Buffered Persistence in B+ Trees

MINGZHE DU* and MICHAEL L. SCOTT, Department of Computer Science, University of Rochester, USA

Non-volatile Memory (NVM) offers the opportunity to build large, durable B+ trees with markedly higher performance and faster post-crash recovery than is possible with traditional disk- or flash-based persistence. Unfortunately, cache flush and fence instructions, required for crash consistency and failure atomicity on many machines, introduce substantial overhead not present in non-persistent trees, and force additional NVM reads and writes. The overhead is particularly pronounced in workloads that benefit from cache reuse due to good temporal locality or small working sets—traits commonly observed in real-world applications.

In this paper, we propose a buffered durable B+ tree (BD+Tree) that improves performance and reduces NVM traffic via *relaxed* persistence. Execution of a BD+Tree is divided into *epochs* of a few milliseconds each; if a crash occurs in epoch e, the tree recovers to its state as of the end of epoch $e-2$. (The persistence boundary can always be made current with an explicit sync operation, which quickly advances the epoch by 2.) NVM writes within an epoch are aggregated for delayed persistence, thereby increasing cache reuse and reducing traffic to NVM.

In comparison to state-of-the-art persistent B+ trees, our micro-benchmark experiments show that BD+Tree can improve throughput by up to 2.4× and reduce NVM writes by up to 90% when working sets are small or workloads exhibit strong temporal locality. On real-world workloads that benefit from cache reuse, BD+Tree realizes throughput improvements of 1.1–2.4× and up to a 99% decrease in NVM writes. Even on uniform workloads, with working sets that significantly exceed cache capacity, BD+Tree still improves throughput by 1–1.3×. The performance advantage of BD+Tree increases with larger caches, suggesting ongoing benefits as CPUs evolve toward gigabyte cache capacities.

CCS Concepts: • **Theory of computation → Data structures and algorithms for data management**; • **Hardware →** *Memory and dense storage*; Non-volatile memory.

Additional Key Words and Phrases: Durable linearizability, nonvolatile memory, periodic persistence

ACM Reference Format:
Mingzhe Du and Michael L. Scott. 2024. Buffered Persistence in B+ Trees. *Proc. ACM Manag. Data* 2, 6 (SIGMOD), Article 226 (December 2024), 24 pages. https://doi.org/10.1145/3698801

1 Introduction

As one of the most popular indexing data structures used in database systems, B+ trees have been extensively studied and optimized. As databases continue to grow, B+ trees have also been rapidly growing, consuming large amounts of expensive DRAM. Recovery in the wake of a crash typically requires reconstruction, based on a full scan of the database, which can be quite expensive. A durable index, on the other hand, if maintained in block storage, requires logging and I/O operations on every update, increasing transaction latency and disk/flash traffic.

Hardware trends suggest (the discontinuation of Intel's Optane products notwithstanding) that systems of the near future may boast large amounts of byte-addressable nonvolatile memory (NVM). Such memory is likely to provide latency and bandwidth only a small constant factor slower than that of DRAM, together with larger capacity and lower cost per byte. It is also likely to be persistent,

*Corresponding author. Authors' addresses: {mdu5, scott}@cs.rochester.edu

ACM 2836-6573/2024/12-ART226
https://doi.org/10.1145/3698801

offering the intriguing opportunity to build persistent B+ trees with both high performance in the absence of crashes and low post-crash recovery time.

To be considered correct, a concurrent data structure is generally required to be *linearizable* [14]—its operations must appear to occur in some sequential order consistent with "real time" (if operation x returns before operation y is called, x must appear to occur before y). A *persistent* data structure is typically required to be *durably* linearizable (DL)—its history, across crash events, must be linearizable when the crashes themselves are elided [17]. Equivalently, each operation must appear to take place *and to persist* at some single point in time between its call and return [10].

The challenge to achieving durable linearizability is that caches are generally volatile, and write their contents back to memory in arbitrary order: an unmodified concurrent structure that is simply placed in NVM is unlikely to be consistent in the wake of a crash.

To build a durably linearizable structure, one must generally (1) force cache lines to be written back to memory (on an Intel machine, via clflush, clflushopt, or clwb); (2) order the writes-back explicitly (e.g., via sfence or mfence); and (3) employ undo or redo logging to allow partially completed operations to be rolled backward or forward after a crash.

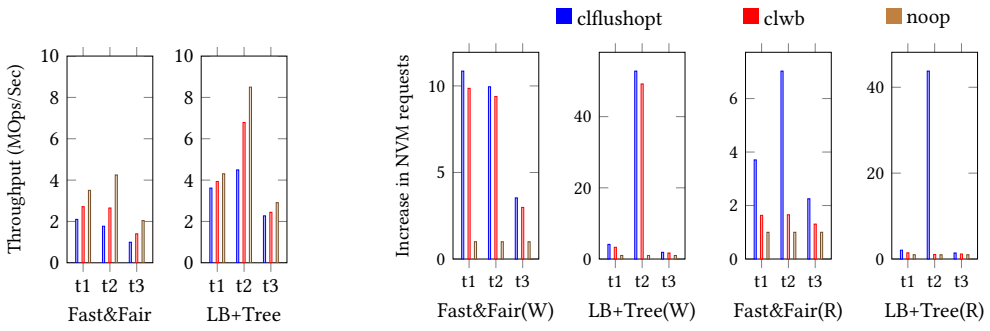
These steps have significant cost. A write-back instruction requires at least a few dozen cycles to hand-shake with the on-chip memory controller, which takes responsibility for persisting the line. If the line is not local, but dirty elsewhere in the system, a few hundred cycles may be required to find it and force it to its own local memory controller. Likewise, an ordering fence may require hundreds of cycles for cross-core or cross-socket communication.

At the same time, a write-back instruction (e.g., clflush or clflushopt) that invalidates the cache line imposes an all-the-way-to-memory miss penalty on the program's next access to the line. Repeated flushed writes to the same cache line will not only incur the latency of the store and any subsequent load: they will also contribute to wear-out in any memory technology with limited long-term endurance. In contrast, ordinary (non-flushed) stores to a given line are typically absorbed by the cache. Even on a machine with a non-flushing write-back instruction (e.g., the clwb of Intel's most recent processors), explicitly ordered NVM writes still consume memory bandwidth and contribute to wear-out.[1] Write-back of log entries (for failure atomicity) can also dilate the critical path.

Recently published durably linearizable B+ trees can be more than 10× slower than their transient counterparts [38] and use twice as much memory [6, 25]. Much work has therefore focused either on redesigning the B+ tree to use fewer persist instructions [5, 15, 25, 26, 33, 38, 43] or on moving those instructions off the critical path, to hide their impact on performance [6, 28]. Similar strategies have characterized work on general-purpose libraries that add persistence to existing volatile structures [9, 11, 19, 21, 24, 30, 37].

Unfortunately, even the best existing durably linearizable B+ trees suffer from significant persistence overhead and/or extra DRAM and NVM consumption. We use three Memcached traces from Twitter [32] (details in Table 1) to measure the persistence overhead and memory traffic of two state-of-the-art persistent trees. Each workload comprises 10 million accesses, with working sets (total amounts of accessed memory) as shown in the table. Experiments were conducted on a server with Intel Optane persistent memory and the real (non-invalidating) clwb instruction. The Fast&Fair tree [15] persists both inner and leaf nodes, and sorts the keys within each node. LB+Tree [26] is a hybrid structure that keeps leaf nodes in NVM but inner nodes in DRAM, to reduce the number of NVM writes and persist instructions. Its inner nodes are sorted, and are rebuilt after a crash; its leaf nodes are unsorted.

[1] In all but the most recent Intel processors, clwb is simply an alias for clflushopt.



(a) Persistence overhead (b) Normalized NVM traffic

Fig. 1. Behavior of persistent trees with different write-back instructions. Left: throughput (higher is better). Right: relative volume of NVM reads and writes (lower is better). Experiments were conducted with one thread.

Performance of the measured trees on the Twitter workloads is plotted in Fig. 1 for a single worker thread, using invalidating (clflushopt) and non-invalidating (clwb) write-back, and comparing to an (inconsistent) baseline with no write-back (noop). Fig. 1(a) shows that persistence instructions lead to performance losses of 20–140%. When the working set is small, invalidating write-back is particularly detrimental due to the instruction overhead and cache miss penalty. Relative to the base case (noop), Fig. 1(b) shows that clflushopt results in a 52× increase in writes and 43× increase in reads for trace t2, whose working set fits in the L3 cache. Although non-flushing clwb instructions mitigate the performance degradation and reduce read traffic, they continue to induce extra bandwidth and contribute to NVM wear-out.

Our work is founded on two key observations. First, cache reuse plays a critical role in persistent applications. As real-world workloads often exhibit small working sets (only a modest subset of total data is accessed over the course of execution) and/or good temporal locality (the same data are accessed repeatedly within a short time span) [2, 4, 40], improving cache reuse can substantially enhance performance by hiding the high latency of access to NVM and reduce NVM writes through in-cache write combining. As server CPU caches evolve towards gigabyte capacity with the advent of technologies like 3D V-Cache stacking, leveraging cache resources to their fullest potential in persistent applications promises to deliver durability with unparalleled performance.

Second, durable linearizability, in its customary strict form, is too strong a correctness criterion. A B+ tree intended to index a database on disk or flash requires persistence only on the millisecond timescale of I/O operations. Loss of the last few index updates is not a problem, so long as the post-crash state is internally consistent (and, presumably, consistent with the actual database).

			working set (MiB)	
traces	R:W ratio	Zipf α	10M ops	100M ops
t1	85:15	0.6372	97	303
t2	50:50	1.7366	4	9
t3	50:50	0	126	750

Table 1. Characteristics of Memcached traces. Zipf α is a measure of workload skew: a higher value indicates a more skewed distribution.

Previous work suggests [17, 31, 37] that performance in the absence of crashes may be significantly improved if writes-back are buffered (batched) and effected on a periodic basis. Toward that end, we present a Buffered Durable B+ Tree (BD+Tree) with relaxed persistence intended to increase performance and decrease NVM wear-out, relative to existing non-buffered trees, by amortizing persistence overhead over large operation batches and optimizing cache reuse. Specifically, BD+Tree employs an *epoch* system, managed by a background thread. The thread maintains a global clock that ticks every few milliseconds, dividing time into epochs. Each BD+Tree operation occurs within a single epoch, and all data are labeled with the epoch in which they were created. Worker threads, which update the content of the tree, keep a record of modified cache lines, which the background thread writes back to NVM at epoch boundaries. Time does not advance from epoch e to $e + 1$ until all data in epoch $e - 1$ has been persisted. Significantly, persisting data from epoch $e - 1$ does not delay operations proceeding in epoch e. After forcing the write-back of data from a given epoch, the background thread issues a fence instruction, after which all data from that epoch are guaranteed to be persistent. If a crash happens in epoch e, BD+Tree recovers to the state that existed at the end of epoch $e - 2$.

In contrast to persistent B+ trees based on periodic checkpoints, BD+Tree eliminates the need for quiescence and reduces the need for extra DRAM (to buffer NVM writes) and NVM (for logging). Experiments confirm that BD+Tree can achieve 1.1–2.4× speedup over existing persistent B+ trees. It also generates fewer NVM writes and consumes less DRAM. Summarizing contributions, we:

- present the first buffered durably linearizable (BDL) B+ tree, together with correctness arguments.
- evaluate the performance of this tree relative to state-of-the-art alternatives, demonstrating significant improvements in per-operation latency, DRAM consumption, and NVM traffic.
- explore sensitivity to algorithmic, architectural, and workload parameters, including epoch length, memory reclamation frequency, cache size, access distribution, and working set size.

2 Background and Related work

Prior research has introduced various ways to mitigate the persistence overhead and NVM writes of durable B+ trees. Common practices include selective persistence, loose ordering within leaf nodes, and dual designs. Selective persistence entails storing inner nodes in DRAM while keeping leaf nodes in NVM [5, 6, 25–27, 38, 42, 43]. With relaxed leaf node ordering, leaf nodes are either semi-sorted (some keys within the same leaf node are sorted) [33, 43] or unsorted (all keys within the same leaf are unsorted) [5, 25–27, 38, 42, 43], thereby reducing the frequency with which keys must be moved in NVM. Dual design entails maintaining a mutable copy in DRAM to absorb lookups and an immutable copy in NVM for durability; these copies are periodically synchronized for consistency [30, 43].

Fast&Fair [15] is the state-of-the-art fully persistent B+ tree. It stores all nodes in NVM and maintains them in a sorted manner. It assumes it is running on a Total Store Order (TSO) machine, which guarantees that dependent stores no not bypass one another; this allows it to eliminate instructions to maintain the ordering of writes generated by in-node shifting, provided they share a cache line. **LB+Tree** [26] is the state-of-the-art hybrid tree. It stores leaf nodes in NVM and inner nodes in DRAM. Its leaf nodes are unsorted, and use *eager movement* to enhance the likelihood that metadata and updated keys will reside in the same cache line. When the metadata line of a node has no empty slots, LB+Tree eagerly moves all co-located keys to another cache line; multiple subsequent updates can then modify only the metadata line.

In addition to custom B+ trees, the literature also offers general-purpose libraries to facilitate persistence of existing transient structures [9, 11, 13, 21, 24, 30, 35, 36]. Unfortunately, these libraries tend to have limited applicability [9, 13, 24] or to result in suboptimal performance compared

to customized persistent structures, due to high persistence overhead, excessive DRAM/NVM consumption, or both [21, 24, 30, 35, 36].

MOD [13] is designed specifically for purely functional data structures. Recipe [24] necessitates non-blocking read operations. Mirror [11] and NVTraverse [9] are intended for fully lock-free data structures, and their performance remains limited by the overhead of write-back. Pronto [30] retains data structures in DRAM and captures periodic snapshots, thereby limiting scalability beyond DRAM capacity and leading to additional memory consumption. TIPS [21] maintains a volatile hash table to record write operations. Both Pronto and TIPS require extra NVM space for logging to guarantee durable linearizability.

Buffered durable linearizability (BDL) [17] relaxes persistence requirements with a weaker consistency guarantee. A data structure is said to be buffered durably linearizable if (1) it is linearizable during crash-free execution, and (2) upon a crash, the data structure preserves a consistent prefix of the linearization order of pre-crash execution. Dalí [31] is a custom-designed BDL hash table. **Montage** [37] is a general-purpose library designed to facilitate the creation of BDL structures. It encourages a design pattern in which only essential data are kept in NVM. For a B+ tree, Montage would keep only a pile of key-value (KV) pairs in NVM; both inner nodes and the links among leaf nodes would be transient. The lack of persistence overhead for inner nodes would improve performance during crash-free operation, but would lead to slow post-crash recovery, due to the need to scan NVM, identify KV pairs, and sort them.

The **TL4X** [1] framework also executes transactions in a BDL manner. It employs four replicas: two volatile and two persistent. Speculative write transactions update the volatile *Main* replica. Irrevocable, read-only transactions access the volatile *Back* replica, which reflects a linearizable execution history. *Back* is periodically synchronized to whichever persistent replica is least up-to-date.

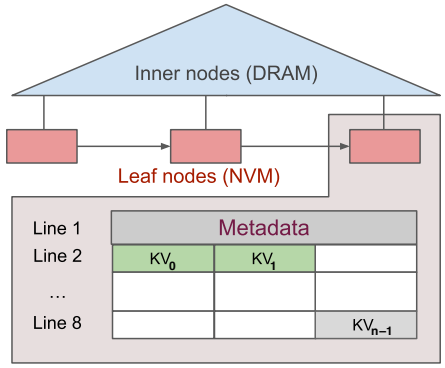
Various other (non-B+) trees can also serve as a persistent database index [20, 23, 29, 34]. **ERT** [34] is a persistent radix tree whose primary objective is to minimize traversal overhead. It employs a large node size to minimize tree height. Each node is organized as extensible hash table, enabling efficient intranode search.

3 BD+Tree design

BD+Tree achieves relaxed persistence by implementing an epoch system and a specialized leaf node design. Fig. 2 provides an overview of the BD+Tree architecture. B+ tree traversal occurs primarily in inner nodes, while the majority of persistence overhead is associated with leaf node updates [38]. Therefore, BD+Tree employs a hybrid design, similar to other persistent B+ trees, where inner nodes are stored in DRAM and leaf nodes are kept in NVM. The inner nodes retain the conventional structure of standard B+ trees, but leaf nodes are distinctive: among other things, they include metadata to indicate the epoch in which each update occurred. This information helps the background, epoch-advancing thread to manage write-back operations at epoch boundaries. In the wake of a crash, the epoch information also allows the recovery procedure to identify the data that were present at the penultimate epoch boundary, and that should therefore appear in the rebuilt tree.

3.1 Epoch system

The epoch system serves two key functions: first, it maintains a global clock that ticks every few milliseconds, segmenting execution into distinct epochs; second, it tracks modified data within each epoch and persists this data to NVM at epoch boundaries. Managed by a background thread, the epoch system ensures that the maintenance and persistence overhead are off the critical path. At any given point in normal (crash-free) execution, epochs can be categorized as




```
class Metadata {
  char fingerprints[CAP];
  /* <epoch, bitmap_e-1, bitmap_e-2> */
  EpochBitmap eb;
  uint32 bitmap_e;
  uint8 tracked_lines;
  uint32 lock;
  bool deleted;
  Leaf* sibling;
} ;
```

```
bitmap_e    1 1…0 0 // add kv_1
bitmap_e-1  1 0…0 0 // add kv_0, rm kv_n-1
bitmap_e-2  0 0…0 1 // add kv_n-1
```

Fig. 2. BD+Tree structure with 512B nodes (left), metadata structure (top right), and a bitmap example (bottom right).

- the *active* epoch e, in which new operations start. Operations in this epoch will not survive a crash that occurs before epoch $e + 2$.
- the *in-flight* epoch $e-1$, in which threads can continue their ongoing operations but are prohibited from starting new ones. Operations in this epoch will not survive a crash that occurs before $(e - 1) + 2 = e + 1$.
- *valid* epochs i, for $i \leq e - 2$. Threads are not allowed to execute or initiate new operations in these epochs. Content in valid epochs has been securely persisted.

Each tree operation, performed by a worker thread, begins and completes within a single epoch. Before progressing to the next epoch, the epoch system ensures that all in-flight operations have been completed and data has been persisted. Waiting for in-flight operations does not hinder system progress, as threads are permitted to initiate new operations in the active epoch. Upon completion of all in-flight operations, the epoch system sends dirty data to NVM and issues a fence instruction to guarantee persistence, after which it increments the epoch number. Following this advancement, the formerly active epoch becomes the in-flight epoch, and the in-flight epoch becomes a valid epoch. In the event of a crash, we recover all and only those operations that occurred in valid epochs.

3.2 Leaf node design

BD+Tree leaf nodes are unsorted: keys within a node appear in any order, but are all greater than keys in nodes to the left and smaller than keys in nodes to the right. The first cache line of a leaf node (Fig. 2) is reserved for metadata; the remaining cache lines are dedicated to key-value (KV) pairs. Our experiments use a leaf node size of 512 bytes, though this could in principle be changed. The metadata comprises several key components:

- **fingerprints** is an array of one-byte hash values for keys, enabling faster searches. A SIMD instruction can efficiently pinpoint the indices of match candidates given the hash value of a search key, thereby identifying potentially matching KV pairs.
- **EpochBitmap eb** is a composite field comprising an epoch number and two bitmaps. The epoch number indicates when the node was last updated. If its value is e, the two bitmaps bitmap_{e-1} and bitmap_{e-2} indicate which KV pairs were visible in and before epoch $e - 1$, respectively. EpochBitmap requires atomic updates, so it is implemented as a 128b integer. It could be scaled to larger sizes using hardware transactional memory (HTM).

- **bitmap$_e$** is a bit array that identifies KV pairs that are visible in epoch e, where e is the epoch number stored in EpochBitmap. A KV pair is considered inserted/deleted into the node when the associated bit is set/cleared.
- **tracked_lines** is a bitmap indicating which cache lines in the node have been modified in the epoch indicated by the value in EpochBitmap. The epoch system tracks dirty data at the granularity of cache lines rather than individual write requests, because writes to the same cache line within the same epoch can be combined and need to be persisted only once at the epoch boundary.

The global epoch number, node epoch number, and three bitmaps together determine the state of a leaf node. Consider the example shown in the bottom right of Fig. 2. Assume that the global epoch number is 5, the same as the node epoch number stored in EpochBitmap, and $\text{bitmap}_e = \langle 1, 1, \ldots, 0, 0 \rangle$, indicating the presence of KV pairs in slots 0 and 1. Similarly, $\text{bitmap}_{e-1} = \langle 1, 0, \ldots, 0, 0 \rangle$, indicating that there was only one KV pair, in slot 0, during epoch 4, and $\text{bitmap}_{e-2} = \langle 0, 0, \ldots, 0, 1 \rangle$, indicating that there was one KV pair, in the last slot, $n-1$, two epochs ago. Together, the three bitmaps imply the execution history: a KV pair was inserted into slot $n-1$ two (or more) epochs ago; in epoch 4, the KV pair in the last slot was removed, and a new KV pair was inserted into slot 0; in epoch 5, a KV pair was inserted into slot 1. Upon a crash in epoch 5, the node would be restored to its state two epochs ago, with only the KV pair in slot $n-1$ remaining. KV pairs in slots 0 and 1 would be discarded. If the node epoch number were 4 instead of 5, the last update to the node would have occurred one epoch ago. The values in bitmap_e and bitmap_{e-1} would identify visible KV pairs from the previous epoch and two epochs ago, respectively. Only the KV pair in slot 0 should be recovered after a crash. Similarly, if the epoch number were 3, bitmap_e would indicate visible KV pairs from two epochs ago. The post-crash recovery procedure would restore the KV pairs in the first two slots. The following section details the mechanism through which BD+Tree manipulates the metadata to achieve BDL.

3.3 Operations

The BD+Tree APIs includes *insertion*, *deletion*, *lookup* and *range query* operations. Internally, write operations (*insertion* and *deletion*) are made visible to the epoch system, enabling it to track the updates for persistence. In contrast, read operations (*lookup* and *range query*) do not alter the tree state and remain transparent to recovery, allowing them to execute without epoch system monitoring.

A writer signals its presence to the epoch system by reading the global epoch number; writing this, together with its thread ID, in an announcement array visible to the epoch system; and double-checking the global epoch number. If the double-check fails, indicating an epoch transition, the writer retries its announcement in the new epoch. Upon successful announcement, a write operation is confined to the epoch it recorded in the announcement array: modified data are tracked, persisted, or discarded together.

Insertion code appears in Listings 2 and 3. Consider the illustration in Fig. 3, where the bottom array stores leaf entries and the top three arrays, from top to bottom, are bitmap_e, bitmap_{e-1}, and bitmap_{e-2}, with each bit aligned with the corresponding slot in the bottom array. In this simple example, each entry occupies one cache line. (In the experiments of Sec. 4, keys and values are 8 bytes each, and a cache line holds 4 entries.) Assume, initially, that the leaf node is empty, the node epoch number is 4, and all bitmaps are 0 (light blue in the figure). Upon successful registration and acquisition of an epoch number 4 (Listing 2 line 3), the writer compares the leaf node epoch number with the operation epoch number *op_epoch*. If the node epoch number exceeds *op_epoch*, implying that the epoch system has advanced the global epoch and the node has been modified

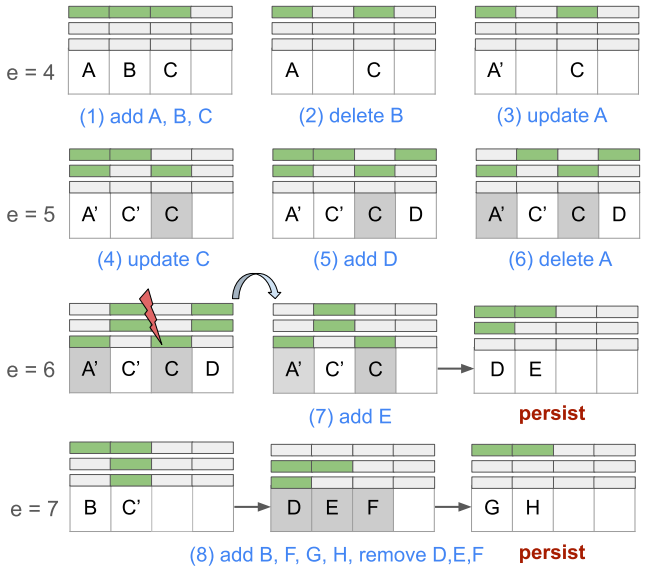


Fig. 3. BD+Tree insert and remove operations.

```
1 uint64 esys_regist(int tid, args...) {
2 retry:
3   uint64 op_e = esys.get_epoch();
4   esys.regist(op_e, tid);
5   uint64 cur_e = esys.get_epoch();
6   if (op_e != cur_e) goto retry;
7   return op_e;
8 }
```

Listing 1. Write operation registration

by another thread in a later epoch, the writer must retry the registration process (Listing 2 lines 12–17). In our example, both node epoch and *op_epoch* are 4, as illustrated in Fig. 3 (1), and the writer can directly insert A, B, and C into empty slots. It then flips the first three bits in bitmap_e and tracked_lines (making them green in the figure), and records the addresses of the updated cache lines in a shared buffer accessible to the epoch system, thereby instructing it to persist those lines (Listing 3 lines 2–15). Entries within the same epoch are updated in place (Fig. 3 (3); Listing 2 lines 35–39). As suggested by tracked_lines, the epoch system does not track those cache lines again. Writes to the same cache line are combined in the cache, allowing the epoch system to persist them with one persist instruction and to generate one write request to NVM.

In epoch 5, the node epoch number and bitmaps must be updated prior to a write operation: the in-flight epoch becomes a valid epoch, and the active epoch becomes the in-flight epoch. Tracked_lines is reset to 0, indicating that all entry cache lines are clean in the current epoch (Listing 2 lines 19–30). As shown in Fig. 3 (4), the first and the third bits in bitmap_{e-1} are set, indicating that A′ and C were inserted in the previous epoch. The modification of an entry from earlier epochs is conducted out-of-place to preserve the old version for recovery: C is preserved in its original position, and the new version C′ is stored in the second slot. To ensure the visibility of only the newest version, the bit in bitmap_e that corresponds to C is cleared (Listing 3 lines 7–9).

Note that while bitmap_e denotes the visibility of entries throughout execution, the occupancy of a slot is determined by the three bitmaps collectively. For instance, in Fig. 3 (7), only C′ and D are visible in the node, while A′ and C have been marked as deleted. Nevertheless, the two slots are still occupied by A′ and C: should a crash occur during epoch 6, the node is expected to revert to its state in epoch 4, when both A′ and C were present (Fig. 3 (1)).

When the node reaches full capacity, as illustrated in Fig. 3 (7), inserting an entry triggers a node split. The larger entries, including both visible entries and logically deleted entries, are relocated to the new sibling node (Listing 3 lines 17–38). Importantly, this new node must be persisted immediately, before removing migrated entries from the original node. The bitmaps in the two nodes are updated to complete this procedure (Listing 3 lines 39–49).

In epoch 7, as the deletion of A′ and C has been confirmed by the epoch system, their slots can be repurposed by other operations.

Deletion code appears in Listing 4. An entry is considered deleted when the corresponding bit in bitmap_e is cleared (Listing 4 lines 30–32). Continuing our example, as shown in Fig. 3 (2), removing B, an entry updated in the same epoch, is straightforward: by clearing the corresponding bit, the slot occupied by B is made ready for reuse immediately. By contrast, entries from previous epochs can only be logically deleted: the actual data is be preserved for recovery purposes. As an example, in Fig. 3 (6), A′ can only be logically deleted; its slot becomes available for reuse by B after two epoch advances, as shown in Fig. 3 (8), at which point all updates from two epochs prior will have been persisted by the epoch system.

A leaf node is deemed empty when there are no visible entries (bitmap_e is zero); an empty leaf node is considered reclaimable when it contains neither visible nor logically deleted entries—i.e., when all bitmaps are zero. In Fig. 3 (8), even though the second node appears empty from an active thread's perspective, it cannot be reclaimed, as D and E must be retained for possible post-crash recovery. An empty node is marked as deleted and its reference is removed from its parent node, but it is retained in the leaf linked list (Listing 4 lines 36–39). Memory reclamation is delegated to a second background thread, which periodically scans the leaf linked list for reclaimable nodes; further details appear in Section 3.6. Like many other B+ trees [15, 18, 26], BD+Tree never merges nodes.

Lookup operations execute independently of the epoch system's monitoring. A reader utilizes bitmap_e to identify valid KV pairs, ignoring those that reside in the node but are marked as deleted. Code appears in Listing 5.

Range-query operations commence by finding the first leaf and then traversing the leaf linked list to accumulate all KV pairs falling within the specific range. During this traversal, nodes that are empty but yet not reclaimable are skipped.

3.4 Correctness

Upon a system crash in epoch e, BD+Tree recovers to the state at the end of epoch $e - 2$. The recovered history is buffered durably linearizable as a result of three properties that hold throughout execution:

(1) operations are linearizable;
(2) operations in epoch $e - 1$ linearize before operations in epoch e;
(3) operations persist in an order consistent with the linearization order.

BD+Tree maintains property 1 by means of per-leaf locks. Each operation appears to occur instantaneously when the associated bit in bitmap_e is set or cleared. During crash-free execution, operations appear to occur in a total order consistent with any ordering that threads are able to observe. To maintain properties 2 and 3, BD+Tree avoids two incorrect scenarios:

```
void insert(k_type k, v_type v, int tid) {
retry_register:
  uint64 op_epoch = esys_regist(tid);
  htm_begin();
  LeafNode* leaf = tree_traversal(k);
  leaf->lock();
  htm_end();
  if (op_epoch < global_epoch) {
    leaf->unlock();
    goto retry_register;
  }
  if (leaf->get_epoch() != op_epoch) {
    if (leaf->get_epoch() > op_epoch) {
      /* do not directly update the node modified in a later epoch */
      leaf->unlock();
      goto retry_register;
    }
    /* initially update the node in a new epoch */
    leaf->tracked_lines = 0;
    EpochBitmap new_eb;
    new_eb.epoch = op_epoch;
    if (leaf->get_epoch() == op_epoch - 1) {
      new_eb.bitmap_{e-2} = leaf->eb.bitmap_{e-1};
      new_eb.bitmap_{e-1} = leaf->bitmap_e;
    } else if (epoch < op_epoch - 1) {
      new_eb.bitmap_{e-2} = leaf->bitmap_e;
      new_eb.bitmap_{e-1} = leaf->bitmap_e;
    }
    CAS(leaf->get_eb(), new_eb);
    add_to_persist(leaf, 0);  // track the first (metadata) cacheline */
  }
  bool oop_upd = false;
  if ((int pos = leaf->find(k)) != -1) {
    if (matched key was from the current epoch) {  // update in place
      leaf->entries[pos].val = v;
      leaf->unlock();
      return;
    } else {
      oop_upd = true;  // update existing KV in a new epoch out-of-place
    }
  }
  leaf->leaf_insert(k, v, oop_upd, pos, op_epoch);
  leaf->unlock();
  return;
}
```

Listing 2. Insertion

- **Problem 1: $store_{e-1}$ overwrites $store_e$**
  When a write operation in the in-flight epoch $e-1$ (call it $store_{e-1}$) targets the same node as $store_e$ in the active epoch e, property 2 would be violated if $store_e$ completed prior to $store_e - 1$, and $store_{e-1}$ overwrote $store_e$.
- **Problem 2: $store_e$ becomes visible before $store_{e-1}$**
  This might happen even if $store_{e-1}$ and $store_e$ modify *different* nodes and $store_e$ completes before $store_{e-1}$. A range query, for example, might see $store_e$ but not $store_{e-1}$.

Problem 1 is avoided by aborting a write operation when it attempts to modify a node that has been updated in a later epoch. When the operation restarts, it will no longer belong to the prior

```
void leaf_insert(k_type k, v_type v, bool oop_upd, int old_p, uint64 op_e) {
  if (!is_full()) {
    int new_pos = find_empty_slot();
    leaf->entries[new_pos] = (k, v);
    leaf->fingerprints[new_pos] = hash(k);
    set_bit(bitmap_e, new_pos); // mark as inserted
    if (oop_upd) {
      unset_bit(bitmap_e, old_p); // delete old KV
    }
    int line_num = get_line_num(new_pos);
    if (!tracked_lines.include(line_num)) {  // only track untracked lines
      add_to_persist(leaf, line_num);
      set_bit(tracked_lines, line_num);
    }
    return;
  } else {  // leaf is full, split it
    LeafNode* new_leaf = pnew();
    Indexes to_move = get_indexes_of_half_largest_KVs();
    int moved_bitset = 0, new_bitmap = 0, to = 0;
    for (int i : to_move) {
      new_leaf.entries[to] = entries[i];
      new_leaf.fingerprints[to] = fingerprints[i];
      if (oop_upd && entries[old_p].key == k && (bitmap_e & (1 << old_p))) {
        old_p = i;  // old KV is moved to new split node;
                    // record its position in new node for deletion
      }
      moved_bitset |= (1 << i);
      new_bitmap |= (1 << to);
      to++;
    }
    new_leaf->bitmap_e = bitmap_e & new_bitmap;
    new_leaf->eb.bitmap_{e-1} = eb.bitmap_{e-1} & new_bitmap;
    new_leaf->eb.bitmap_{e-2} = eb.bitmap_{e-2} & new_bitmap;
    new_leaf->eb.epoch = op_e;
    new_leaf->sibling = sibling;
    if (k >= split_key) {
      new_leaf->leaf_insert(k, v, oop_upd, old_p, op_e);
    }
    persist(new_leaf);  // persist split node immediately
    sibling = new_leaf;
    bitmap_e &= ~(moved_bitset); // clear moved KVs
    EpochBitmap new_eb;
    new_eb.epoch = op_e;
    new_eb.bitmap_{e-1} = bitmap_{e-1} & ~(moved_bitset);
    new_eb.bitmap_{e-2} = bitmap_{e-2} & ~(moved_bitset);
    CAS(eb, new_eb);
    if (k < split_key) {
      leaf_insert(k, v, oop_upd, old_p, op_e);
    }
    /* update the inner tree */
  }
}
```

Listing 3. Leaf insertion

epoch. **Problem 2** is avoided by coordinating read and write operations: once $store_e$ secures a lock on a node, it double-checks the global epoch number and aborts if it has changed. If a reader has observed some $store_{e+1}$, $store_e$ will either abort (due to epoch check failure) or will already hold the

```
void delete(k_type k, int tid) {
retry_register:
  uint64 op_epoch = esys_regist(tid);
  htm_begin();
  LeafNode* leaf = tree_traversal(k);
  leaf->lock();
  htm_end();
  if (op_epoch < global_epoch) {
    leaf->unlock();
    goto retry_register;
  }
  if (leaf->get_epoch() != op_epoch) {
    if (leaf->get_epoch() > op_epoch) {
      leaf->unlock();
      goto retry_register;
    }
    leaf->tracked_lines = 0;
    EpochBitmap new_eb;
    new_eb.epoch = op_epoch;
    if (leaf->get_epoch() == op_epoch - 1) {
      new_eb.bitmap_{e-2} = leaf->eb.bitmap_{e-1};
      new_eb.bitmap_{e-1} = leaf->bitmap_e;
    } else if (epoch < op_epoch - 1) {
      new_eb.bitmap_{e-2} = leaf->bitmap_e;
      new_eb.bitmap_{e-1} = leaf->bitmap_e;
    }
    CAS(leaf->get_eb(), new_eb);
    add_to_persist(leaf, 0);
  }
  if ((int pos = leaf->find(k)) != -1) {
    unset_bit(bitmap_e, pos); // mark the KV as deleted
  }
  if (leaf->bitmap_e) { // leaf is non-empty
    leaf->unlock();
    return;
  } else {  // leaf is visibly empty, mark it as deleted
    leaf->deleted = true;
    /* keep node in leaf layer; remove from inner tree */
  }
}
```

Listing 4. Deletion

node lock. In the latter case, the reader will wait for the lock to be released, and is guaranteed to observe $store_e$.

Thus, property 2 is satisfied, as all stores in $e-1$ occur and are observed prior to stores in e. The epoch system, for its part, guarantees that updates in epoch $e-1$ are persisted before those in epoch e and that updates from the same epoch are persisted or discarded together, thereby maintaining property 3.

3.5 Concurrency control

We leverage hardware transactional memory (HTM), mutex locks, and sequence locks (seqlocks) [22] for concurrency control. HTM protects the tree traversal and lock acquisition, with fallback to a global reader-writer lock in the (rare) event of HTM aborts. Mutex locks and seqlocks are employed for inner nodes and leaf nodes, respectively.

```
v_type lookup(k_type k) {
retry_lookup:
  htm_begin();
  LeafNode* leaf = tree_traversal(k);
  uint32 lock_val = leaf->read_seqlock();
  htm_end();
  while (leaf->locked()) {  // wait until writer completes its op
  }
  if ((int pos = leaf->find(k)) != -1) {
    v_type val = leaf->entries[pos].val;
    if (leaf->read_seqlock() == lock_val) {
      return val;
    } else {
      goto retry_lookup;
    }
  }
  return NULL;
}
```

Listing 5. Lookup

A writer traverses the tree inside a transaction, acquiring locks on inner and leaf nodes during this process. Similarly, a reader traverses the tree under the protection of HTM but does not acquire locks. Instead, it checks the seqlock value in the leaf node before and after reading. The read is retried if the seqlock value has changed, indicating that a writer has modified the leaf node during the read.

3.6 Memory reclamation

As inner nodes are allocated in DRAM and reclaimed much as in other B+ trees, our discussion focuses on memory reclamation for leaf nodes in NVM. Memory reclamation includes removing redundant KV pairs and reclaiming empty leaf nodes.

A key may be paired with multiple values ("duplicates") if it undergoes updates across multiple epochs. For duplicates that are older than two epochs, only the most recent needs to be retained for post-crash recovery. Suppose, for example, that A was inserted in epoch 5 and updated in epochs 6, 7, 8, and 9. There would be five instances of A: A_5, A_6, A_7, A_8 and A_9. Instances A_5 and A_6 are redundant and can be removed, as A will be restored to A_7 after a crash in epoch 9. Our implementation ensures that redundant KV duplicates are promptly removed and restricts each KV pair to a maximum of three instances. Given that all KV instances are stored in the same leaf node, when duplicating a KV pair we also logically remove the old KV pair by clearing the corresponding bit in bitmap_e. If the same KV pair is updated 2 epochs later, the space occupied by the old KV pair can be repurposed for other operations because the removal is known to be valid.

As discussed in Sec. 3.3, despite appearing empty, a node cannot be reclaimed immediately if it contains KV pairs from the previous two epochs. To identify reclaimable nodes, BD+Tree employs a background thread that periodically scans the leaf layer. If the global epoch is e, a node is considered reclaimable if it is marked as deleted and (1) the node epoch is e and all bitmaps are zero, (2) the node epoch is $e-1$ and both bitmap_e and bitmap_{e-1} are zero, or (3) the node epoch is smaller than $e-1$ and bitmap_e is zero.

4 Evaluation

We evaluate BD+Tree in comparison to state-of-the-art alternatives, focusing on operation throughput, NVM traffic, and post-crash recovery time.

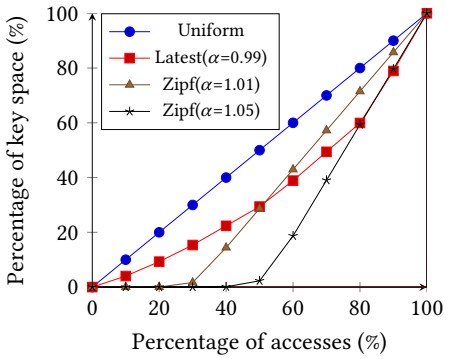


Fig. 4. Key distributions of synthetic workloads.

System configuration: all experiments were conducted on a Linux 5.15.0 (Ubuntu) server with two Intel Xeon Gold 6330 CPUs, with 28 physical cores (56 hyperthreads) in each socket. Each socket has a total of 1.3 MiB L1 D-cache, 0.9 MiB L1 I-cache, 35 MiB L2 cache, and 42 MiB L3 cache. L1 and L2 cache banks are per-core private; the L3 cache is shared by all cores on a given socket. Each socket has 8 channels of 16 GiB DRAM and 4 channels of 128 GiB second-generation Intel Optane persistent memory, for a two-socket total of 256 GiB of DRAM and 1 TiB of NVM. NVM is configured in direct access (DAX) mode using the ext4 file system.

Tested data structures: We benchmark BD+Tree against three persistent B+ trees: Fast&Fair (fully persisted, DL), LB+Tree (hybrid, DL), and Montage-B+Tree (hybrid, BDL). The Fast&Fair tree and LB+Tree are described in Section 2; we use the original source code without modifications. The Montage-B+Tree is constructed using the Montage framework [37]. In contrast to the other three trees, which store KV pairs in persistent leaf nodes, Montage places the entire tree in DRAM. KV pairs are allocated and persisted in NVM, with pointers to those pairs appearing in volatile leaf nodes. The Montage-B+Tree employs the same concurrency control as BD+Tree (Sec. 3.5). Fast&Fair, BD+Tree, and Montage-B+Tree utilize a 512-byte node size (with testing on BD+Tree under varying node sizes revealing negligible performance variations), while LB+Tree employs a 256-byte node size as it demonstrates the optimal performance across diverse workloads. Key and value sizes in our evaluation are set to 8 bytes each. This choice accommodates the design of LB+Tree, which achieves significant benefits by collocating multiple KV pairs and leaf node metadata in the same cache line when possible. All trees use the Ralloc allocator for NVM [3].

4.1 Synthetic workload evaluation

We employ synthetic workloads to examine the performance, scalability, and space efficiency of BD+Tree, adjusting system parameters (epoch length, reclamation frequency), workload key distribution, and read-write ratio to determine their impact on the behavior of buffered persistence. Operations sample our key space using one of three distributions: Uniform, Zipfian, or “Latest.” These distributions are representative of typical key-value database scenarios [2, 4, 39]. The Zipfian generator scatters hot keys throughout the key space. In contrast, the "Latest" generator [7] is based on a skewed Zipfian distribution favoring the items accessed most recently in the past. These distributions are depicted graphically in Fig.4. The concentration of hot keys in Zipfian workloads is regulated by the constant α, where a higher value of α indicates a more skewed distribution. For instance, with α=1.01, 60% of accesses are focused on 40% of the key space, whereas with α=1.05, the same proportion of accesses is concentrated on 10% of the key space.

Our experiments employ five synthetic workloads, shown in Table 2. U, L and Z indicate Uniform, Latest and Zipfian workloads, respectively. U1 is smaller than U2; Z1 is less skewed than Z2. Numbers of worker threads are indicated for each experiment. Each thread runs operations in a tight loop for 10 seconds, concurrent with all other threads. Each test is repeated three times, after which we report the average numbers of operations per second or NVM writes per second.

4.1.1 Comparative throughput of trees. Our initial experiment measures the throughput of our four different trees on (a) a single thread, (b) 24 threads within a single socket, and (b) 48 threads across two sockets. The epoch length of BD+Tree is set to 50 ms for 1 and 24 threads, and 500 ms for 48 threads. As recommended by the authors [37], Montage-B+Tree's epoch length is fixed at 50 ms. (Changing the epoch length in Montage-B+Tree on 48 threads yields negligible performance variation.) Each tree is initially populated with half the possible keys, selected uniformly at random (500,000 keys for workload U1 and 5 million keys for workloads U2, L1, Z1, and Z2). As tree prefilling is performed using each tree's own insertion operations, BD+Tree may start with a lower cache miss rate due to the minimal use of cache-invalidating write-back instructions. Operations may be either reads (lookups) or writes (insertions or deletions, each with probability 0.5).

Results appear in Fig. 5. BD+Tree exhibits superior performance in workloads with small working sets (U1), or good temporal locality (L1, Z1, Z2). The performance gains are particularly significant when the workloads are smaller (relative to the cache size) and more skewed. BD+Tree performs particularly well for workload U1 because its working set fits within the L3 cache, allowing loads and stores to NVM within the same epoch to be served directly by the cache. It performs better for Z2 than Z1, as Z2 demonstrates stronger temporal locality.

In the most challenging scenario—uniform access distribution and a working set that exceeds cache capacity (U2)—BD+Tree's performance edge is reduced or even eliminated for two primary reasons. First, random accesses to leaf nodes in a large tree induce a large number of cache misses, leading to cache replacement that is driven not by persistence instructions but by simple capacity limits. Thus, BDL yields marginal performance improvement. Second, BDL incurs extra overhead to keep track of dirty data and to maintain KV duplicates (for rollback to a previous epoch boundary on recovery). The background persisting thread may also interfere with worker threads by contending for memory resources. The overhead and interference are closely related to epoch length (Section 4.1.2). The varying performance behavior of BD+Tree in Fig. 5 indicates that it tends to deliver better performance with larger cache sizes, a finding confirmed by experiments presented in Section 4.2.1.

We attribute the disappointing performance of Fast&Fair to its use of a fully persistent tree (both internal and external nodes) and to its decision to sort the keys in every node: sorting often requires that the entire node (comprising several cache lines) be flushed to NVM, rather than just the line(s) containing modified KV pair(s). Montage-B+Tree suffers from higher access latency and memory management overhead due to extra indirection—i.e., to its use of volatile leaves containing pointers

workload	distribution	key space size
U1	uniform	1 million
U2	uniform	10 million
L1	latest (α=0.99)	10 million
Z1	Zipfian (α=1.01)	10 million
Z2	Zipfian (α=1.05)	10 million

Table 2. Synthetic workloads

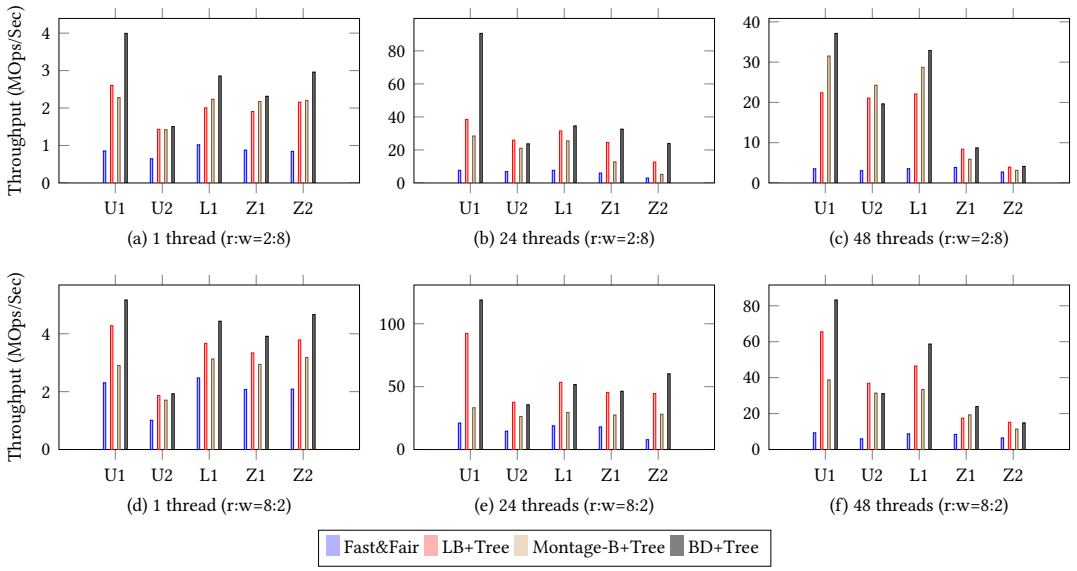


Fig. 5. Throughput of persistent B+ trees with write-heavy (top) or read-heavy (bottom) workloads, with 1 (left), 24 (middle), or 48 (right) threads.

to KV pairs in NVM. Each KV pair in Montage-B+Tree occupies a separately allocated block of NVM; in BD+Tree and LB+Tree, NVM allocation and deallocation occur only when (many-key) leaf nodes are created or destroyed. The performance difference between Montage-B+Tree and BD+Tree diminishes when worker threads are distributed across two sockets, as Montage-B+Tree's transient leaves render it less sensitive to NUMA effects [12].

4.1.2 Epoch length. While BDL reduces persistence overhead and NVM traffic by amortizing persistence overhead across large operation batches and optimizing cache reuse, extending epoch length does not invariably yield improved performance. Epoch length might be expected to impact performance directly in two ways:

(1) **Cache misses:** The background thread, running at epoch boundaries, may write data back to memory using instructions (clflushopt or, on some machines, clwb) that evict the data from the cache, inducing subsequent misses in any worker thread that accesses the same data. A longer epoch does not necessarily result in fewer cache misses, however: by increasing both the working set of an epoch and the number of copies that must be retained from earlier epochs (some of which share cache lines with active data), it may induce more capacity misses. (We explore the issue of retained copies further in Section 4.1.7.)

(2) **Memory interference:** In a multithreaded environment, an NVM writer thread may negatively impact the read and write performance of other threads due to contention for shared memory resources, including the Read/Write Pending Queues (RPQ/WPQ) in integrated Memory Controllers (iMC) and the XPBuffers in NVM DIMMs, particularly under high contention [8, 41]. The interference between the background persisting thread and worker threads is influenced by the frequency of persistence, the data volume written to NVM, and (again) the number of data copies retained from earlier epochs, all of which depend on epoch length.

We assess sensitivity to epoch length using workload U2, as it consistently displays the lowest throughput. Experiments employ the same configuration as in Sec 4.1, tested with 50% insert

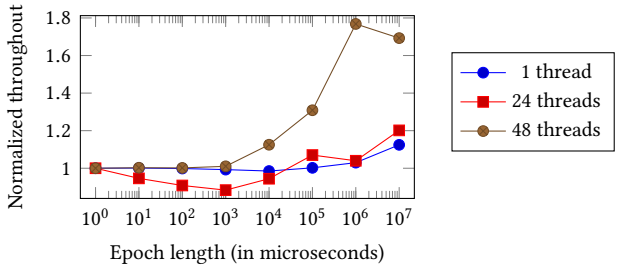


Fig. 6. Variance in throughput as a function of epoch length (relative to epoch length = 1 μs). At the shortest epoch length, actual throughputs for 1, 24, and 48 threads are 1.4, 20.6, and 11.4 million operations per second, respectively.

operations and 50% delete operations, with epochs ranging from 1 μs to 10 seconds. BD+Tree requires ~148 MiB of NVM to store leaf nodes (excluding KV duplicates) and ~9 MiB of DRAM to store inner nodes. Results appear in Fig. 6.

The impact of epoch length on performance depends heavily on the cache size and thread count. With a single thread, performance remains stable. With 48 threads, however, performance improves with longer epochs. This behavior is mainly due to cache effects: in the single-thread case, misses are mainly due to cache replacement, and epoch length is largely irrelevant. With 48 threads, however, spread across two sockets, we have a combined 154 MiB of cache—enough to hold a significant portion of the tree. Misses are then largely due to explicit flushes, and longer epochs allow these to occur less often.

With 24 threads on a single socket, performance degrades by about 10% as the epoch length increases from 1 μs to 1 ms, primarily due to eager eviction of now-cold data in very short epochs. At this point, however, performance begins to improve, as longer epochs reduce contention for cache and memory resources between workers and the background persisting thread: throughput at an epoch length of 10 s is roughly 13% higher than at 1 ms.

Our results suggest that when ample cache resources are available (relative to the working set size), extending the epoch length can improve performance by maximizing cache reuse. When workloads cannot benefit from cache reuse, if memory contention is low, shorter epochs can reduce post-crash data loss without hurting performance. Under high memory contention, longer epochs are preferred to prevent the background writer from hindering the progress of foreground workers. Results elsewhere in this paper use an epoch length of 50 ms when running on a single socket and 500 ms when running across two sockets.

4.1.3 Memory reclamation frequency. When the worker thread and memory reclaimer run on the same socket, our experiments demonstrate that memory reclamation frequency has a negligible impact on performance, regardless of workload characteristics and access patterns: contention between between the background memory reclaimer and worker threads is low, and leaf nodes become empty—and thus reclaimable—infrequently. However, when the memory reclamation thread runs on a different socket, a high frequency leads to significant performance loss, as the metadata cache lines of all leaf nodes are moved to shared mode in the cache, forcing workers to re-acquire exclusive ownership before making subsequent updates. We mitigate this issue by binding the memory reclamation thread to the same socket as the worker threads. When threads operate across multiple sockets, we employ a longer inter-reclamation interval (500 ms, rather than 50 ms). Since empty slots resulting from deletions are often quickly refilled by insertions, the

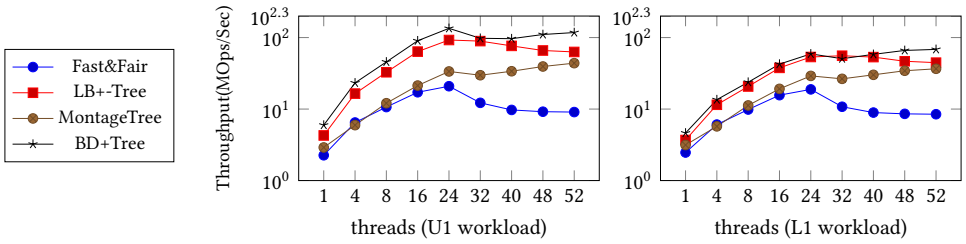


Fig. 7. Scalability of persistent B+ trees for a read-heavy (r:w=8:2) workload.

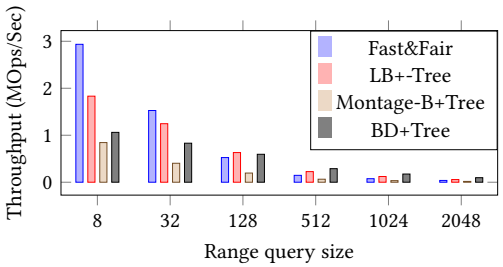


Fig. 8. Range query performance v. query size.

majority of nodes remain occupied [18]. Setting the reclamation interval at 500 ms does not lead to a noticeable increase in memory usage for any of our workloads.

4.1.4 Scalability. Given the results of epoch length and memory reclamation frequency analysis, we conducted experiments to evaluate the scalability of the four persistent trees. We set the read-write ratio at 8:2, reflecting the prevalence of read requests in real-world workloads. Results appear in Fig. 7. Performance decline was observed for all trees when accesses span across sockets. Notably, while BD+Tree is designed to provide a significant performance advantage in write-intensive workloads (by batching writes-back and increasing cache reuse), it also performs extremely well in read-intensive workloads.

4.1.5 Range queries. When searching for a range keys, BD+Tree performs better for large ranges than for small ones. Details appear in Fig. 8. For ranges smaller than a single node, BD+Tree must sort the pairs within the node. If a query spans nodes, it must still sort the starting and ending nodes, but the middle ones avoid this overhead. Fast&Fair shows superior performance for small-sized queries due to its sorted leaf nodes. As the query size increases, however, the advantage diminishes, and the higher traversal overhead in NVM-based inner nodes becomes more pronounced. LB+Tree outperforms BD+Tree for small-sized queries, as BD+Tree's larger node size leads to higher NVM read and sorting overheads. Montage-B+Tree exhibits the lowest performance due to the necessity of following pointers in leaf nodes to retrieve each KV pair.

4.1.6 Overhead in BD+Tree. BD+Tree introduces two types of overhead: bookkeeping and computation. Bookkeeping overhead, from tracking dirty data within an epoch, is negligible as it is offloaded to the background thread. Computation overhead, from retrieving and updating epoch-related information within leaf nodes, also incurs minimal performance impact. This is because metadata cache lines, which contain epoch information, need to be loaded into the cache before

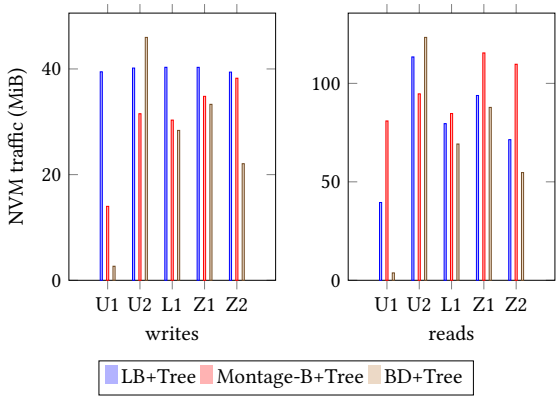


Fig. 9. NVM traffic per million operations (in MiB), with 24 active threads (r:w=2:8).

subsequent operations. Therefore, reads and writes of epoch information are typically served by the cache. Additionally, epoch-related updates occur only once every few milliseconds.

4.1.7 NVM traffic and space efficiency. We compared the NVM traffic (reads and writes) and space efficiency of the three hybrid and unsorted trees: LB+Tree, Montage-B+Tree, and BD+Tree. Fast&Fair was excluded from this comparison due to its fully persistent and sorted nature.

We used Intel's PCM tool [16] to measure NVM traffic, normalizing to MiB per million operations to account for varying throughput among trees. PCM counts the number of requests (in cache lines) directed to the iMC. For a small workload (U1), as shown in Fig. 9, BD+Tree reduces both reads and writes to NVM by ~90% relative to LB+Tree. For larger workloads with good temporal locality (L1, Z1, Z2), BD+Tree reduces NVM writes by 30–44% and reads by 6–24%. In the less favorable situation of uniform access to a large working set (U2), BD+Tree incurs 12% more NVM writes and 9% more NVM reads due to KV duplicates (for keys updated in the two most recent epochs) and to the inability to combine writes in cache.

To quantify the space overhead of each tree, we measured the amount of DRAM and NVM used to store tree nodes. Results appear in Table 3. Montage-B+Tree requires 10× more DRAM, since the entire tree (including leaves) resides in DRAM. It utilizes less NVM because it persists only KV pairs. LB+Tree and BD+Tree, by contrast, store data at the granularity of leaf nodes, which include metadata and empty slots not occupied by any KV pairs. BD+Tree consumes less DRAM than LB+Tree due to its larger leaf node size; NVM consumption is almost the same.

One could expect space consumption in BD+Tree to depend on epoch length: with long epochs, a larger number of keys will be updated in each epoch, and must be retained until two epoch boundaries have passed. We model this using a uniform access pattern as it serves as an upper bound. Consider a tree whose key values are drawn from $1..N$. Suppose we initially fill the tree with $N/2$ distinct keys and then perform a long sequence of update operations (50:50 inserts and

	LB+Tree	Montage-B+Tree	BD+Tree
DRAM	16.32	172.09	9.11
NVM	168.63	96.27	175.80

Table 3. DRAM and NVM usage (in MiB) of persistent trees under workload L1.

removes), with a uniformly random choice of key for each. Suppose further that M operations can be performed per epoch.

The probability that a specific key is accessed by a given write operation is $1/N$. The probability that the key is *not* accessed for an entire epoch is thus $((N-1)/N)^M$, and the probability that the key is accessed at least once in the epoch is $1-((N-1)/N)^M$. Since a key instance is created only on insertion, the probability of generating a key instance in the key's final operation of the epoch (if any) is $[1-((N-1)/N)^M]/2$. This calculation holds for the current epoch e and for the previous epoch $e-1$. The probability of retaining a key instance from epoch $e-2$ or earlier should converge to $1/2$. Consequently, for a given key, the number of instances maintained by BD+Tree can be expressed as

$$\overbrace{\tfrac{1}{2}}^{<\text{e-1}} + \overbrace{\tfrac{1}{2}\left(1-\left(\tfrac{N-1}{N}\right)^M\right)}^{e-1} + \overbrace{\tfrac{1}{2}\left(1-\left(\tfrac{N-1}{N}\right)^M\right)}^{e} = \frac{3}{2} - \left(\frac{N-1}{N}\right)^M$$

With a space of N possible keys, the total number of key instances maintained by BD+Tree can be expected to be

$$N \cdot \left(\frac{3}{2} - \left(\frac{N-1}{N}\right)^M\right)$$

So long as M is small relative to N, $[(N-1)/N]^M$ will be close to 1, and we will have only slightly more than $N/2$ total key instances. (If M is $N/10$, for example, we will have about 0.6 instances per key.) To validate our model, we executed workload U2 with a single thread for 10 seconds. Using a single thread ensures a constant throughput (operations per second) across different epoch lengths, allowing us to adjust the total number of operations, M, within an epoch by altering the epoch length. Experimental results matched the model very closely. The number of key instances remained very close to $N/2$ for all epoch lengths below a tenth of a second, indicating a negligible increase in NVM space consumption.

4.2 Real-world workload evaluation

We use trace data from Memcached (Table 1) to further investigate the performance and NVM read/write behavior of the four persistent B+ trees under real-world workloads. With each workload comprising 100 million operations, working sets are as shown in the last column of the table. Results appear in Fig. 10. Relative to LB+Tree, BD+Tree achieves a 1.1–1.4× speedup under the read-heavy workload (t1) and a 1.5–2.4× speedup when the working set is small due to high temporal locality (t2). Notably, despite displaying inferior performance in the large synthetic uniform workload (U2), BD+Tree is comparable to its competitors (with fewer than 40 threads) or even better (with more than 40 threads) in the real-world uniform workload with its markedly larger working set (t3).

In contrast to our synthetic microbenchmarks, real-world workloads often exhibit high temporal locality even when all keys are accessed equally often on a large time scale.

In another contrast to the synthetic case, Montage-B+Tree performs comparably to LB+Tree and BD+Tree on the real-world t1 and t3 workloads. We attribute this to the larger real-world working sets. With synthetic workloads, LB+Tree and BD+Tree can retain part of the leaf layer in the cache, thereby reducing NVM accesses. In contrast, Montage-B+Tree fills the cache with volatile leaf nodes, leading to more frequent and costly NVM accesses. With larger workloads, LB+Tree and BD+Tree experience a higher NVM access penalty—first, because a larger working set results in lower cache reuse; second, because LB+Tree and BD+Tree read both metadata and KV cache lines from NVM, whereas Montage-B+Tree stores the metadata cache line in DRAM and only loads the

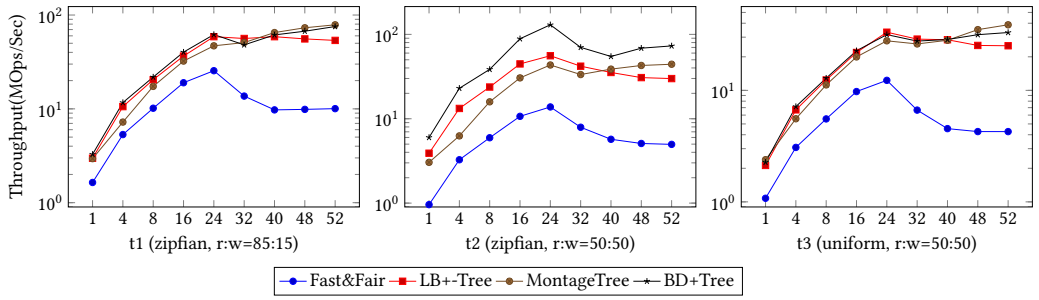


Fig. 10. Throughput of persistent B+ trees running with real-world workloads.

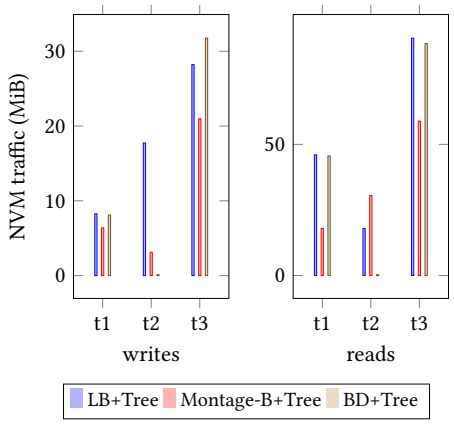


Fig. 11. NVM traffic generated per million operations (in MiB). with 24 active threads.

KV pair from NVM, effectively halving its NVM accesses. Notably, Montage-B+Tree achieves this saving at the cost of 10× more DRAM usage.

As shown in Fig. 11, BD+Tree reduces NVM traffic by 99% on workload t2 as a result of better cache reuse. With the large uniform workload t3, it increases NVM write traffic by ~10%, while only expanding space usage by ~1%.

4.2.1 Performance with very large caches. As cache sizes continue to grow, it becomes increasingly important to avoid self-eviction in persistent applications, to fully utilize cache resources. BD+Tree can be expected to deliver higher performance with larger caches due to better cache reuse, even for uniformly distributed workloads, as long as the working set is not significantly larger than the cache size. To confirm this expectation, we conducted experiments on a server with an AMD EPYC 7773X processor featuring 768 MiB of L3 cache (8 chiplets, 96 MiB each). We compared BD+Tree, LB+Tree, and Montage-B+Tree using the large uniform workloads U2 and t3, where BD+Tree loses its advantage over prior art on the Intel Server due to its smaller cache size (2 sockets, 42 MiB each). A single thread was used due to AMD's lack of HTM support. As the AMD server does not support real NVM, our results underestimate the advantages of BD+Tree due to the lower cache miss penalty of DRAM accesses. Even so, BD+Tree achieved speedups of 1.55× and 1.45× compared to LB+Tree and Montage-B+Tree, respectively, under U2 (20% reads, 80% writes), and 1.31× and 1.56× when U2 is read-heavy (80% reads, 20% writes). Furthermore, BD+Tree outperforms LB+Tree and Montage-B+Tree by 1.63× and 1.13×, respectively, under workload t3. The results underscore

the importance of effectively utilizing cache resources, and highlight the performance potential of BD+Tree.

4.3 Recovery time

Fast&Fair achieves instant post-crash recovery due to its full persistence. BD+Tree and LB+Tree reconstruct the internal tree from the persistent (linked) leaf layer. Montage-B+Tree, however, stores the entire tree in DRAM and persists only KV pairs. Consequently, the recovery process needs to scan the entire heap to gather those KV pairs, sorting them and reconstructing both the internal tree and the leaves. To recover a B+ tree containing 10 million KV pairs, LB+Tree and BD+Tree take ~200 μs, while Montage-B+Tree needs more than 1.4 ms.

5 Conclusions

We have presented BD+Tree, a buffered durably linearizable B+ tree that attains better performance and reduced NVM wear-out, relative to prior persistent trees, via relaxed persistence. Relaxation amortizes persistence overhead over large batches of operations, and enhances cache reuse to improve throughput and reduce the volume of NVM writes. Our evaluation demonstrates that for workloads with small working set or with good temporal locality—common traits in real-world scenarios—BD+Tree can deliver a performance improvement ranging from 1.1× to 2.4× and a reduction in NVM writes of up to 99% compared to state-of-the-art hybrid persistent B+ trees. Our results highlight the critical role of cache reuse in persistent memory programming. As index sizes for real-world data repositories continue to exceed the capacity of even the largest on-chip caches, fully capitalizing on this resource will be imperative for both performance and NVM endurance.

Acknowledgments

This work was supported in part by the U.S. National Science Foundation, grants numbers CNS-1955498 and CNS-1900803, and by a Google Faculty Research Award.

References

[1] Gal Assa, Andreia Correia, Pedro Ramalhete, Valerio Schiavoni, and Pascal Felber. 2023. TL4x: Buffered Durable Transactions on Disk as Fast as in Memory. In *28th ACM Symp. on Principles and Practice of Parallel Programming*. 245–259.

[2] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload Analysis of a Large-scale Key-value Store. In *12th ACM Joint Intl. Conf. on Measurement and Modeling of Computer Systems*. 53–64.

[3] Wentao Cai, Haosen Wen, H Alan Beadle, Chris Kjellqvist, Mohammad Hedayati, and Michael L Scott. 2020. Understanding and Optimizing Persistent Memory Allocation. In *19th ACM SIGPLAN Intl. Symp. on Memory Management*. 60–73.

[4] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conf. on File and Storage Technologies*. 209–223.

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

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

[7] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *1st ACM Symp. on Cloud Computing*. 143–154.

[8] Björn Daase, Lars Jonas Bollmeier, Lawrence Benson, and Tilmann Rabl. 2021. Maximizing Persistent Memory Bandwidth Utilization for OLAP Workloads. In *2021 Intl. Conf. on Management of Data*. 339–351.

[9] Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E Blelloch, and Erez Petrank. 2020. NVTraverse: In NVRAM Data Structures, the Destination Is More Important Than the Journey. In *41st ACM Conf. on Programming Language Design and Implementation*. 377–392.

[10] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. 2018. A Persistent Lock-Free Queue for Non-Volatile Memory. In *23rd ACM Symp. on Principles and Practice of Parallel Programming*. 28–40.

[11] Michal Friedman, Erez Petrank, and Pedro Ramalhete. 2021. Mirror: Making Lock-free Data Structures Persistent. In *42nd ACM Conf. on Programming Language Design and Implementation*. 1218–1232.
[12] Shashank Gugnani, Arjun Kashyap, and Xiaoyi Lu. 2020. Understanding the Idiosyncrasies of Real Persistent Memory. *Proceedings of the VLDB Endowment* 14, 4 (2020), 626–639.
[13] Swapnil Haria, Mark D. Hill, and Michael M. Swift. 2020. MOD: Minimally Ordered Durable Datastructures for Persistent Memory. In *25th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*. 775–788.
[14] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems* 12, 3 (July 1990), 463–492.
[15] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree. In *16th USENIX Conf. on File and Storage Technologies*. 187–200.
[16] Intel Corporation. [n. d.]. Intel Performance Counter Monitor. https://github.com/intel/pcm.
[17] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. 2016. Linearizability of Persistent Memory Objects Under A Full-system-crash Failure Model. In *30th Intl. Symp. on Distributed Computing*. 313–327.
[18] Theodore Johnson and Dennis Shasha. 1989. Utilization of B-trees with Inserts, Deletes and Modifies. In *8th ACM Symp. on Principles of Database Systems*. 235–246.
[19] Ana Khorguani, Thomas Ropars, and Noel De Palma. 2022. ResPCT: Fast Checkpointing in Non-volatile Memory for Multi-threaded Applications. In *17th European Conf. on Computer Systems*. 525–540.
[20] Wook-Hee Kim, R Madhava Krishnan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. 2021. PACTree: A High Performance Persistent Range Index Using PAC Guidelines. In *28th ACM Symp. on Operating Systems Principles*. 424–439.
[21] R Madhava Krishnan, Wook-Hee Kim, Xinwei Fu, Sumit Kumar Monga, Hee Won Lee, Minsung Jang, Ajit Mathew, and Changwoo Min. 2021. TIPS: Making Volatile Index Structures Persistent with DRAM-NVMM Tiering. In *2021 USENIX Annual Technical Conf.* 773–787.
[22] Christoph Lameter. 2005. Effective Synchronization on Linux/NUMA Systems. In *Gelato Conference*, Vol. 2005.
[23] Se Kwon Lee, K Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H Noh. 2017. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems. In *15th USENIX Conf. on File and Storage Technologies*. 257–270.
[24] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. Recipe: Converting Concurrent Dram Indexes to Persistent Memory Indexes. In *27th ACM Symp. on Operating Systems Principles*. 462–477.
[25] Tongliang Li, Haixia Wang, Airan Shao, and Dongsheng Wang. 2022. SSB-Tree: Making Persistent Memory B+-Trees Crash-Consistent and Concurrent by Lazy-Box. In *36th IEEE Intl. Parallel and Distributed Processing Symp.* 70–80.
[26] Jihang Liu, Shimin Chen, and Lujun Wang. 2020. LB+ Trees: Optimizing Persistent Index Performance on 3DXPoint Memory. *Proceedings of the VLDB Endowment* 13, 7 (2020), 1078–1090.
[27] Mengxing Liu, Jiankai Xing, Kang Chen, and Yongwei Wu. 2019. Building Scalable NVM-based B+ tree with HTM. In *48th Intl. Conf. on Parallel Processing*. 1–10.
[28] Yongping Luo, Peiquan Jin, Zhou Zhang, Junchen Zhang, Bin Cheng, and Qinglin Zhang. 2021. Two Birds with One Stone: Boosting Both Search and Write Performance for Tree Indices on Persistent Memory. *ACM Transactions on Embedded Computing Systems* 20, 5s (2021), 1–25.
[29] Shaonan Ma, Kang Chen, Shimin Chen, Mengxing Liu, Jianglang Zhu, Hongbo Kang, and Yongwei Wu. 2021. ROART: Range-Query Optimized Persistent ART. In *19th USENIX Conf. on File and Storage Technologies*. 1–16.
[30] Amirsaman Memaripour, Joseph Izraelevitz, and Steven Swanson. 2020. Pronto: Easy and Fast Persistence for Volatile Data Structures. In *25th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*. 789–806.
[31] Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B. Morrey III, Dhruva R. Chakrabarti, and Michael L. Scott. 2017. Dalí: A Periodically Persistent Hash Map. In *31st Intl. Symp. on Distributed Computing*. 37:1–37:16.
[32] Storage Networking Industry Association. [n. d.]. Key-Value Traces. http://iotta.snia.org/traces/key-value.
[33] Chundong Wang and Sudipta Chattopadhyay. 2020. Isle-tree: A B+-tree with Intra-cache Line Sorted Leaves for Non-volatile Memory. In *38th IEEE Intl. Conf. on Computer Design*. 573–580.
[34] Ke Wang, Guanqun Yang, Yiwei Li, Huanchen Zhang, and Mingyu Gao. 2023. When Tree Meets Hash: Reducing Random Reads for Index Structures on Persistent Memories. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–26.
[35] Ziqi Wang, Chul-Hwan Choo, Michael A Kozuch, Todd C Mowry, Gennady Pekhimenko, Vivek Seshadri, and Dimitrios Skarlatos. 2021. NVOverlay: Enabling Efficient and Scalable High-frequency Snapshotting to NVM. In *48th Intl. Symp. on Computer Architecture*. 498–511.
[36] Yuanhao Wei, Naama Ben-David, Michal Friedman, Guy E Blelloch, and Erez Petrank. 2022. FliT: A Library for Simple and Efficient Persistent Algorithms. In *27th ACM Symp. on Principles and Practice of Parallel Programming*. 309–321.
[37] Haosen Wen, Wentao Cai, Mingzhe Du, Louis Jenkins, Benjamin Valpey, and Michael L. Scott. 2021. A fast, General System for Buffered Persistent Data Structures. In *50th Intl. Conf. on Parallel Processing*. 1–11.

[38] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *13th USENIX Conf. on File and Storage Technologies*. 167–181.
[39] Juncheng Yang, Yao Yue, and KV Rashmi. 2021. A Large-scale Analysis of Hundreds of In-memory Key-value Cache Clusters at Twitter. *ACM Transactions on Storage* 17, 3 (2021), 1–35.
[40] Yue Yang and Jianwen Zhu. 2016. Write Skew and Zipf Distribution: Evidence and Implications. *ACM Transactions on Storage* 12, 4 (2016), 1–19.
[41] Jifei Yi, Benchao Dong, Mingkai Dong, Ruizhe Tong, and Haibo Chen. 2022. MT^2: Memory Bandwidth Regulation on Hybrid NVM/DRAM Platforms. In *20th USENIX Conf. on File and Storage Technologies*. 199–216.
[42] Xinjing Zhou, Lidan Shou, Ke Chen, Wei Hu, and Gang Chen. 2019. DPTree: Differential Indexing for Persistent Memory. *Proceedings of the VLDB Endowment* 13, 4 (2019), 421–434.
[43] Xiaomin Zou, Fang Wang, Dan Feng, Tianjin Guan, and Nan Su. 2022. ROWE-tree: A Read-Optimized and Write-Efficient B+-tree for Persistent Memory. In *51st Intl. Conf. on Parallel Processing*. 1–11.

Received April 2024; revised July 2024; accepted August 2024