`, which commits the insertion. In this way, NBTree achieves lock-free insert on the leaf nodes.

Insert-UDS Conflicts: Those conflicts are naturally solved in NBTree. First, an insertion always writes the data into the unused space, which does not affect the UDS operations. Second, NBTree commits an insertion by atomically updating the `bitmap`, which makes the new insertion visible to UDS operations. Therefore, UDS operations always operate on the completed insertions.

UDS-UDS Conflicts: UDS-UDS conflicts in NBTree are resolved in eADR-enabled platforms by using atomic primitives. As mentioned above, updates and deletions only need to modify 8-byte keys or values. Therefore, they can be completed atomically without exposing the intermediate states. With persistent CPU cache, those modifications can also be persisted at the same time. Thus, the order of committing and visibility for concurrent updates and deletions are always maintained, and the search operation always reads the latest committed data.

B. Structural Modification Operations

Structural modification operations (SMOs) are initiated when a key-value entry is inserted into a full leaf. The conventional procedure of SMO is to copy the entries from the full old leaf to the newly allocated leaves, and then replace the old leaf with the new leaves. However, since the copy phase cannot be completed atomically, lock-free concurrent modifications to the old leaf may not be synchronized to the new leaves, resulting in the *lost update* anomaly. Moreover, the lock-free search might read *dirty* or *stale* data due to the inconsistency between the old leaf and new leaves.

Table II shows the approaches used by NBTree to resolve the potential anomalies and facilitate lock-free accesses. In NBTree, SMO is divided into three phases (copy phase, sync phase, and link phase). During each phase, different approaches are used to handle concurrent operations on the SMO leaf. For UD (update/delete) operations, NBTree resolves the *lost update* anomaly with the sync phase of the SMO and the *sync-on-write* technique. For search operations, NBTree uses *sync-on-read* to prevent the *inconsistent read*. We also propose *cooperative SMO*, which enables concurrent insertions to complete SMO cooperatively.

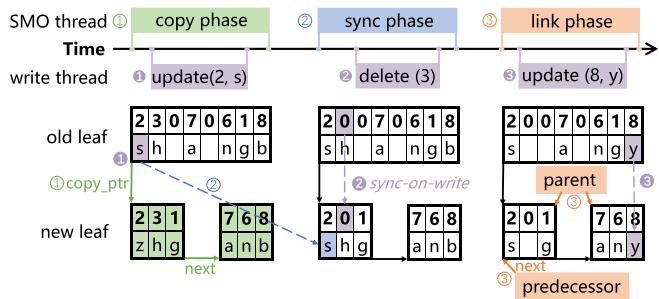


Fig. 4. Procedure of *three-phase SMO* and *sync-on-write* when updates and deletions operate on an SMO leaf.

Three-phase SMO: As shown in Fig. 4, different from the SMO of traditional B^+ -Trees that only includes the copy phase and link phase, NBTree adds a sync phase to avoid the *lost update* anomaly caused by UD operations during the copy phase. In the following, we will describe the procedure of each phase.

In the copy phase, SMO copies the valid entries with non-zero keys in the full leaf to new leaves. As shown in Fig. 4, NBTree allocates two new leaves if the number of valid entries exceeds a certain threshold (half of the leaf capacity by default). Otherwise, only one new leaf is allocated. Then, NBTree distributes key-value pairs to the new leaves and constructs their metadata layer. Finally, NBTree sets the `copy_ptr` in the old leaf to indicate the address of the first new leaf.

In the sync phase, NBTree synchronizes the lost UD operations to new leaves. During the copy phase, concurrent UD operations still write to the old leaf. As the copy phase cannot be completed within an atomic instruction, those UD operations, such as `update(2, s)` in Fig. 4, might not have been migrated to new leaves yet. Therefore, in the sync phase, NBTree employs CAS to synchronize the missed UD operations to new leaves.

The link phase replaces the old leaf in NBTree with new leaves. NBTree first links new leaves into the singly linked-list of the key-value layer and the metadata layer by changing the `next` pointer of the previous leaf. Then, NBTree installs new leaves to the parent node (described in Section IV-C).

Sync-on-write: In the post-copy phases of SMO, UD operations resolve the *lost update* anomaly by adopting a *sync-on-write* approach, which actively synchronizes the modification from the old leaf to the new leaf. Specifically, for an update, after modifying a key-value in the old leaf, it re-searches the target key in the new leaf. If the corresponding value in the new leaf is not up-to-date, NBTree synchronizes the latest update to the new leaf using CAS. With persistent CPU cache, CAS can atomically modify and persist the synchronization. Similar to the update, the deletion also re-executes in the new leaf if it contains the target key. NBTree imposes low-overhead on *sync-on-write* because it only incurs one additional search on the new leaf and one CAS primitive.

During the post-copy phases of SMO, *sync-on-write* prevents lock-free UD operations from suffering the *lost update* anomaly. As illustrated in Fig. 4, during the copy phase, any UD operation that happens on the old leaf (e.g., `update(2, s)`)

TABLE III
LATEST AND CLEAN LEAVES IN DIFFERENT PHASES OF SMO

SMO Phase	Latest		Clean	
	old leaf	new leaf	old leaf	new leaf
Copy	✓		✓	
Sync	✓			✓
Link	✓	✓		✓
After SMO		✓		✓

The latest leaf contains all committed writes. The clean leaf does not contain any uncommitted dirty write.

will be synchronized to the new leaf in SMO's sync phase. However, UD operations happen after the copy phase (e.g., `update(8, y)`, `delete(3)`) may still be lost. To avoid the *lost update* anomaly, UD operations need to actively synchronize the modification by calling *sync-on-write*.

Through *sync-on-write* and *three-phase SMO*, we ensure that NBTree always maintains a consistent state that includes all committed operations after a crash. As we previously mentioned, SMO's *durability point* is when the new leaves replace the old leaf in PM by linking themselves into the key-value layer during the link phase. If a crash occurs before the *durability point*, the old leaf will remain in the key-value layer. During SMO, any modification must first operate on the old leaf. Therefore, as illustrated in Table III, the old leaf always holds both the *latest* committed and uncommitted operations during SMO. The uncommitted operations in the old leaf are consistent since they come from the UD operations that do not finish the *sync-on-write*. The *sync-on-write* is unnecessary since the new leaves are discarded when the crash happens before *durability point*. If a crash occurs after the *durability point*, the new leaf will be linked into the key-value layer. At that time, all UD operations committed in the copy phase have already been synchronized to the new leaf in the sync phase. For UD operations that happen after the copy phase, they are only committed when they write to a new leaf. As a result, new leaves hold the *latest* committed operations after the *durability point*. To summarize, the consistent leaf with the *latest* committed operations will survive whenever the crash happens.

SMO threads (sync phase) and UD threads (*sync-on-write*) may synchronize the same value to the new leaf concurrently. NBTree can serialize those synchronizations using the highest two bits of each entry's value. We will discuss this scenario in Section V.

Sync-on-read: To deal with the potential inconsistency between the old leaf and the new leaves, the search operations employ the *sync-on-read* approach to synchronize the corresponding key-value entries from the old leaf to the new leaf. Specifically, NBTree searches the target key in both old and new leaves. If the returned results differ, the search operation updates or deletes the key-value in the new leaf to match the one in the old leaf. The overhead of the *sync-on-read* is as low as the *sync-on-write*.

Sync-on-read guarantees that a lock-free concurrent search returns the latest and committed version of a key-value entry. During SMO's sync phase, reading from either old or new leaves without performing *sync-on-read* may lead to the *inconsistent*

read. As illustrated in Table III, in the sync phase, the old leaf is possibly *dirty* because the UD operations might have not committed due to an on-going *sync-on-write*. Meanwhile, the new leaf is likely *stale* as the SMO thread may not have finished synchronizing the *latest* modification that happened during the copy phase. To address this problem, the search operation uses *sync-on-read* to synchronize the *latest* key-value from the old leaf to the new leaf. It makes sure that the target key-value pair in the new leaf is both the *latest* and *clean* before returning the search result.

Search operations on the leaf that is not in the sync phase can directly read the correct value without calling *sync-on-read*. Table III shows the destination of reads, which is the leaf that holds both the *latest* and *clean* key-value pairs. During the copy phase, incoming reads go to the old leaf, which is both the *latest* and *clean* since concurrent UD operations directly commit in the copy phase. After the sync phase, reads go to the new leaf. This is because previous UD operations that happened in the copy phase have already been synchronized to the new leaf, while later UD operations are committed once they are visible in the new leaf.

Cooperative SMO: In NBTree, concurrent insertions to the leaf during SMO employ *cooperative SMO*. The insertion thread that encounters a leaf with an in-flight SMO will help complete its SMO before continuing. NBTree uses atomic primitives, such as CAS, to coordinate multiple SMO threads, making sure only the fastest modification can be visible. In this way, instead of waiting for the completion of SMO, NBTree guarantees that SMO moves forward at the fastest speed, even when a certain SMO thread is suspended.

Specifically, in the copy phase, multiple SMO threads prepare new nodes respectively and use CAS primitive to atomically install the `copy_ptr`. In the sync phase, the synchronization of each key-value entry can also be completed cooperatively by using CAS primitive. In the link phase, NBTree uses CAS to link the new leaf into the metadata layer and the key-value layer. Then, NBTree uses HTM (described in Section IV-C) to atomically update the parent node.

C. Inner Node Operations

We propose a *shift-aware search* algorithm and use the *HTM-based update* to coordinate concurrent inner node operations.

HTM-based Update: NBTree uses HTM to atomically update inner nodes following FPTree [4]. HTM is an optimistic concurrency control tool that uses hardware transactions to make multiple writes atomically visible in a lock-free manner. HTM transaction only aborts when data conflicts are detected. Wrapping updates in HTM is efficient because the conflict of inner node modifications rarely happens. Moreover, updates do not expose intermediate states to other threads, which avoids the read thread viewing the inconsistent state.

Shift-aware Search: Although the modifications to the inner nodes are atomic, lock-free inner node search might find the wrong child pointer due to the read-write conflict. As illustrated in Fig. 5, the search operation performs a linear search (`Search(18)`) to retrieve the key of 20 (②). However, before

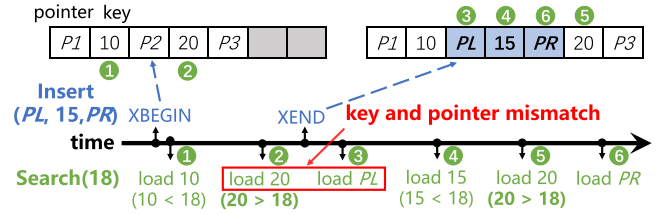


Fig. 5. *Shift-aware search* on the inner node under the read-write conflict. (XBEGIN means the start of the transaction, XEND means the end of the transaction).

it fetches the corresponding pointer $P2$, an insertion (`Insert(PL, 15, PR)`) modifies the node. Therefore, the search operation loads the wrong pointer PL (③) instead of $P2$ or PR . As a result, an existing key might be missed in the search operations.

NBTree proposes a *shift-aware search* algorithm to ensure the correctness of lock-free inner node search despite concurrent write transactions (insertion or deletion) on the same node. As shown in Fig. 5, before proceeding to the next level of the tree through PL , NBTree checks if the fetched key (20 in ②) has been shifted by concurrent write transactions. If so, NBTree re-searches the node from the current position (④-⑥), making sure that the target key lies within the sub-tree indicated by the returned pointer.

The *shift-aware search* algorithm always finds the correct pointer for the following two reasons. First, NBTree keeps searching and data shifting in the same direction. As shown in Fig. 5, during the search, the insertion shifts the data from left to right. If the search proceeds in the same direction, then it never misses the newly inserted key. Inspired by FAST&FAIR, we maintain a `switch_counter` in each node, which is increased when the insertion and deletion on the inner node take turns. Second, we adopt the concurrency protocol of the B-link tree [43] to handle SMO. NBTree maintains a `high_key` (the largest key in the node) and `sibling_ptr` in each inner node. NBTree re-searches in the sibling node if the target key is less than the `high_key`, which indicates an SMO has happened on the node.

V. IMPLEMENTATION

In this section, we first describe the implementation of NBTree: insert (Section V-A), update/delete (Section V-B), and search (Section V-C). Then we discuss the generality of our work (Section V-D).

A. Insert

Algorithm 1 describes the insert operation. After locating the target leaf, NBTree occupies the next free slot using the atomic primitive (Line 2). If the leaf is full, NBTree initiates SMO by setting the `smo_bit`, the highest bit of the `bitmap` (Line 3-5). NBTree commits the insertion by updating the `bitmap` using CAS (Line 8).

Algorithm 1: Insert(K key, V val).

```

1 leaf = findLeaf(key);
2 pos = fetch_and_add(&leaf→num, 1);
3 if pos ≥ LEAF_NODE_SIZE then
4   leaf→setSMOBit();
5   SMO(leaf);
6   goto Line 1;
7 leaf→insert(key, val, pos);
8 leaf→setBitmap(pos);

```

Algorithm 2: SMO(Leaf Leaf).

```

1 if ! leaf→copy_ptr then // Copy phase
2   splitKey = leaf→findSplitKey();
3   leftLeaf, rightLeaf = allocNewNode();
4   while (entry = leaf→next()).key != 0 do
5     entry.val |= copy_bit; // Set the copy_bit
6     copy(entry, leftLeaf, rightLeaf, splitKey)
7   end
8   setMetadata(leftLeaf, rightLeaf);
9   CAS(&(leaf→copy_ptr), NULL, leftLeaf);
10 if ! leaf→flag.sync then // Sync phase
11   while (entry = (key, val) = leaf→next()).key!=0 do
12     syncLeaf = key < splitKey ? leftLeaf : rightLeaf;
13     (k, v) = syncEntry = syncLeaf→next();
14     if k == key then
15       if (v & ~(copy_bit | sync_bit)) != (val & ~
16         ~(copy_bit | sync_bit)) then
17         CAS(&syncEntry.val, v|copy_bit,
18           val|sync_bit);
19     else if leaf→flag.link then return ;
20     else if ! entry.key then syncLeaf→prev() ;
21     else
22       syncEntry.key = 0;
23       mfence();
24       goto Line 13;
25   end
26 leaf→flag.sync = 1;
27 if ! leaf→flag.link then // Link phase
28   pred = findPredLeaf(key);
29   CAS(&(pred→data_ptr→next), leaf→data_ptr,
30     leftLeaf→data_ptr);
31   CAS(&(pred→next), leaf, leftLeaf);
32   if pred→isSMO() then
33     SMO(pred);
34     goto Line 25;
35   xbegin(); // Start an HTM transaction
36   if ! leaf→flag.link then
37     update_parent(leaf);
38     leaf→flag.link = 1;
39   xend();

```

The procedure of SMO is listed in Algorithm 2. In the copy phase (Line 1-9), NBTree distributes valid entries with non-zero keys to newly allocated leaf nodes and constructs the metadata layers for the new leaves (Line 4-8). For each copied entry, NBTree sets the highest bit (`copy_bit`) of the value (Line 5). The `copy_bit` indicates that the value is copied from the old leaf in the copy phase, which will be used in the sync phase to avoid the repeatable synchronizations of an update.

In the sync phase (Line 10-24), NBTree sequentially obtains the valid entries in the old leaf (`entry`, Line 11) and new

Algorithm 3: Update(K key, V val).

```

1 leaf = findLeaf(key);
2 entry = leaf→update(key, val);
3 if leaf→isSMO() and leaf→copy_ptr then
4   (nKey, nVal) = nEntry = leaf→copy_ptr→find(key);
5   if nKey and ((v=nVal) & ~(copy_bit | sync_bit)) !=
6     (val=entry.val) and v & (copy_bit | sync_bit) then
7     if ! CAS(&nVal, v, val|sync_bit) then
8       goto Line 5;
9   return true;

```

Algorithm 4: Delete(K key).

```

1 leaf = findLeaf(key);
2 leaf→delete(key);
3 if leaf→isSMO() and leaf→copy_ptr then
4   nEntry = leaf→copy_ptr→find(key);
5   if nEntry then
6     nEntry.key = 0;
7     mfence();
8   return true;

```

leaves (`syncEntry`, Line 13). If their keys match but values mismatch (Line 14-16), NBTree uses CAS to synchronize the lost update. Meanwhile, NBTree clears the `copy_bit` and sets the `sync_bit` (the second highest bit) of the target value in the new leaf (Line 16). CAS will abort if the `copy_bit` of the target value has been cleared. This ensures that any key-value pair is synchronized at most once during the sync phase. The `sync_bit` indicates the value is synchronized from the old leaf, which will be used in `sync_on_write` to avoid overwriting the latest updates. If the key of `entry` and `syncEntry` mismatch (Line 17-22), NBTree synchronizes the lost deletion because it usually indicates the key of `syncEntry` has been deleted in the old leaf (Line 19-22). However, there are two exceptions that the key of `entry` has been deleted in the new leaf. (1) SMO has been completed by another SMO thread (Line 17). As a result, the latest operations directly delete the entry in the new leaf without having to go through the old leaf. In this case, NBTree directly aborts SMO. (2) A concurrent operation deletes the key of `entry` on the old leaf after the `entry` has been read (Line 18). Meanwhile, the deletion has been synchronized by other threads before `syncEntry` is read. In this case, synchronization is not required.

In the link phase (Line 25-36), NBTree uses CAS to link the leaf to both the key-value layer and the metadata layer (Line 27-28). If the previous leaf is in SMO, NBTree will join the *cooperative SMO* to avoid the lost update of `next` pointer (Line 29-31). Finally, NBTree employs HTM to update the parent node and set the `flag.link` atomically (Line 32-36).

B. Update/Delete

Algorithm 3 shows the process of the update operation. NBTree first performs an *in-place update* in the target leaf (Line 1-2). If the leaf is in the post-copy phases of its SMO and the target key exists in the new leaf but its value is not the latest, NBTree will perform *sync-on-write* via CAS (Line 3-7). *Sync-on-write* clears the `copy_bit`, which prevents the synchronization invoked by SMO threads from overwriting the

Algorithm 5: Search(K key).

```

1 leaf = findLeaf(key);
2 entry = leaf→find(key);
3 val = entry.key ? (entry.val & ~(copy_bit | sync_bit)) : 0;
4 if leaf→isSMO() and leaf→copy_ptr then
5   if leaf→flag.sync then
6     return leaf→copy_ptr→search(key);
7   (nKey, nVal) = nEntry = leaf→copy_ptr→find(key);
8   if ! nKey then return 0 ;
9   if entry.key == 0 then
10    nEntry.key = 0;
11    mfence();
12  else if (nVal & ~(copy_bit | sync_bit)) != val then
13    CAS(&(nEntry.val), nVal|copy_bit, val|sync_bit);
14  val = nEntry.key ? (nEntry.val & ~(copy_bit|sync_bit)) :
15  0;
16 return val;

```

current update. It also sets the `sync_bit` to distinguish itself from the update directly operated on the new leaf. When an update performs *sync-on-write*, the SMO might have been completed by other threads. At that time, the latest updates directly operate on the new leaf without setting `sync_bit` or `copy_bit`. *Sync-on-write* does not overwrite those new updates to keep the linearizability. Besides, if the value in the old leaf has been changed by new updates, NBTree will synchronize the latest one.

Algorithm 4 depicts the delete operation. Similar to the update, it will synchronize the deletion if necessary.

C. Search

The search operation is shown in Algorithm 5. For the inner node search (Line 1), we directly reuse the code of FAST&FAIR and add the key-checking procedure before returning the child pointer to detect if any update happens. For the leaf node search (Line 2-15), NBTree directly returns the search result on the target leaf if SMO is not taking place or it is in the copy phase. Otherwise, NBTree searches the target key in the new leaf (Line 4-14). NBTree performs *sync-on-read* if the SMO is in the sync phase and the search results in two leaves mismatch (Line 9-14).

D. Discussion

Due to the huge application value and mature ecosystem of persistent memory, many manufacturers are developing their PM products [44], [45], [46]. Moreover, emerging compute express link (CXL) [47], [48] is also promising to be used for memory capacity expanding. Although NBTree is deployed on Intel Optane DCPMM in this paper, our design can be generalized to other persistent memory products (the platforms need to support persistent CPU cache), volatile memory, and future CXL memory.

The decoupled leaf node design is applicable to other persistent memory products and CXL memory devices, which have higher latency and lower throughput than DRAM. Decoupling the metadata layer from the leaf nodes and storing it in fast DRAM can reduce the access of slow memory devices, thus improving B⁺-Tree performance.

The concurrency protocol of NBTree is not limited to being used solely on Optane DCPMM. Our lock-free design ensures the concurrent consistency of B⁺-Tree regardless of whether it is stored in volatile memory or persistent memory. Moreover, for PM-based B⁺-Tree, our concurrency protocol can further ensure crash consistency if CPU cache is persistent.

Calloc is also a generalized persistent allocator for any PM-equipped platforms with persistent CPU cache. Calloc can dramatically reduce the persistence overhead for PM management by absorbing most of PM writes in persistent CPU cache.

VI. EVALUATION

In this section, we evaluate the performance of NBTree against other state-of-the-art persistent B⁺-Trees. We first describe our experiment setup (Section VI-A). Then, we perform single-threaded evaluation (Section VI-B), multi-threaded evaluation (Section VI-C), and YCSB evaluation (Section VI-D). After that, we individually evaluate the contribution of each NBTree design component (Section VI-E). Then, we examine the overhead of persistent allocators during the index operations (Section VI-F). Moreover, we compare the performance of indexes in two persistence modes (eADR and ADR) to unveil the impact of eADR (Section VI-G). In addition, we evaluate the performance of NBTree on DRAM to demonstrate the generality of our design (Section VI-H). Finally, we integrate the evaluated trees into Redis, measuring the performance in real-world systems (Section VI-I).

A. Experiment Setup

Testbed: Our testbed machine is a dual-socket Dell R750 server with two Intel Xeon Gold 6348 CPUs, the third generation Xeon Scalable processors that support eADR and TSX. Each CPU has 28 cores and a shared 42 MB L3 cache, while each CPU core has a 48 KB L1D cache, 32 KB L1I cache, 1280 KB L2 cache. The system is equipped with 512 GB DRAM and 4 TB PM (eight 256 GB Barlow Pass DIMM per socket). Due to the significantly larger NUMA effects for PM than they are for DRAM [17], [18], we adhere to the experimental configuration of previous works [4], [5], [15], [28], [49], binding all threads to a single NUMA node by default and restricting their access to the local DRAM and PM to avoid NUMA effects. We install a PM-aware file system (Ext4-DAX) in `fsdax` mode to manage PM devices. Then, we map large files into the virtual address using PMDK [50] to serve tree nodes allocation. We evaluate the performance of two persistence modes, eADR and ADR. To persist a store, we use `clwb` and `mfence` in ADR mode and solely use `mfence` in eADR mode.

Compared Systems: We compare NBTree against seven state-of-the-art persistent B⁺-Trees, including NVTree, WB⁺ Tree, FPTree, RNTree, BzTree, FAST&FAIR, and uTree. We directly use the open-sourced code of uTree [51], FAST&FAIR [49], BzTree [52], and RNTree [53]. We borrow Liu's [53] implementations of WB⁺ Tree, FPTree, and NVTree. We skip the evaluation of the multi-threaded performance of NVTree and WB⁺ Tree as their implementations do not support concurrency control. For variable-sized keys, we only compare NBTree with

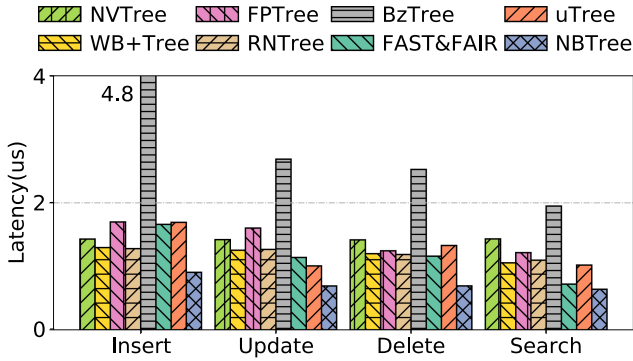


Fig. 6. Average latency of base operations. (Single thread, uniform access).

BzTree as the implementations of other trees do not support this function.

Default Configuration: We warm up each tree with 16 million key-value pairs and then run enough time for different workloads. By default, we use 8-byte keys and values. For variable-sized keys and values, we store them in the external memory region, and only keep the pointers (48-bit) in indexes to indicate their addresses. The node size of each tree is configured to 1 KB. We run all trees in eADR mode except in Section VI-G.

For fairness comparisons, all persistent indexes employ a naive persistent allocator by default, which pre-allocates sufficient memory space for each worker thread, thereby eliminating the performance impact of PM allocations/deallocations. In Section VI-E, we conduct a separate evaluation to assess the impact of persistent allocators on index performance.

B. Single Thread Evaluation

In this section, we evaluate the single-thread performance of base operations (search, insert, update, and delete) in eADR mode. We run individual operations under random key-access distribution and then calculate the average latency.

As shown in Fig. 6, NBTree achieves the lowest latency in every base operation. As the persistence overhead of PM writes is hidden by CPU cache in eADR mode, we attribute the good performance of NBTree to low *PM line reads*. In most cases, NBTree only causes one *PM line read* in each operation because it places the metadata of the leaf nodes in DRAM and uses fingerprints to filter the unmatched keys. The only PM overhead of NBTree comes from accessing the matched key-value pairs.

In contrast, as illustrated in Table I, other persistent B^+ -Trees produce more *PM line reads*, resulting in higher latency. We conclude the source of *PM line reads* in the following aspects: (1) Most of the B^+ -Trees (except uTree) need to access the metadata of the leaf node in each base operation. The metadata is often stored in different PM lines from the actual key-value pair, resulting in additional *PM line reads*. (2) Searching in the leaf node causes multiple *PM line reads*. FAST&FAIR and NVTree use linear search to locate the key-value pair, which needs to traverse half of the leaf on average. WB+Tree, RNTree, and BzTree perform the binary search, which has a similar PM

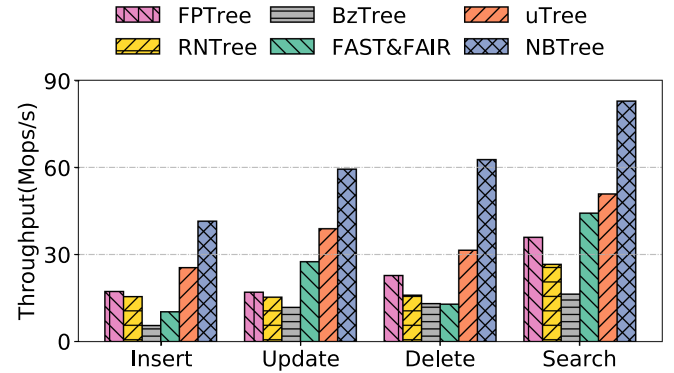


Fig. 7. Throughput of base operations. (56 threads, uniform access).

overhead to the linear search when the array size is small. (3) FAST&FAIR and BzTree produce extra *PM line reads* when they perform inner node search because they store inner nodes in PM. (4) uTree invokes additional *PM line reads* to access the sibling node in the linked list, which shows poor locality with the current node. (5) BzTree applies $PMwCAS$ [40] to atomically persist the modification. Each $PMwCAS$ produces multiple *PM line reads* to access a descriptor with a default size of 256 bytes.

C. Multi-Threaded Evaluation

We evaluate the multi-threaded performance of base operations under random key access distribution. As shown in Fig. 7, NBTree achieves the highest throughput in each operation. Compared with other trees, the throughput of NBTree in 56 threads is $1.6-7.5\times$ higher on insert, $1.5-5.0\times$ higher on update, $2.0-4.9\times$ higher on delete, $1.6-5.1\times$ higher on search. This is primarily because NBTree minimizes both *PM line reads* and writes. Reducing *PM line writes* in eADR-enabled platforms is important. The reason is that the modified PM lines in CPU cache are eventually evicted to PM with low bandwidth. Multi-threaded writes can saturate CPU cache and WPQ, resulting in high latency. Besides, excessive *PM line reads* also degrade the multi-threaded performance due to the high latency.

NBTree scales well in the multi-threaded evaluation as it limits the *PM line read/writes* per operation to 1 in most cases. The decoupled leaf node of NBTree absorbs the leaf metadata accesses in DRAM. Consequently, the base operations of NBTree only generate 1 *PM line read/write* to read or modify a single key-value entry in the persistent data layer.

As shown in Table I, other persistent indexes have lower scalability for their high *PM line read/writes*. We have analyzed the cost of PM reads in detail in Section VI-B. Compared with NBTree, other trees produce extra *PM line writes* in the following aspects: (1) They modify the persistent metadata of the leaf nodes for various usages, such as correct recovery, traversal acceleration, and concurrency control. (2) BzTree produces the most *PM line writes* because it needs to record a 256-byte descriptor in each $PMwCAS$, resulting in the lowest scalability. (3) FAST&FAIR causes additional *PM line writes* to maintain the order of leaf nodes. As a result, its throughputs on insertions and deletions are low.

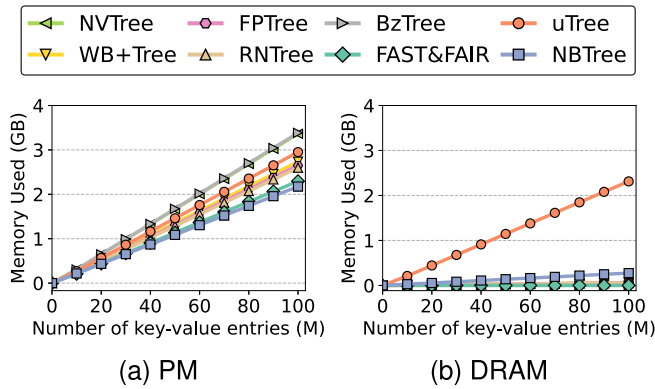


Fig. 8. Space consumption of DRAM and PM varying the number of 8B-8B key-value entries.

TABLE IV
MEMORY USAGE (MB) OF DIFFERENT COMPONENTS IN NBTree VARYING THE NUMBER OF KEY-VALUES (MILLION)

Number of key-values		20	40	60	80	100
Leaf	key-value (PM)	445	891	1336	1782	2227
	Node metadata (DRAM)	45	90	136	181	226
Inner Node (DRAM)		10	21	37	43	55

At the cost of performance improvement, Fig. 8 illustrates that NBTree requires more DRAM space than most previous works except uTree to store leaf metadata. However, we argue that the additional DRAM consumption from the metadata layer is tolerable. As shown in Table IV, the ratio of DRAM and PM consumption in NBTree is around 1:8, which matches the ratio of our testbed configuration (1:8). In practice, this ratio will be much smaller if the value size is large because the values only reside in PM. Meanwhile, after a crash, NBTree cannot achieve instant recovery like Bztree and FAST&FAIR, due to our hybrid DRAM-PM architecture. Rebuilding NBTree from persistent leaf nodes with 16 million key-value entries needs to take 0.32 s with a single thread. Nonetheless, the recovery time of NBTree is comparable with the state-of-the-art hybrid DRAM/PM persistent indexes, such as FPTree (0.16 s) and RNTree (0.34 s), and significantly outperforms uTree (7.25 s).

D. YCSB Evaluation

In this section, we evaluate the performance of persistent B⁺-Trees with real-world YCSB [54] workloads. We generate the skewed (zipfian key access distribution) and read-write mixture workloads based on YCSB. By default, the write operations in the workload are upsert. Upsert will insert a new key if the target key does not exist. Otherwise, it performs an update.

Overall Evaluation: Fig. 9 reports the evaluation results under YCSB workload (read:write=50:50) in a zipfian key access distribution with the default 0.99 skewness. We observe that NBTree has almost linear scalability on throughput and near-constant 99% tail latency with the increase of threads, while other trees only scale up to 14 threads. In 56 threads, NBTree achieves 6.0 \times higher throughput and 32 \times lower 99% tail latency than other trees.

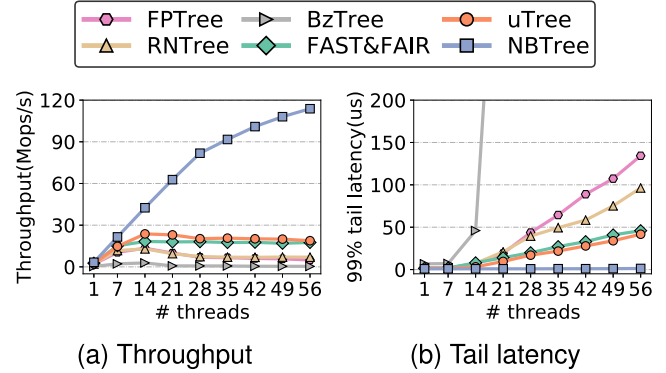


Fig. 9. Throughput and 99% tail latency under YCSB workload. (Zipfian access, skewness = 0.99, Read:write = 50:50).

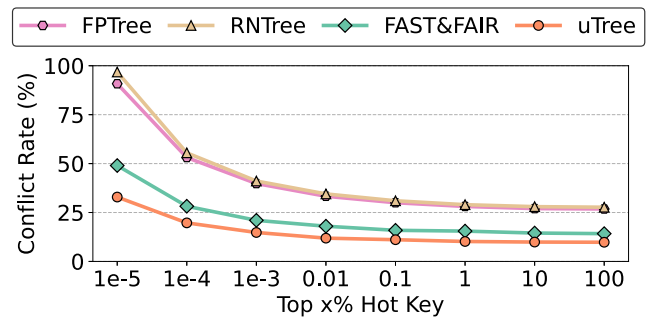


Fig. 10. Frequency of leaf-level conflicts during the updates to hot keys. (56 threads, zipfian access distribution, skewness = 0.99).

We attribute the high performance of NBTree to our efficient lock-free design. The skewed workload often introduces a lot of leaf-level conflicts. The lock-free leaf node operations in NBTree can scale well under high contentions. When operating on the leaf that is not performing SMO, NBTree employs atomic primitives to support lock-free access. When operating on the leaves in SMO, UDS operations fix the potential anomaly by our proposed techniques, such as *three-phase SMO*, *sync-on-write*, and *sync-on-read*, which only introduces at most one additional leaf node search and one CAS primitive. Concurrent insertions also apply *cooperative SMO* to achieve lock-free accesses. Besides, as the writes happen infrequently in inner nodes, our proposed *shift-aware search* and *HTM-based* updates can also scale well.

In contrast, the scalability of other persistent B⁺-Trees is limited by the less efficient concurrency control. For write operations, those trees (except BzTree) employ a write lock for each leaf nodes, which result in poor concurrency under high contentions. As shown in Fig. 10, these trees exhibit a substantial volume of leaf-level conflicts when accessing hot key-value entries in the skewed workloads. Despite all four persistent B⁺-Trees employing a node-level lock for each leaf node, uTree and FAST&FAIR outperform FPTree and RNTree, demonstrating higher performance and lower conflict rates. It is because that FPTree and RNTree necessitate re-traversing the inner node when accessing a locked leaf node, while uTree and

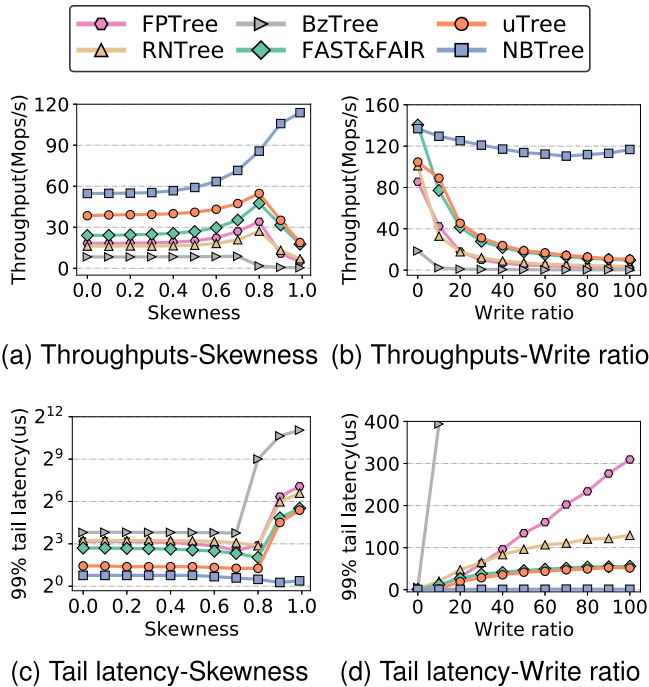


Fig. 11. Performance varying the write ratio and skewness under YCSB workload.

FAST&FAIR only need to await the release of the leaf lock. Meanwhile, uTree has a lower conflict rate than FAST&FAIR since it reduces the locking duration at the leaf nodes by moving the slow PM accesses out of the critical path. BzTree achieves lock-free by utilizing $PMwCAS$, which is an optimistic approach implemented by a series of CAS and RDCSS [41] operations. However, $PMwCAS$ is vulnerable to high contentions and brings high software overhead [55]. Therefore, BzTree has the lowest scalability despite its lock-free design. For read operations, FPTree uses the read-write locks, which means that the read operations on the leaf nodes are blocked by any write operation. RNTree uses HTM to handle read-write conflicts in the leaf nodes, which causes a lot of abortions under high contentions. uTree and FAST&FAIR perform better than RNTree and FPTree, due to their lock-free search algorithm.

Effect of Skewness: Fig. 11(a) and (c) report the evaluation results when we vary the skewness (zipfian coefficient) in YCSB workload (read:write=50:50). We notice that NBTree has better performance with the increase of skewness. The reason is two-fold: (1) Our efficient lock-free designs prevent the concurrency control from becoming the performance bottleneck. (2) Our cache-crafty designs, including *in-place update* and *log-structured insert*, have larger effects with the increase of the skewness. Those designs increase the possibility of *write combining* and *write hits* in CPU cache, which saves the PM write bandwidth.

With the increase of skewness, other trees have a slight performance improvement when the skewness is less than 0.8, which benefits from better cache utilization. When the skewness is larger than 0.8, they have a dramatic performance drop because they cannot scale well under frequent leaf-level contentions.

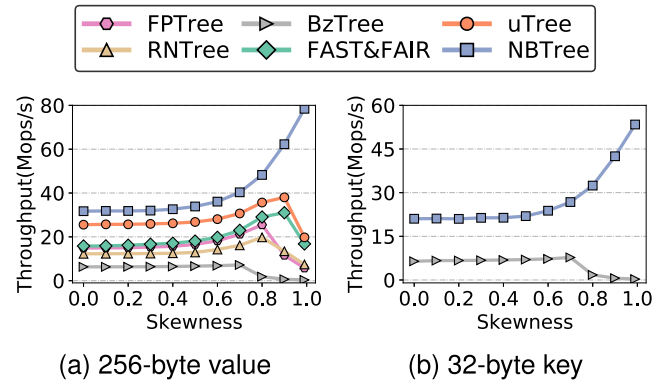


Fig. 12. YCSB performance of the indexes with large key/value. (Zipfian access, skewness = 0.99, Read:write = 50:50).

Effect of Write Ratio: Fig. 11(b) and (d) show the evaluation results when we vary the write ratio of the YCSB workload (skewness=0.99). We observe that the performance gap becomes larger between NBTree and other B^+ -Trees with the increase of write ratio. NBTree achieves $11\times$ higher throughput and $43\times$ lower 99% tail latency under the write-only workload. This is because NBTree applies efficient lock-free algorithms for both reads and writes. In contrast, previous works focus on the optimization of concurrent reads but do not support efficient concurrent writes.

Effect of Large Key/Value: Fig. 12 reports the evaluation of indexes when the key-value size is larger than 8 bytes. We observe that NBTree still achieves significantly higher throughput than other indexes, especially when the skewness is high. However, the performance gap between NBTree and other indexes becomes smaller. The reason is two-fold: (1) The PM write bandwidth is dominated by persisting large values, dwarfing the performance benefits from our optimization on NBTree. (2) NBTree employs the 8-byte pointers to indicate variable-sized keys, which incurs a lot of pointer dereferences in inner node search. In contrast, Bztree continuously stores the variable-sized keys in a single node, which avoids expensive pointer dereferences.

E. Factor Analysis for Design Components

We conduct two optimizations for NBTree, including decoupled leaf node design, and lock-free design, which contribute the most to the high scalability of NBTree. In this section, we analyze the impact of these design components in detail.

Decoupled Leaf Node Design: In NBTree, the decoupled leaf node design minimizes the PM line accesses during the base operations, allowing NBTree to outperform other persistent B^+ -Tree. In order to demonstrate the effect of the decoupled leaf node design, we implement a variant of NBTree that disables this design component by storing both metadata and data of the leaf node in a single persistent layer. Fig. 13 shows that decoupled leaf node design achieves $2.4\times$ performance improvement in the insert operations since the modifications of leaf metadata are absorbed in DRAM. Furthermore, although the UDS operations do not modify the metadata due to the optimization of NBTree,

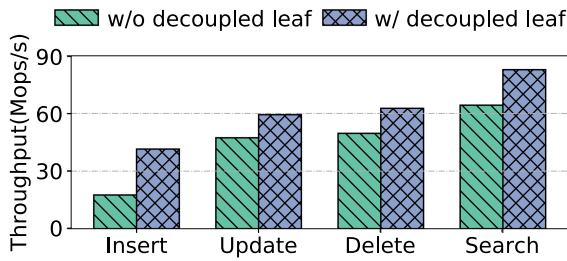


Fig. 13. Effect of decoupled leaf node design in NBTree. (56 threads, uniform access).

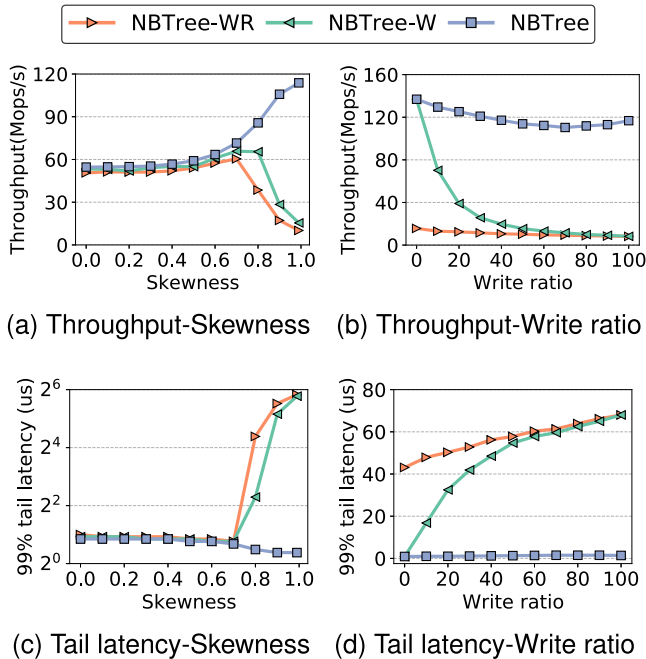


Fig. 14. Effect of lock-free design in NBTree. (56 threads, zipfian access. NBTree-W disables lock-free write schemes. NBTree-WR disables both lock-free write and read schemes).

the decoupled leaf node design still improves their throughput by up to 29% due to fewer *PM* line reads.

Lock-free Design: The lock-free design is one of the most essential components of NBTree, which substantially improves the multi-core scalability of NBTree for skewed workloads. To intuitively demonstrate the impact of our lock-free design, we implement two variant of NBTree (NBTree-W and NBTree-WR). NBTree-W disables our lock-free write schemes and applies node-grained write-locks instead, which is similar to FAST&FAIR and uTree. As shown in Fig. 14, NBTree-W can not scale well in write-intensive workloads with high skewness since the high contentions of write operations become the performance bottleneck. NBTree achieves 5.5 \times higher throughput and 48.6 \times lower tail latency than NBTree-W due to our lock-free write design. NBTree-WR further disables our lock-free read approaches and replaces them with node-grained locks for both read and write operations. We find that NBTree-W achieves up to 1.8 \times higher throughput and 47.9 \times lower tail latency than NBTree-WR in read-intensive workloads.

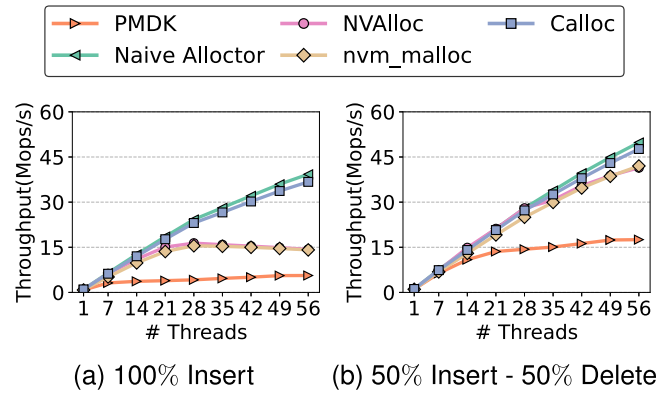


Fig. 15. Performance of NBTree under different persistent allocators. (Uniform access).

F. The Impact of Persistent Allocator

To show the superiority of Calloc, we evaluate the performance of NBTree using different persistent allocators, including Calloc, the naive allocator mentioned in Section VI-A, and several open-sourced persistent allocators (PMDK [50], NValloc [39], nvm_malloc [56]).

As shown in Fig. 15(a), Calloc achieves up to 6.5 \times higher throughput than PMDK, NValloc, nvm_alloc during insert operations. This is mainly attributed to our cache-crafty design, including allocation bitmap, reclaimed ring buffer, and recyclable log, which utilize persistent CPU cache to absorb most of PM writes generated during allocation, deallocation, and logging. In contrast, PMDK, NValloc, and nvm_alloc incur significantly higher PM writes in memory management due to persistent modifications applied to various metadata (e.g., bitmap, logging, and bookkeeping).

Fig. 15(b) illustrates that the performance gap between Calloc and other state-of-the-art persistent allocators diminishes in the workload involving both insert and delete operations. This reduction results from the decreased frequency of memory allocations/deallocations triggered by SMO, which is around 4.0 \times lower compared to workloads consisting solely of insertions.

Moreover, it is noteworthy that the naive allocator sets the performance ceiling for persistent allocators since it eliminates the persistence overhead of memory management by omitting support for crash consistency and garbage collection. Calloc achieves comparable performance to the naive allocator, which further demonstrates that the persistence overhead of Calloc is minimized.

Furthermore, we also expand the experiments of NBTree to use both NUMA nodes of our server by adopting per-NUMA persistent allocators. As shown in Fig. 16, NBTree can achieve 1.7 \times speedup compared with using a single persistent allocator since per-NUMA allocators mitigate the impact of NUMA effects by allocating and initializing tree nodes within local NUMA nodes. However, the cross-NUMA accesses cannot be thoroughly eliminated since the subsequent index operations may still access tree nodes located in the remote NUMA node, which incurs higher latency and consumes more bandwidth.

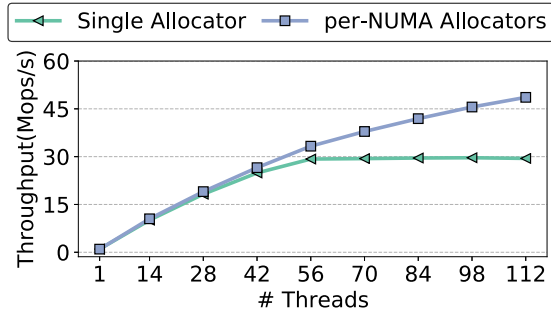


Fig. 16. Insert performance of NBTree when using two NUMA nodes. (Uniform access).

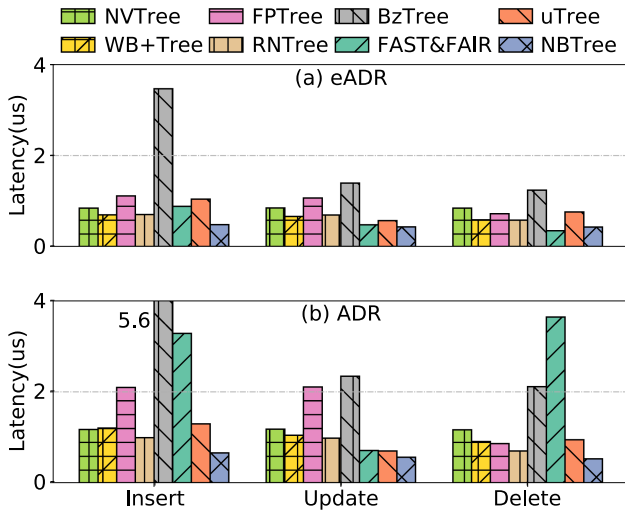


Fig. 17. PM overhead of the leaf nodes in eADR/ADR mode. (Single thread, uniform access).

G. The Impact of eADR

In this section, we compare the performance of persistent indexes between two persistence modes (ADR and eADR) to show the impact of eADR.

Fig. 17 reports the PM overhead of the leaf nodes in two persistence modes. First, we find that the PM overhead of all trees is reduced in eADR mode, compared with ADR mode. The primary reason is that the latency on the critical path caused by flush instructions is removed. For example, FAST&FAIR has a significant performance improvement because eADR minimizes the large overhead of data shifts to keep arrays sorted. The performance of BzTree also improves a lot because a large number of flush instructions needed by PMwCAS are removed. However, PMwCAS is still costly due to excessive PM accesses, resulting in the poor performance of BzTree. Second, we observe that NBTree has the lowest PM overhead in ADR mode. This is attributed to our PM-friendly designs, which cost only one flush instruction in each write operation.

Fig. 18 shows the performance of two persistence modes under YCSB workload (56 threads, read:write=50:50, skewness=0.99). We have the following four observations. First,

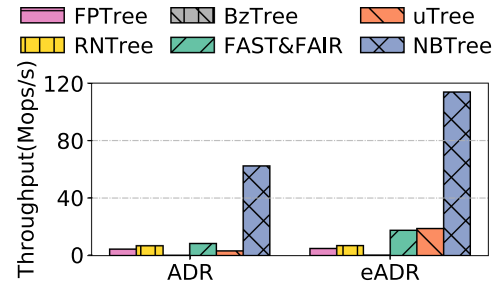
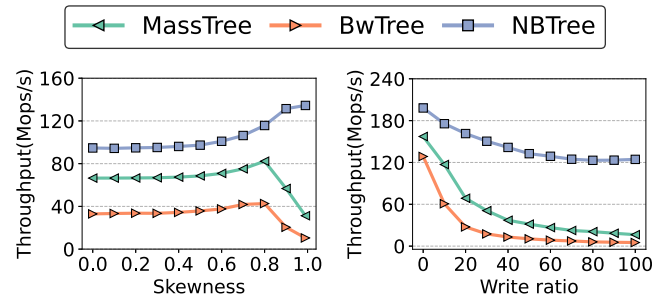


Fig. 18. Performance in ADR/eADR mode under YCSB workloads. (56 threads, Zipfian access, skewness = 0.99, Read:write = 50:50).



(a) Throughput-Skewness (b) Throughput-Write ratio

Fig. 19. YCSB performance on DRAM. (56 threads).

NBTree achieves the most performance improvement with the eADR support. This is because the cache-crafty designs (e.g., *in-place update*, and *log-structured insert*) of NBTree take effect in eADR mode. Second, NBTree also performs the best among all trees in ADR mode due to the lock-free design. However, the *dirty read* anomaly is likely to happen in ADR mode because the CPU cache is volatile. Third, uTree and FAST&FAIR also speed up significantly because of the better cache utilization in eADR mode. Fourth, RNTree, FPTree, and BzTree do not benefit from eADR because their concurrency control is vulnerable to high contentions.

H. Performance on DRAM

In order to demonstrate the generality of the NBTree design, we compare the YCSB performance of NBTree with Masstree [57] (a state-of-the-art volatile B⁺-Tree) and BwTree [23] (a lock-free volatile B⁺-Tree) on DRAM. As shown in Fig. 19, NBTree outperforms Masstree and BwTree by up to 7.6× and 24.2×, respectively, in skewed and write-intensive workloads. These evaluation results affirm the efficacy of our proposed lock-free concurrency protocol, applicable not only for PM but also for DRAM. The performance of Masstree lags behind NBTree due to the use of node-grained write locks in Masstree. BwTree achieves lock-free concurrency through a delta update policy, which atomically links an updated record to the tree node using a pointer. However, in skewed workloads, this delta update policy generates a long chain of delta records for frequently updated pages, incurring a substantial overhead on page consolidations and page traversal. Moreover, the delta

update fails to leverage the CPU cache to absorb frequent updates of hot key-value entries. In contrast, NBTree supports lock-free in-place update, which can not only avoid the chaining overhead but also fully utilize the CPU cache to enhance write performance.

I. End-to-End Evaluation

Redis [58] is a popular in-memory key-value store using a hash table as its index. We use the multi-threaded version of the Redis [59] and replace its internal index with our evaluated trees. We run 28 threads on the Redis server in our evaluation. NBTree achieves the throughput of 1719.4 Kops/s, which is 1.13-1.53× higher than other state-of-the-art persistent B⁺-Trees under the YCSB-A workload [54]. The evaluation results confirm our previous experiments. Note that the performance gap among indexes becomes smaller due to the high software overhead of Redis.

VII. RELATED WORK

A. Indexes Optimized for PM

In ADR-based PM systems, the slow write is the performance bottleneck of persistent indexes because flush instructions introduce high latency on the critical path. Therefore, previous works have proposed various ways to optimize the write performance of persistent indexes. Most persistent B⁺-Trees, such as WB⁺ Tree [27], NVTree [26], FPTree [4], RNTree [28], and LB⁺ Tree [7], implement the unsorted leaf nodes. The reason is that inserting or deleting an element of the sorted leaf node produces lots of PM writes to shift array elements. FPTree first proposes the selective persistence technique to place inner nodes in DRAM, which speeds up the inner node operations. The volatile inner nodes can be reconstructed from persistent leaf nodes after a crash. RNTree and ROART [8] further remove the flush instructions when modifying reconstructable metadata in the leaf node to reduce the critical path latency. uTree places the sorted leaf nodes in DRAM and adds a persistent shadow list-based layer to ensure crash consistency. In this way, uTree offloads the expensive structural refinement operations (SRO) to DRAM. Persistent hash indexes, such as level hashing [12], path hashing [60], and CCEH [32], also make lots of efforts to write-efficient designs.

B. Concurrency Control for Persistent Indexes

Previous works propose various concurrency control strategies for persistent indexes to leverage the benefits of multi-core processors. For persistent B⁺-Trees, FPTree proposes the selective concurrency technique, which handles the concurrency of inner nodes by HTM and serializes the accesses of leaf nodes by the node-grained locks. It improves the scalability in the situation with infrequent contentions but performs poorly in skewed workloads. Based on FPTree, RNTree excludes some slow persistent instructions out of the critical section to achieve more concurrency in the leaf nodes. FAST&FAIR designs a lock-free search algorithm inspired by B-link tree [43], which tolerates the transient inconsistent states

caused by write transactions. It improves search performance but tends to cause consistency problem [9]. uTree supports lock-free concurrency control for the list layer but still uses the coarse-grained locks in the leaf nodes. BzTree [15] develops the first lock-free persistent B⁺-Tree with PM_wCAS [40], which guarantees both the atomicity and persistence of multi-word writes. However, PM_wCAS causes high software overhead, and it is also vulnerable to high contentions [55]. As for hash-based persistent indexes, most of them are lock-based, such as level hashing [12], CCEH [32], and CMAP [61]. P-CLHT [9] is a persistent version of CLHT [62], which supports lock-free search. Clevel hashing [14] is the concurrent version of level hashing, which uses atomic primitives to implement lock-free algorithms. However, it doesn't address the *dirty read* anomaly.

C. Persistent Allocators

The persistent allocator constitutes a vital component in persistent memory systems. Compared to volatile allocators, persistent allocators encounter a number of additional challenges, including ensuring crash consistency and preventing memory leaks. Therefore, persistent allocators [39], [50], [56], [63], [64] typically produce excessive PM accesses on various types of metadata, resulting in performance degradation in persistent memory systems. nvm_malloc [56] reduces PM reads by maintaining volatile copies for metadata headers. NVAlloc [39] proposes an interleaved mapping approach to create a metadata layout that prevents costly cacheline reflashes. Makalu [63] employs a lazy persistence strategy for auxiliary metadata, mitigating the persistence overhead during the memory management. However, none of these works can entirely eliminate the overhead associated with persisting bitmap during small allocations and maintaining persistent bookkeeping during large allocations. In terms of ensuring memory safety, a number of works, including PMDK [50], nvm_malloc [56], and NVAlloc [39], adopt a logging approach during allocations and deallocations, leading to additional PM writes. Other persistent allocators, such as Makalu [63] and DCMM [8], employ a post-crash GC strategy to eliminate memory leaks, which introduces a significant recovery overhead. In contrast, our proposed Calloc leverages the advantages of persistent CPU cache to minimize the persistence overhead during the allocation, deallocation, and logging.

VIII. CONCLUSION

Existing persistent indexes suffer from low scalability and high PM overhead. Fortunately, the new platform feature for persistent memory (PM) called eADR offers opportunities to build lock-free persistent indexes and unleash the potential of PM. In this paper, we propose a lock-free PM-friendly B⁺-Tree, named NBTree, which leverages the benefits of eADR. To achieve high scalability, NBTree develops lock-free concurrency control strategies. To reduce PM overhead, NBTree proposes a decoupled leaf node structure and a cache-crafty persistent allocator, which reduces PM line accesses and improves write locality. The real-world YCSB evaluation shows that NBTree achieves up to 11× higher throughput and 43× lower 99% tail latency than state-of-the-art persistent B⁺-Trees.

REFERENCES

- [1] J. Handy, "Understanding the Intel/Micron 3D XPoint memory," in *Proc. Storage Developer Conf.*, 2015, pp. 1–30.
- [2] Intel, "Deprecating the PCOMMIT instruction," 2016. [Online]. Available: <https://software.intel.com/content/www/us/en/develop/blogs/deprecate-pcommit-instruction.html>
- [3] Intel, "eADR: New opportunities for persistent memory applications," 2021. [Online]. Available: <https://software.intel.com/content/www/us/en/develop/articles/eadr-new-opportunities-for-persistent-memory-applications.html>
- [4] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner, "FPTree: A hybrid SCM-DRAM persistent and concurrent b-tree for storage class memory," in *Proc. Int. Conf. Manage. Data*, 2016, pp. 371–386.
- [5] Y. Chen, Y. Lu, K. Fang, Q. Wang, and J. Shu, "uTree: A persistent B⁺-tree with low tail latency," *Proc. VLDB Endowment*, vol. 13, no. 12, pp. 2634–2648, 2020.
- [6] S. Venkataraman et al., "Consistent and durable data structures for non-volatile byte-addressable memory," in *Proc. USENIX Conf. File Storage Technol.*, 2011, pp. 61–75.
- [7] J. Liu, S. Chen, and L. Wang, "LB+ trees: Optimizing persistent index performance on 3DXPoint memory," *Proc. VLDB Endowment*, vol. 13, no. 7, pp. 1078–1090, 2020.
- [8] S. Ma et al., "ROART: Range-query optimized persistent art," in *Proc. 19th {USENIX} Conf. File Storage Technol.*, 2021, pp. 1–16.
- [9] S. K. Lee, J. Mohan, S. Kashyap, T. Kim, and V. Chidambaram, "Recipe: Converting concurrent DRAM indexes to persistent-memory indexes," in *Proc. 27th ACM Symp. Operating Syst. Princ.*, 2019, pp. 462–477.
- [10] R. M. Krishnan et al., "TIPS: Making volatile index structures persistent with DRAM-NVMM tiering," in *Proc. USENIX Annu. Tech. Conf.*, 2021, pp. 773–787.
- [11] X. Zhou, L. Shou, K. Chen, W. Hu, and G. Chen, "DPTree: Differential indexing for persistent memory," *Proc. VLDB Endowment*, vol. 13, no. 4, pp. 421–434, 2019.
- [12] P. Zuo, Y. Hua, and J. Wu, "Write-optimized and high-performance hashing index scheme for persistent memory," in *Proc. 13th {USENIX} Symp. Operating Syst. Des. Implementation*, 2018, pp. 461–476.
- [13] Y. Chen, Y. Lu, F. Yang, Q. Wang, Y. Wang, and J. Shu, "FlatStore: An efficient log-structured key-value storage engine for persistent memory," in *Proc. 25th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2020, pp. 1077–1091.
- [14] Z. Chen, Y. Huang, B. Ding, and P. Zuo, "Lock-free concurrent level hashing for persistent memory," in *Proc. {USENIX} Annu. Tech. Conf.*, 2020, pp. 799–812.
- [15] J. Arulraj, J. Levandoski, U. F. Minhas, and P.-A. Larson, "BzTree: A high-performance latch-free range index for non-volatile memory," *Proc. VLDB Endowment*, vol. 11, no. 5, pp. 553–565, 2018.
- [16] A. Rudoff, "Persistent memory programming," *Login, Usenix Mag.*, vol. 42, no. 2, pp. 34–40, 2017.
- [17] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An empirical guide to the behavior and use of scalable persistent memory," in *Proc. 18th {USENIX} Conf. File Storage Technol.*, 2020, pp. 169–182.
- [18] S. Gugnani, A. Kashyap, and X. Lu, "Understanding the idiosyncrasies of real persistent memory," *Proc. VLDB Endowment*, vol. 14, no. 4, pp. 626–639, 2020.
- [19] Z. Wang, X. Liu, J. Yang, T. Michailidis, S. Swanson, and J. Zhao, "Characterizing and modeling non-volatile memory systems," in *Proc. 53rd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2020, pp. 496–508.
- [20] T. Karnagel et al., "Improving in-memory database index performance with intel transactional synchronization extensions," in *Proc. IEEE 20th Int. Symp. High Perform. Comput. Archit.*, 2014, pp. 476–487.
- [21] A. Rudoff, "Persistent memory programming without all that cache flushing," in *Proc. Storage Developer Conf.*, 2020, pp. 1–38.
- [22] A. Alexandrescu, "Generic< programming>: Lock-free data structures," in *C++ Users J.*, 2004, pp. 1–7.
- [23] J. J. Levandoski, D. B. Lomet, and S. Sengupta, "The BW-tree: A B-tree for new hardware platforms," in *Proc. IEEE 29th Int. Conf. Data Eng.*, 2013, pp. 302–313.
- [24] T. L. Harris, "A pragmatic implementation of non-blocking linked-lists," in *Proc. Int. Symp. Distrib. Comput.*, Springer, 2001, pp. 300–314.
- [25] M. M. Michael, "High performance dynamic lock-free hash tables and list-based sets," in *Proc. 14th Annu. ACM Symp. Parallel Algorithms Archit.*, 2002, pp. 73–82.
- [26] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He, "NV-Tree: Reducing consistency cost for NVM-based single level systems," in *Proc. 13th {USENIX} Conf. File Storage Technol.*, 2015, pp. 167–181.
- [27] S. Chen and Q. Jin, "Persistent B⁺-trees in non-volatile main memory," *Proc. VLDB Endowment*, vol. 8, no. 7, pp. 786–797, 2015.
- [28] M. Liu, J. Xing, K. Chen, and Y. Wu, "Building scalable nvm-based B⁺ tree with HTM," in *Proc. 48th Int. Conf. Parallel Process.*, 2019, pp. 1–10.
- [29] D. Hwang, W.-H. Kim, Y. Won, and B. Nam, "Endurable transient inconsistency in byte-addressable persistent B⁺-tree," in *Proc. 16th {USENIX} Conf. File Storage Technol.*, 2018, pp. 187–200.
- [30] F. Xia, D. Jiang, J. Xiong, and N. Sun, "HiKV: A hybrid index key-value store for DRAM-NVM memory systems," in *Proc. {USENIX} Annu. Tech. Conf.*, 2017, pp. 349–362.
- [31] S. K. Lee, K. H. Lim, H. Song, B. Nam, and S. H. Noh, "{WORT}: Write optimal radix tree for persistent memory storage systems," in *Proc. 15th {USENIX} Conf. File Storage Technol.*, 2017, pp. 257–270.
- [32] M. Nam, H. Cha, Y.-R. Choi, S. H. Noh, and B. Nam, "Write-optimized dynamic hashing for persistent memory," in *Proc. 17th {USENIX} Conf. File Storage Technol.*, 2019, pp. 31–44.
- [33] J. Xu and S. Swanson, "{NOVA}: A log-structured file system for hybrid volatile/non-volatile main memories," in *Proc. 14th {USENIX} Conf. File Storage Technol.*, 2016, pp. 323–338.
- [34] S. Zheng, M. Hoseinzadeh, and S. Swanson, "Ziggurat: A tiered file system for non-volatile main memories and disks," in *Proc. 17th USENIX Conf. File Storage Technol.*, 2019, pp. 207–219.
- [35] J. Condit et al., "Better I/O through byte-addressable, persistent memory," in *Proc. ACM SIGOPS 22nd Symp. Operating Syst. Princ.*, 2009, pp. 133–146.
- [36] S. R. Dulloor et al., "System software for persistent memory," in *Proc. 9th Eur. Conf. Comput. Syst.*, 2014, pp. 1–15.
- [37] J. Coburn et al., "NV-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories," *ACM SIGARCH Comput. Archit. News*, vol. 39, no. 1, pp. 105–118, 2011.
- [38] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," *ACM SIGARCH Comput. Archit. News*, vol. 39, no. 1, pp. 91–104, 2011.
- [39] Z. Dang et al., "NVALloc: Rethinking heap metadata management in persistent memory allocators," in *Proc. 27th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2022, pp. 115–127.
- [40] T. Wang, J. Levandoski, and P.-A. Larson, "Easy lock-free indexing in non-volatile memory," in *Proc. IEEE 34th Int. Conf. Data Eng.*, 2018, pp. 461–472.
- [41] T. L. Harris, K. Fraser, and I. A. Pratt, "A practical multi-word compare-and-swap operation," in *Proc. Int. Symp. Distrib. Comput.*, Springer, 2002, pp. 265–279.
- [42] W.-H. Kim, R. M. Krishnan, X. Fu, S. Kashyap, and C. Min, "PACTree: A high performance persistent range index using PAC guidelines," in *Proc. ACM SIGOPS 28th Symp. Operating Syst. Princ. CD-ROM*, 2021, pp. 424–439.
- [43] P. L. Lehman and S. B. Yao, "Efficient locking for concurrent operations on B-trees," *ACM Trans. Database Syst.*, vol. 6, no. 4, pp. 650–670, 1981.
- [44] Z. Liu, "Fujitsu targets 2019 for NRAM mass production," 2018. [Online]. Available: <https://www.tomshardware.com/news/fujitsu-nram-nantero-carbon-nanotube,37437.html>
- [45] Micron, "Non volatile dual in line memory module NVDIMM market research report," 2021. [Online]. Available: <https://dataintel.com/report/global-non-volatile-dual-in-line-memory-module-nvdimm-market/>
- [46] U. Xian, "NVDIMM products," 2022. [Online]. Available: <https://www.unisemicon.com/index.php?m=content&c=index&a=lists&catid=56>
- [47] C. Consortium, "Compute express linkTM: The breakthrough cpu-to-device interconnect," 2022. [Online]. Available: <https://www.computeexpresslink.org>
- [48] A. Benjamin, "Compute express link CXL: Advancing the next generation of data centers," 2022. [Online]. Available: <https://www.snia.org/pm-summit>
- [49] D. Hwang, "Fast&fair," 2020. [Online]. Available: https://github.com/DICL/FAST_FAIR
- [50] Intel, "Persistent memory development kit," 2021. [Online]. Available: <http://pmem.io/pmdk>
- [51] Y. Chen, "uTree," 2020. [Online]. Available: <https://github.com/thustorage/nvm-datastructure>
- [52] J. Arulraj, "BzTree," 2019. [Online]. Available: <https://github.com/sfudis/bztree>
- [53] M. Liu, "RNTree," 2019. [Online]. Available: <https://github.com/liumx10/ICPP-RNTree>

- [54] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. 1st ACM Symp. Cloud Comput.*, 2010, pp. 143–154.
- [55] L. Lersch, X. Hao, I. Oukid, T. Wang, and T. Willhalm, "Evaluating persistent memory range indexes," *Proc. VLDB Endowment*, vol. 13, no. 4, pp. 574–587, 2019.
- [56] D. Schwalb, T. Berning, M. Faust, M. Dreseler, and H. Plattner, "NVM malloc: Memory allocation for NVRAM," *Adms@ Vldb*, vol. 15, pp. 61–72, 2015.
- [57] Y. Mao, E. Kohler, and R. T. Morris, "Cache craftiness for fast multicore key-value storage," in *Proc. 7th ACM Eur. Conf. Comput. Syst.*, 2012, pp. 183–196.
- [58] Redis, "Redis," 2009. [Online]. Available: <https://redis.io>
- [59] Vipshop, "Redis," 2017. [Online]. Available: <https://github.com/vipshop/vire>
- [60] P. Zuo and Y. Hua, "A write-friendly and cache-optimized hashing scheme for non-volatile memory systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 5, pp. 985–998, May 2018.
- [61] Intel, "Key/value datastore for persistent memory," 2019. [Online]. Available: <https://pmem.io/pmemkv/index.html>
- [62] T. David, R. Guerraoui, and V. Trigonakis, "Asynchronized concurrency: The secret to scaling concurrent search data structures," *ACM SIGARCH Comput. Archit. News*, vol. 43, no. 1, pp. 631–644, 2015.
- [63] K. Bhandari, D. R. Chakrabarti, and H.-J. Boehm, "Makalu: Fast recoverable allocation of non-volatile memory," *ACM SIGPLAN Notices*, vol. 51, no. 10, pp. 677–694, 2016.
- [64] I. Oukid, D. Booss, A. Lespinasse, W. Lehner, T. Willhalm, and G. Gomes, "Memory management techniques for large-scale persistent-main-memory systems," *Proc. VLDB Endowment*, vol. 10, no. 11, pp. 1166–1177, 2017.



Bowen Zhang received the BS degree from Shanghai Jiao Tong University in 2021. He is currently working toward the PhD degree with Shanghai Jiao Tong University. His research interests include persistent memory-based storage systems, key-value stores, and distributed systems.



Shengan Zheng received the BS and PhD degrees from Shanghai Jiao Tong University, in 2014 and 2019, respectively. He is currently an assistant professor with Shanghai Jiao Tong University. His research interests include persistent memory-based storage systems, file systems, and distributed systems.



Liangxu Nie received the BS degree from Shanghai Jiao Tong University in 2021. Currently, he is working toward the MS degree with the same institution. His areas of research focus primarily on persistent memory-based storage systems and key-value stores.



Zhenlin Qi received the BS degree from Shanghai Jiao Tong University in 2021. He is currently working towards the PhD degree with Shanghai Jiao Tong University. His research interest includes file system, memory management system, and distributed system design.



Hongyi Chen received the BS degree from Shanghai Jiao Tong University in 2023. His research interests include persistent memory-based storage systems and persistent memory management.



Linpeng Huang (Senior Member, IEEE) received the MS and PhD degrees in computer science from Shanghai Jiao Tong University, in 1989 and 1992, respectively. He is currently a professor in computer science with the Department of Computer Science and Engineering, Shanghai Jiao Tong University. His research interests include distributed systems and service-oriented computing.



Hong Mei (Fellow, IEEE) received the PhD degree from Shanghai Jiao Tong University, China, in 1992. He is a professor of computer science with Shanghai Jiao Tong University, China and Peking University, China. His main research interests range over software engineering and system software. He is a fellow of the TWAS, a member of Chinese Academy of Sciences, and a foreign member of the Academia Europaea.