Reconciling Hardware Transactional Memory and Persistent Programming with Buffered Durability

Mingzhe Du
mdu5@cs.rochester.edu
University of Rochester
Rochester, New York, USA

Ziheng Su
zsu8@u.rochester.edu
University of Rochester
Rochester, New York, USA

Michael L. Scott
scott@cs.rochester.edu
University of Rochester
Rochester, New York, USA

Abstract

Hardware Transactional Memory (HTM) simplifies concurrent programming and can accelerate multithreaded execution through lock elision. Non-Volatile Memory (NVM) combines the speed and byte addressability of DRAM with the durability of storage, enabling the construction of high-performance, persistent data structures. Unfortunately, the write-back instructions typically needed to ensure post-crash consistency in NVM cause HTM transactions to abort, precluding the straightforward combination of HTM and persistent data structures. The problem goes away on machines with persistent caches, but these require special battery-backed circuitry and are far from commonplace.

To combine HTM and persistent data structures, we advocate for *buffered durable linearizability (BDL)*, a relaxed correctness criterion that enables recovery to a "recent" consistent state in the wake of a crash, allowing writes-back to occur outside transactions.

Significantly, BDL retains the persistence guarantees of storage systems—such as databases backed by disks or flash—that have relied on buffering for decades.

The combination of HTM and buffered durability enables three separate usage scenarios. First, we add durability to an existing HTM-based structure (a van Emde Boas tree due to Khalaji et al.); second, we use HTM to simplify an existing persistent structure (a skiplist due to Wang et al.); third, we "back port" an HTM-based structure optimized for persistent caches (a hash table due to Zhang et al.) to work well on more conventional processors. The first two scenarios yield several-fold improvements in throughput; the third sees very little slowdown.

CCS Concepts

• **Theory of computation** → **Concurrent algorithms**; *Data structures and algorithms for data management*; • **Software and its engineering** → **Concurrency control**; • **Hardware** → *Non-volatile memory*; • **Computer systems organization** → **Processors and memory architectures**.

SPAA '25, Portland, OR, USA

ACM ISBN 979-8-4007-1258-6/2025/07
https://doi.org/10.1145/3694906.3743321

Keywords

hardware transactional memory, non-volatile memory, persistent concurrent data structures

ACM Reference Format:
Mingzhe Du, Ziheng Su, and Michael L. Scott. 2025. Reconciling Hardware Transactional Memory and Persistent Programming with Buffered Durability. In *37th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '25), July 28-August 1, 2025, Portland, OR, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3694906.3743321

1 Introduction

Implementing correct, high-performance multithreaded code can be both challenging and error-prone. For many concurrent structures, Hardware Transactional Memory (HTM) is a powerful tool, enabling concurrent execution of disjoint operations without fine-grain locking or subtle nonblocking protocols. Commodity HTM implementations, such as Intel's Transactional Synchronization Extensions (TSX), provide instructions to bracket code regions that should execute atomically. The implementation of these instructions relies on the cache coherence protocol to track and manage speculatively read and written lines. Changes are made globally visible only upon a successful commit—in particular, in the absence of conflicting concurrent activity—ensuring that all accesses of a given transaction appear to happen atomically. If a conflict arises (detected via cache eviction), the transaction is aborted. When this occurs, the processor rolls back any speculative changes, restoring the system to its prior state without affecting other cores. Despite the fact that current ("best-effort") hardware has limited transaction size and is subject to spurious aborts, HTM remains a powerful tool for fine-grained concurrency control due to its easy-to-use interface and minimal overhead compared to traditional locking mechanisms [6, 41, 44, 47, 49, 64]. Unfortunately, existing uses of HTM do not transfer in any straightforward way to persistent structures.

Non-volatile memory (NVM) offers DRAM-like performance with the durability of storage, allowing persistent data structures to be retained across system crashes. Unlike their transient counterparts, however, persistent structures require additional programming effort to ensure crash consistency. Two problems account for this extra effort. First, operations that have only partially completed when a crash occurs must be rolled forward or backward during recovery. This problem can be addressed with logging or nonblocking algorithms. Second, CPU caches remain volatile on most machines, and cache lines are written back to memory in an unpredictable order determined by the cache replacement policy. As a result, data may reach NVM out of order from the program's perspective, potentially leading to inconsistent state after a crash. To address this problem, programmers use explicit write-back (e.g.,

clflush, clflushopt, and clwb on x86) and fence instructions to ensure that data reach NVM in the correct order. Without these extra "persist" instructions, critical data could remain dirty in cache and be lost in the event of a crash, resulting in data corruption.

While Intel has disabled TSX by default in recent products (due to side-channel security concerns), IBM's mainframe (z-series) machines continue to leverage HTM for concurrency control, and ARM has recently integrated HTM support into their instruction set. Likewise, while Intel's Optane persistent memory was canceled in 2022, manufacturers are pursuing future products, and the widespread adoption of CXL [13] seems likely to allow even disaggregated DRAM to appear nonvolatile from the host's perspective. These trends suggest a future in which both HTM and NVM continue to play important roles.

Naively, one might hope that HTM could be used to achieve not only isolation among concurrent operations but also post-crash atomicity and consistency: simply enclose each operation—including its write-back instructions—in a hardware transaction. Unfortunately, write-back instructions are incompatible with HTM because they violate isolation: while HTM requires memory operations to be speculatively executed and invisible to other cores until a transaction commits, writes-back force data to be written to NVM, making them globally visible prematurely. As a result, developers generally use locks to synchronize concurrent NVM writes, leading to tradeoffs between programming complexity (lock granularity) and performance [29, 32, 35, 51, 52]. Designing lock-free persistent data structures is even more challenging and error-prone: in addition to managing concurrency, programmers must carefully insert persist operations before reads and after writes to ensure that updates are both atomic and durable [25]. Beyond the added complexity, excessive use of persist instructions introduces significant performance overhead and contributes to NVM wear-out [2, 12, 54].

Persistent cache, marketed by Intel as eADR and available on third generation Xeon processors, resolves the incompatibility between NVM and HTM, at least in part. With CPU caches included in the persistence domain, explicit persist operations are no longer needed to ensure the durability of data stored in the cache. Data structures designed for volatile memory (even those synchronized with HTM) and those designed for nonvolatile memory (with persist instructions removed) can be backed by NVM on an eADR machine and will remain consistent after a crash, though logging may still be required for roll-forward or roll-back of operations that use locks (either by default or when HTM fails).

Even with eADR, however, transient data structures originally designed for DRAM may experience suboptimal performance on NVM. With Intel Optane memory, for example, the physical medium is accessed at multi-cache-line granularity (256 bytes in the first generation of hardware, 128 bytes in the second). Moreover, NVM writes have one third the bandwidth and three times the latency of reads [4, 22]. As a result, transient data structures running on NVM may suffer from write amplification, unnecessary wear-out, and significant performance degradation.

Recent work suggests that data structures should be custom designed for persistent cache and NVM, together, in order to maximize performance [62, 63, 65, 66]. Some of this work makes use of HTM [62, 63]. Much of it can be seen as a step toward full-system persistence and crash recovery [26, 27, 60]. Given the cost and complexity of the battery backup required for eADR, however, most systems seem likely to come with volatile caches for the foreseeable future. To integrate HTM with NVM on "plain ADR" machines, designers have developed methods to avoid persist instructions within transactions [10, 14, 20, 33, 40, 42, 57, 59]. Oukid et al. [42], for example, keep interior tree nodes in DRAM, where they are updated using HTM; leaf nodes, in NVM, are synchronized with locks. David et al. [14] describe a *link cache*, synchronized with HTM, that heuristically delays the write-back of updated pointers. Liu et al. [33] use HTM for tree traversal but also for atomic leaf update: writers synchronize among themselves by holding a spin lock during updates and write-back; they use a hardware transaction *inside* the spin lock critical section to synchronize with readers, which do not acquire the lock. Genç et al. [20] use two HTM transactions to commit a single operation: the first transaction gathers all NVM writes in a redo log that is persisted after the transaction commits. A second transaction then performs the actual program writes in NVM. Other researchers have suggested special hardware for persistent logging [1, 7, 38].

The incompatibility between HTM and NVM programming arises from the write-back instructions used to achieve strict durability—to persist each critical update immediately. We argue, however, that such strictness is unnecessary for many data structures—notably the trees, skiplists, and hash tables that serve as indexes in storage systems. These systems have, for decades, successfully relied on buffering within the operating system, typically on a millisecond timescale. By aligning the persistence model of an index with its underlying storage—i.e., by extending buffered persistence to byte-addressable data structures—we can arrange for writes-back to be delayed until after the completion of each HTM transaction, resolving the conflict of NVM and HTM without compromising overall persistence guarantees, as long as the indexing structure remains consistent with the storage system.

Rather than persist each operation before returning to the caller, a *buffered durably linearizable (BDL)* [25, 37] data structure performs persistence in the background and arranges, in the wake of a crash, to recover to a bounded-ly recent consistent state. Typically, the history of a persistent data structure is divided into *epochs* of a few milliseconds each. NVM writes are buffered within an epoch, and a crash in epoch e recovers to the state of the structure at the end of epoch $e - 2$. By delaying persist operations, buffered durability removes HTM-conflicting persistence from transactions, allowing persistent data structures to run safely with commodity HTM without requiring hardware changes or software logging. Buffered durability also allows the system to continue executing new epochs while persisting data from earlier epochs, avoiding execution pauses. Unlike checkpointing or snapshots, BDL does not require that all data fit in DRAM, or that they be copied in their entirety in the course of periodic backups.

The combination of buffered durability and hardware transactions offers benefits in at least three key scenarios. First, transient structures that use HTM for scalability and performance can be made persistent with buffered durability. Second, persistent structures that rely on complex locking and logging protocols can be simplified (and often made faster) using HTM and buffering. Third, persistent structures that use HTM on eADR machines can be "back

ported" to plain ADR while retaining optimizations for NVM block size, latency, bandwidth, and write endurance.

Summarizing contributions, we:

- add buffered durability to the HTM-based van Emde Boas tree of Khalaji et al. [28], achieving throughput up to 4× higher than in state-of-the-art persistent trees.
- add buffered durability and HTM to the persistent skiplist of Wang et al. [54], achieving throughput up to 3× higher than that of the original nonblocking version while retaining most of its preemption tolerance.
- adapt the HTM-based hash table of Zhang et al. [62], which was optimized for persistent caches, to run on more conventional processors, preserving its optimizations, its use of HTM, and most of its performance.
- provide guidelines, including optimization opportunities and pitfalls, for combining HTM with buffered durability.

Source code for our example data structures is available at https://github.com/urcs-sync/BD-HTM.git.

2 Background

2.1 Persistent Programming Model

Linearizability [23] is the standard safety criterion for concurrent data structures. It requires (1) that every sequence of (potentially overlapping) operations have the same effect (same arguments and return values) as some sequential (nonoverlapping) history that respects the semantics of the structure and (2) that if operation A returns before operation B is called, then A appears to happen before B. Equivalently, one must be able to identify an instruction (the *linearization point*) in every operation at which the operation appears to take effect.

Durable linearizability (DL) [19, 25] extends the notion of linearizability to accommodate "full system" crashes, in which all threads stop running and any future operations employ a new set of threads. Equivalently, it must be possible to identify an instruction (the *persist point*) in every operation after which its effects will survive a crash. In any execution, linearization points and persist points must occur in the same order.

Buffered durable linearizability (BDL) [25] 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 the previous inter-crash interval. Compared to DL, BDL allows operations to be batched and persisted together, thereby amortizing persistence overhead. Previous work [3, 37, 55] shows that BDL can greatly improve performance and reduce NVM writes.

2.2 Hardware Transactional Memory

A sequential data structure can easily be made thread safe by adding a coarse-grain lock, but performance is likely to be terrible. Fine-grain locks may enable concurrent access, but their integration into nontrivial structures is typically both labor intensive and error prone. Faced with this tradeoff, programmers often make compromises. Search trees, for example, may give up on dynamic rebalancing, abandoning the guarantee of log-time operations [5, 16]. In a similar vein, Kułakowski's parallelization of the van Emde Boas (vEB) search tree, whose sequential version is doubly logarithmic, fails to maintain this time complexity and (in the original published version) admits extreme cases in which a successor operation returns a failure indication instead of an actual result [30, 31].

Transactional memory (TM) aims to combine the simplicity of coarse-grain locks with the concurrency of fine-grain locks. It offers a simple programming model—label the blocks of code that need to be atomic—on top of a speculative implementation—assume that concurrent atomic sections are mutually independent, execute them in parallel, and back out and retry in the event of conflict. Researchers have used HTM to create fully featured versions of many concurrent structures, including binary [47] and vEB [28] trees.

Current hardware TM, as offered by Intel, IBM, and Arm, is said to offer "best effort" behavior: its transactions may abort not only due to conflict with other transactions but for a variety of additional reasons. On Intel platforms, speculative writes are limited to the size of the L1 cache, and reads to that plus a Bloom-filter summary of evicted lines [9]. Transactions that exceed these limits may abort deterministically. They may also abort in response to various transient events, including page faults, timer expiration, I/O interrupts, or L1 associativity misses. Finally, they will abort if they attempt to execute various unsupported instructions—notably including the clflush, clflushopt, and clwb instructions used to flush or write lines back to NVM. Given the possibility of repeated (even indefinitely repeated) aborts, best-effort HTM must be combined with a "fallback path" in software that guarantees forward progress, typically using a global lock.

To minimize the size and duration of transactions, some researchers have used HTM to build atomic primitives that can serve as building blocks for more complex operations on lock-free concurrent structures. Makreshanski et al. [49] use HTM to implement a Multi-word Compare-and-Swap (MwCAS) that can update arbitrary memory locations atomically. Brown et al. [6] provide a similar HTM version of Load-Link/Store-Conditional (LL/SC): their primitive snapshots multiple locations and then atomically updates one of them if none has been modified by a writer in between.

2.3 Persistent Data Structures and Concurrency Control

Persistent data structures can be created automatically from nonblocking transient versions by inserting persist instructions at appropriate points in the code [25], but this approach does not lead to good performance. Much research focuses on developing new designs that minimize overhead and accommodate the idiosyncrasies (block size, latency, bandwidth, wear-out) of NVM. Any such design must address two key correctness challenges. First, persistent data structures must carefully manage the order of persistence to avoid inconsistency after crashes. A newly inserted list node, for example, must be persisted before creating a pointer to it. Otherwise, the pointer could be written back to memory first, leaving it referencing garbage in the event of an intervening crash. Second, there is typically a discrepancy between the point of visibility (PoV) and the point of persistence (PoP) for a given write—these occur at different instructions. A write that has become visible to other threads can

be lost if its line remains dirty in a volatile cache at the time of a crash, leading to post-crash inconsistencies if a dependent write in another thread has made its way to NVM first.

Table 1 summarizes the concurrency control in various persistent dictionary data structures from the literature. Many of these use locks to delay the PoV until after data has safely persisted. (This typically requires a post-crash recovery routine to clean up held locks and partially completed operations.) For nonblocking structures whose operations linearize with atomic primitives like CAS, there is a risk that a store will become visible to other threads before it has persisted, leading to a dirty read anomaly [63]. Programmers typically handle this by persisting critical NVM *reads* in addition to writes [18, 25]. To simplify the design and implementation of persistent lock-free structures, Pavlovic et al. [45] and Wang et al. [54] have developed multi-word CAS (PMwCAS) operations that are both atomic and persistent. PMwCAS begins by initializing a descriptor, which stores the operation status and the addresses, old values, and new values of the target words. It then announces its intent by installing pointers to the descriptor in all of the target words, in canonical order. Finally, after using a single CAS to change the status of the descriptor from pending to committed, it replaces the pointers with updated values. If a conflict arises during the installation process (i.e., if a target address already holds a pointer to another descriptor), the thread helps complete the conflicting PMwCAS before proceeding (or retrying). Throughout this process, persist instructions ensure that descriptor changes are durably written to NVM, allowing a PMwCAS that is interrupted by a crash to either roll back or complete.

Persist operations are crucial for maintaining crash consistency. Unfortunately, given the latency of NVM, they can lengthen the critical path of operations by up to an order of magnitude [61]. They can also result in excessive NVM writes, exacerbating wear-out and stressing bandwidth limitations. Despite efforts to reduce persist instructions, they remain heavily used in state-of-the-art structures—especially lock-free structures that need to avoid dirty read anomalies. CCEH [36], a lock-based hash table, uses at least 3 persist instructions per insert. BzTree [2], a lock-free persistent B-tree built with PMwCAS, requires 11 persist instructions per insert.

Persistent cache (e.g., Intel eADR) eliminates the PoV–PoP discrepancy and the need for persist operations. It allows persistent data structures to use HTM for finer-grained concurrency control. This can be particularly helpful for structures with fat nodes (hash table buckets, tree leaves), in which multiple threads can access the same node safely as long as they operate on different entries [62, 63]. In lock-free data structures, eliminating the need for persist instructions significantly reduces overhead, improving scalability and performance. It also enables designs that choose when to write back data. Hot data can be kept in the cache to maximize hits and minimize NVM traffic. Small, cold writes can be buffered and combined into larger blocks in the cache, again to minimize traffic.

Table 1: Concurrency control in persistent mappings.

	Type	Data structure
Lock	Hash-based	Level hashing [68], Plush [51], CCEH [36], DASH [34]
	Tree-based	LB+Tree [32], FPTree [42], NVTree [58], DPTree [67]
	Trie-based	ROART [35], PACTree [29], ERT [53]
Lock-free	Hash-based	Clevel [12], SOFT [69]
	Tree-based	BzTree [2], μTree [11], Fast&Fair [24], NBTree [63]
	Trie-based	Heart [39]
HTM	Hash-based	Spash [62]

3 Combining HTM and Buffered Durability

We advocate using buffered durability to reconcile the incompatibility between HTM and NVM programming without requiring hardware modifications. HTM provides simple, low-overhead concurrency control (isolation and consistency); buffering allows persistence (atomicity and durability) to be added without compromising concurrency control. The remainder of this section describes the epoch system that transforms data structures into their buffered durable versions. The following section presents three case studies that leverage this epoch system.

Our epoch system builds on mechanisms implemented in Montage [55], with extensions to accommodate HTM and to improve performance. A background thread increments the value of a global clock every few milliseconds, dividing execution into epochs. At any given point in normal (crash-free) operation, these epochs can be categorized as

- 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 operations that have already begun can continue to execute, but in which new operations will no longer be created. Operations in this epoch will not survive a crash that occurs before $(e - 1) + 2 = e + 1$.
- *valid* epochs i, for $i \leq e - 2$. All operations in these epochs have completed and successfully persisted.

The epoch system API exposes methods to guarantee that each operation will occur within a single epoch, and that data will be properly buffered for delayed persistence. API methods are listed in Table 2.

Table 2: Epoch system API.

API	Description
beginOp()	Register the calling thread as active in the current epoch, and begin to track its NVM writes.
endOp()	Schedule tracked writes for persistence and disassociate the calling thread from any epoch.
abortOp()	Disassociate the calling thread from any epoch and discard its tracked writes.
pNew()	Allocate a memory block from NVM.
pSet()	Update a memory block.
pRetire()	Track a memory block for future reclamation.
pDelete()	Reclaim a memory block and return it to the NVM allocator.
pTrack()	Track a memory block in the current epoch.
b->getEpoch()	Return epoch in which memory block b was tracked.
b->setEpoch()	Set epoch of memory block b.

Epochs eliminate the quiescence issue present in snapshot and checkpointing systems, where all worker threads must be paused (or copy-on-write enabled) when synchronizing data to persistent storage. The epoch system advances from epoch e to $e + 1$ only after all updates from epoch $e - 2$ have safely persisted. While waiting for this advance, worker threads can continue their work in the in-flight epoch $e - 1$ or initiate new operations in epoch e. Internally, the epoch system maintains per-epoch announcement arrays and per-thread buffers to track operations and writes from worker threads.

Each thread brackets operations with `beginOp()` and `endOp()`. Each operation is confined to a single epoch. The code for `beginOp()` updates the caller's slot in the announcement array to signal that it has started a new operation in the current epoch. The code for `endOp()` clears this slot, informing the epoch system that tracking for this thread can stop. In between, threads use `pNew()` for NVM allocation and `pSet()` for NVM writes. NVM writes generated in a given epoch are tracked in the per-thread buffers and persisted together during crash-free execution—or discarded entirely in the event of a crash. More detail can be found in the original Montage paper [55].

Unfortunately, Montage is not HTM-compatible. To update a block in a BDL manner, Montage checks the epoch in which the block was last modified and, if this is not the operation's epoch, allocates a new NVM block and buffers the original block for reclamation in a future epoch. This out-of-place update ensures that operations interrupted by a crash do not corrupt the original data, allowing the system to recover to a consistent state. Because it requires persistence instructions, however, the `pNew()` method aborts any active HTM transaction. The `pDelete()` method, which reclaims NVM blocks, causes similar aborts. To support HTM, we introduced new API calls and modified the persistence strategy.

We use an HTM-based hash table to illustrate the modified strategy. Assume the hash table contains an array of buckets, each holding up to `BUCKET_SIZE` elements. For simplicity, we omit cases of full buckets and table resizing. As illustrated in Listing 1, a thread marks its operation with `beginOp()` (line 8) and `endOp()` (line 55). After finding the target key-value (KV) block in NVM, the thread must check the epoch number before performing an update (lines 20–32). An NVM block can be updated directly if it was created in the current epoch (line 29). Otherwise, if it was created in a later epoch by another thread, the current thread must abort with `OldSeeNewException`, as overwriting a newer block in an old epoch would violate BDL. The thread then restarts its operation in the current (newer) epoch (lines 39–41). If the block's epoch number is smaller than the operation's epoch, the thread replaces the block with a preallocated one and marks the original block for reclamation (lines 24–28).

NVM allocation must occur outside the HTM transaction to avoid aborts. The need for allocation, however, is only determined once the block's tracking epoch is retrieved within the transaction. Preallocation is required in the general case, but can be wasteful when the block can be updated in place. To avoid unnecessary overhead, each thread maintains a thread-local NVM block, `new_blk`, and only allocates a new one when `new_blk` has been used (lines 28 and 36). In Montage [55], the epoch system tags each NVM block with an epoch number at allocation time. Because we may not use a preallocated block for an arbitrary amount of time, we modify the epoch system to assign new blocks an invalid epoch number. We update this number with a new API method (`setEpoch()`) when

```
1  EpochSys* esys;
2  Mutex global_lock;
3  thread_local void* new_blk, *retire_blk, *persist_blk;
4
5  Insert(k_type k, v_type v) {
6    auto k_hash = hash(k);
7  retry_regist:
8    uint64_t op_epoch = esys->beginOp();
9    // skip allocation if a block is already available
10   if (!new_blk) new_blk = esys->pNew();
11   // initialize memory block
12   new (new_blk) KVPair(k, v, INVALID_EPOCH);
13 retry_txn:
14   int status = _xbegin();
15   if (status == _XBEGIN_STARTED) {
16     if (global_lock.locked()) _xabort(Locked);
17     new_blk->setEpoch(op_epoch);
18     auto bucket = hash_table->get_bucket(k_hash);
19     for (int i = 0; i < BUCKET_SIZE; ++i) {
20       if (bucket[i]->get_key() == k) {  //found, update
21         if (bucket[i]->getEpoch() > op_epoch) {
22           //do not update block modified in later epoch
23           _xabort(OldSeeNewException);
24         } else if (bucket[i]->getEpoch() < op_epoch) {
25           retire_blk = bucket[i]; // reclaim old block
26           bucket[i] = new_blk; // add new block
27           persist_blk = new_blk; // track new block
28           new_blk = nullptr; // preallocated block used
29         } else bucket[i]->pSet(k, v); //in-place update
30         _xend();
31         goto op_done;
32       }
33     }
34     bucket->insert(new_blk); // not found, insert
35     persist_blk = new_blk;
36     new_blk = nullptr;
37     _xend();
38   } else {
39     if (status == OldSeeNewException) {
40       esys->abortOp(); // abort to restart in new epoch
41       goto retry_regist;
42     } else if (status == Locked) {
43       while (global_lock.locked()) {} // spin
44       goto retry_txn;
45     }
46     global_lock.acquire();
47     /* fallback path similar to lines 20-36 */
48     global_lock.release();
49   }
50 op_done:
51   if (retire_blk) esys->pRetire(retire_blk);
52   if (persist_blk) esys->pTrack(persist_blk);
53   retire_blk = nullptr;
54   persist_blk = nullptr;
55   esys->endOp();
56 }
```

Listing 1: BDL HTM strategy applied to the `Insert` operation of a simple hash table. API calls are shown in blue. The `_xbegin`, `_xend`, and `_xabort` calls are Intel HTM intrinsics.

we're about to use the block (line 17), and reset it to invalid when starting a new operation (line 12). In all cases, memory persistence and reclamation are delayed until after an HTM transaction has committed (lines 51–52). During recovery, NVM blocks with an invalid epoch number are simply reclaimed.

4 Case Studies

Our case studies cover three cutting-edge dictionary structures: a vEB tree (transient, HTM-synchronized, with doubly logarithmic complexity), a skiplist (persistent, lock-free, with logarithmic complexity), and a hash table (persistent, HTM-synchronized, customized for eADR, with $O(1)$ complexity). In Sec. 4.1, we convert the vEB tree to a buffered durable version, evaluate its persistence overhead, and compare its performance to that of persistent alternatives. In Sec. 4.2, we relax the persistence requirement of the persistent skiplist and use HTM to optimize its concurrency control, reducing synchronization overhead, persistence overhead, and programming complexity. In Sec. 4.3, we modify the eADR-based hash table to be buffered durable, preserving its HTM compatibility and maintaining NVM optimizations for systems with volatile caches. In all cases, our BDL structures store only critical data in NVM, rebuilding indices as needed during post-crash recovery.

All experiments were conducted on a 40-core machine (80 hardware threads) consisting of two Intel Xeon Gold 6230 processors with 20 cores each. Each processor is equipped with six channels of 128 GB Intel Optane DC persistent memory configured in AppDirect mode. The YCSB benchmark is employed to generate workloads with uniform and Zipfian distributions, with the Zipfian constant set (unless otherwise noted) to the default value of 0.99. For the sake of consistency with previous studies in the literature, we evaluate the data structures using 8-byte keys and 8-byte values. Data structures were prefilled with pairs representing half of the key space, and write operations employed a 50/50 mix of inserts and removes to keep the sizes of structures stable. For buffered durable structures, the epoch length was set (again, unless otherwise noted) to 50 ms. Hyper-threading was disabled to avoid capacity-related HTM aborts.

4.1 Porting HTM-based Structures to NVM

Sets and dictionaries (maps) are among the most important (sequential and concurrent) data abstractions. While hash tables offer O(1) complexity for most operations, their lack of ordering precludes efficient range, successor and predecessor queries. B-trees and tries provide logarithmic complexity. A few structures—notably van Emde Boas (vEB) trees [50]—are doubly logarithmic.

Briefly, a vEB tree for a universe of U elements has a root node containing (1) an array of pointers to up to $\sqrt{U}$ subtrees, each for a sub-universe of size $\sqrt{U}$, and (2) one additional such subtree—the *summary*, that indicates which of the main subtrees exist (i.e., are nonempty). The purpose of the summary—and of its analogues at lower levels of the tree—is to support rapid location of successor keys, thereby enabling fast range queries. When keys are uniformly distributed across the universe set, space consumption can be $O(U)$—a serious disadvantage when the keys are also sparse. In practice, however, with non-uniform key distribution, many subtrees will typically be missing.

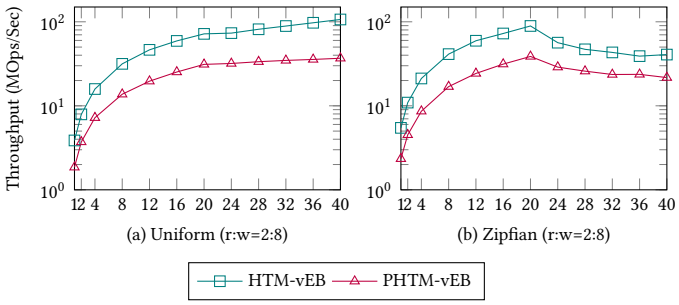

Figure 1: Throughput of transient and buffered durable vEB trees (write-heavy workload). The X-axis represents the number of threads. The universe size of keys is 2^{26}. Zipfian constant is set to 0.99.

Implementing a concurrent vEB tree is extremely challenging, and previous attempts have tended to compromise complexity, safety, or both. To the best of our knowledge, the only publicly available implementation that supports fully linearizable concurrent accesses while preserving time complexity is due to Khalaji et al. [28]. Their tree (referred to here as **HTM-vEB**) protects each operation with an HTM transaction; sadly, this strategy precludes the straightforward addition of durable linearizability.

Following the strategy of Listing 1, we enhance HTM-vEB with buffered durability instead, creating a **PHTM-vEB** tree. In the transient vEB tree, only values are stored in leaf nodes: keys are implicit in the location in the tree. For the sake of speed and simplicity, we keep the PHTM-vEB index structure in DRAM, with pointers at the leaves that reference KV pairs in NVM. During crash-free execution, a PHTM-vEB lookup traverses the tree as HTM-vEB does and then retrieves a value from NVM. After a system crash, PHTM-vEB reconstructs the tree by scanning KV pairs.

To assess the overhead introduced by buffered durability, we compare the throughput of HTM-vEB and PHTM-vEB (Fig. 1). While the latter is slower, the difference seems quite reasonable: Optane memory has roughly one third the bandwidth of DRAM, 3× the latency for reads, and 10× the latency for writes, but on a single socket (thread count less than 20), PHTM-vEB remains within roughly a factor of 2 of HTM-vEB bandwidth. On two sockets, where Optane is known to perform more poorly than DRAM [46], PHTM-vEB is within a factor of 3 of HTM-vEB for uniform write-heavy workloads (Fig. 1(a)) and still within a factor of 2 for Zipfian or read-heavy (not shown) workloads.

The primary source of overhead in PHTM-vEB turns out to be memory management for KV pairs. When a pair is inserted in the tree, a new NVM block must be allocated if the key was not previously present or was last given a value in a prior epoch. NVM reclamation occurs when a KV pair is deleted or replaced by a newer version in a subsequent epoch. The Ralloc persistent allocator [8], used in our experiments, is fast and nonblocking, but its work is avoided entirely in HTM-vEB, which keeps values in the leaves of the main tree. The overhead of the epoch system itself is offloaded to a background thread in PHTM-vEB, and has relatively minor impact on the critical path.

In a larger context, we can see the value of doubly logarithmic time—and the success of PHTM-vEB—by comparing to state-of-the-art persistent search trees. For this we consider the **LB+Tree** of Liu et al. [32] and the **OCC-ABTree** and **Elim-ABTree** of Srivastava and Brown [48]. The LB+Tree is a persistent B+ tree customized for NVM. It employs per-node locks for concurrency control, keeps inner nodes in DRAM for fast tree traversal, and puts leaf nodes in NVM. Much as in PHTM-vEB, the internal tree is rebuilt by scanning the leaf layer in the wake of a crash. The (a, b) trees are B-tree variants that allow between a and b keys per node ($a \leq b/2$). The OCC-ABTree uses fine-grained versioned locks for update atomicity, and version-based validation (optimistic concurrency control) for search correctness. The Elim-ABTree, optimized for skewed workloads, introduces *publishing elimination*, which reorders and eliminates concurrent inserts and deletes to reduce both the total number of operations and the number of writes to NVM. Both OCC-ABTree and Elim-ABTree are fully persistent. Despite these many optimizations, our throughput results, shown in Fig. 3, demonstrate that (at least for workloads that can tolerate the space consumption) vEB trees are a major win: by combining HTM and buffered durability, PHTM-vEB outperforms LB+Tree by 1.2–2.8×, and Elim-ABTree and OCC-ABTree by 1.6–4×.

The vEB tree delivers excellent performance at the cost of significantly higher space consumption. We evaluated the DRAM and NVM usage of each tree, initialized with half the keys (uniformly distributed) in a universe of size 2^{26}. Results appear in table 3. PHTM-vEB and volatile HTM-vEB have the same DRAM consumption, which is 16× higher than that of LB+Tree (with only the inner tree stored in DRAM). Fully persistent trees—Elim-Tree and OCC-Tree—on the other hand, use no DRAM. In addition, PHTM-vEB incurs higher NVM consumption because it stores not only the KV pairs but also their copies from previous epochs. The other trees maintain only a single record per KV pair. Furthermore, the KV pairs in PHTM-vEB include extra metadata required for post-crash recovery, such as epoch number and block type (e.g., allocated but not used, deleted). The NVM space consumption of PHTM-vEB is dependent to some degree on the epoch length; we discuss this in Section 5.1.

As buffering increases memory footprint, one could imagine this leading to higher cache eviction rates and thus transaction aborts. One could also imagine background writes-back from the epoch system aborting transactions that happened to be using the written-back lines. To investigate these possibilities, we measured the transaction abort rate (Fig. 2) for varying thread counts and workloads (with the epoch length remaining at 50 ms). Despite the increased tree size and background cache line evictions, we see no significant difference in transaction abort rates between HTM-vEB and PHTM-vEB. Moreover, as Fig. 1 and Fig. 2 show,

Table 3: Space consumption (in MiB) of search trees with 2^{25} keys, drawn from a universe of 2^{26}.

Tree	HTM-vEB	PHTM-vEB	LB+Tree	Elim-Tree	OCC-Tree
DRAM	1073	1073	67	0	0
NVM	0	1257	698	774	779

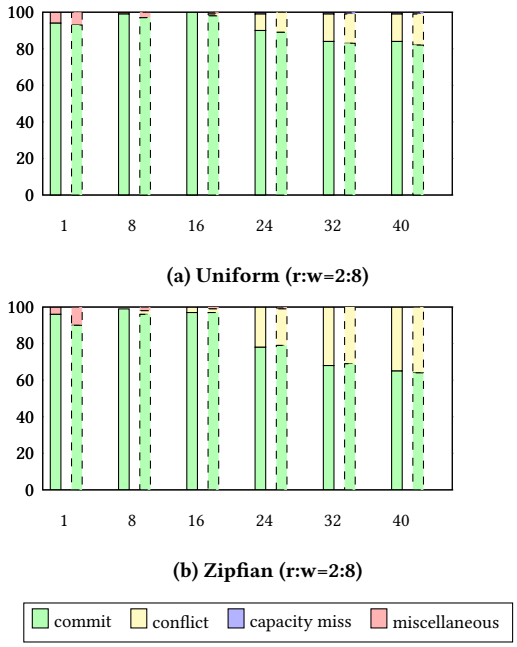

Figure 2: HTM commit and abort rates (in percentage) for HTM-vEB and PHTM-vEB trees. Key space size is 2^{26}. The X-axis shows the number of threads. In each figure, the left bars (solid border) correspond to HTM-vEB, and right bars (dashed border) correspond to PHTM-vEB.

PHTM-vEB scales nearly as well as its transient cousin. Conflict-driven aborts increase with the number of threads, but remain below 15% for uniform workloads and 35% for Zipfian workloads. Mysteriously, at low thread counts, up to half of our transactions initially aborted with a reported cause of "incompatible memory type" (ABORTED_MEMTYPE). Similar aborts have been reported in previous work [21]. Intel documentation attributes such aborts to the use of uncacheable memory, but there is none in either HTM-vEB or PHTM-vEB. Interestingly, the problem did not appear when we repeated experiments on a separate, single-processor machine (10-core Xeon Gold 5215; one 128 GB Optane module in AppDirect mode). As a work-around, we added a non-transactional "pre-walk" of the tree after each MEMTYPE abort before retrying the transaction; this reduced the single-core abort rate to about 5% (red bars in Fig. 2).

4.2 Optimizing Concurrency Control in Already-persistent Data Structures

Durably linearizable data structures must persist critical updates immediately to maintain a consistent execution history in NVM. This requirement precludes the use of HTM and introduces significant overhead. The overhead is particularly pronounced in persistent lock-free structures, which often employ complex protocols to update multiple locations atomically and persistently. Consider, for example, the durably linearizable lock-free **DL-Skiplist** of Wang et al. [54], which employs a persistent multi-word compare-and-swap (PMwCAS). Each PMwCAS operation starts by allocating a

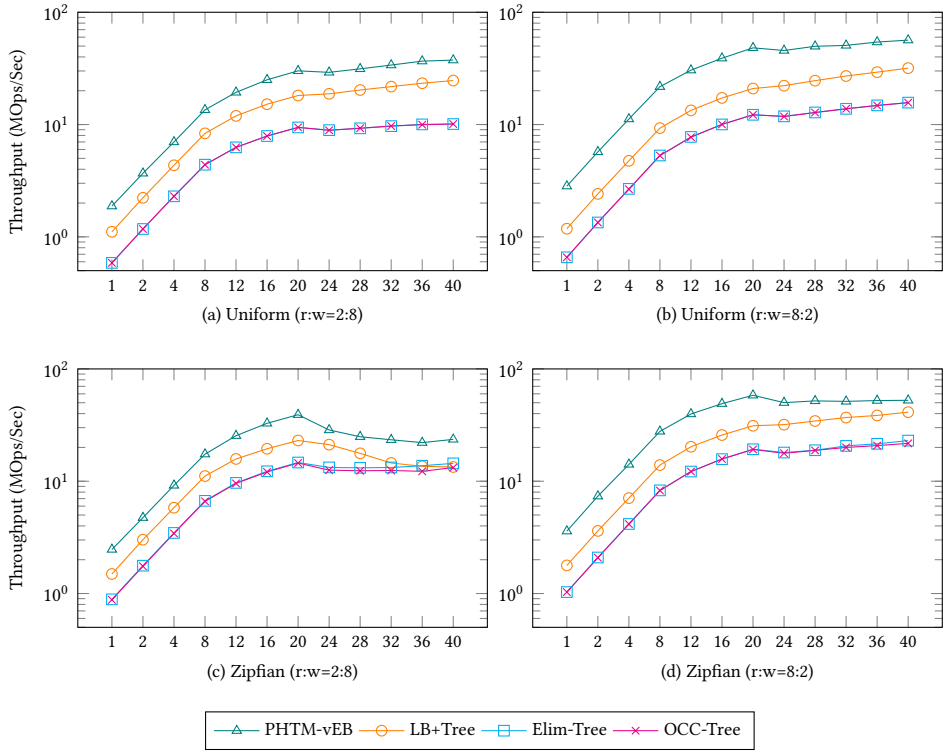

Figure 3: Throughput of persistent trees with uniform (top) or Zipfian (bottom) key distributions, and write-heavy (left) or read-heavy (right) workloads. X is the number of threads.

descriptor in NVM, which it must persist immediately to avoid a memory leak. The operation then initializes the descriptor with a set of memory addresses, together with old and new values, which need to be updated atomically; these values also require immediate persistence. To effect its updates, PMwCAS then swaps (installs) the descriptor into each of the target addresses, persisting each one in turn. After performing (and persisting) a single CAS to change the descriptor from "in progress" to "committed," PMwCAS then swaps new values in place of the descriptor address in each of the modified words; these changes must also persist before the descriptor can be reclaimed. Along the way, PMwCAS may issue extra persist operations to assist pending PMwCAS operations from other threads. When all is done, PMwCAS deallocates the descriptor and persists the deallocation. This entire process incurs not only significant persistence overhead but also additional maintenance costs due to memory management.

To assess the cost of PMwCAS, we conducted a simple experiment in which threads repeatedly update 2, 4, or 8 randomly selected locations within an array of one million cache line-aligned slots in NVM. As a baseline, the Mw-WR (multi-word write) bars in Fig. 4 show the throughput of performing the updates without any synchronization or persistence. The HTM-MwCAS bars reflect updates performed inside an HTM transaction, again without persistence. These incur only modest cost relative to Mw-WR. MwCAS—a transient version of PMwCAS—omits persist instructions; it is slower than HTM-MwCAS because it relies on the descriptor-based protocol, which has much more overhead than HTM. PMwCAS incurs a performance drop of over 10×, relative to even MwCAS, due to the cost of persistence instructions and cache miss penalties (all write-back instructions on our machine invalidate target lines). These results suggest that buffered durability, together with HTM, might significantly improve the performance of PMwCAS-based structures.

To pursue this possibility, we modified the DL-Skiplist and ran experiments with 1 million keys for 30 s (as before, keys and values each comprise 8 bytes). Results appear in Fig. 5. The original algorithm is fully persistent, in the sense that all nodes reside in NVM. It's also strictly DL: critical updates are persisted before returning to the caller. By contrast, our BDL-Skiplist is buffered durably linearizable, enabling the use of HTM. It maintains its indexing data in DRAM and persists only KV pairs in NVM. Across a range of thread counts, BD-Skiplist sustains about 3× the throughput of DL-Skiplist. We attribute this improvement to three factors. First, by keeping the skiplist towers in DRAM, we obtain substantially faster searches, at least when lists do not fit in the cache. Second, we reduce persistence overhead by writing back KV pairs periodically in the background, rather than on the critical path. Third, we reduce the overhead of MwCAS by using HTM.

By building additional (nonsensical) algorithms, shown as extra bars in Fig. 5, we can tease apart the impact of these factors. P-Skiplist-no-flush removes persist instructions from DL-Skiplist

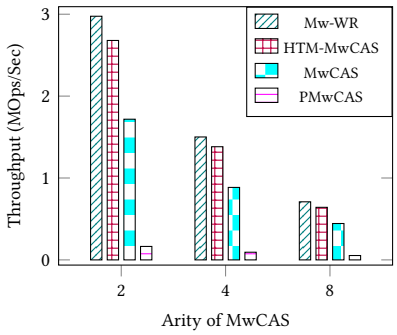

Figure 4: Throughput of MwCAS implementations with a single thread. X-axis indicates the number of memory locations updated atomically.

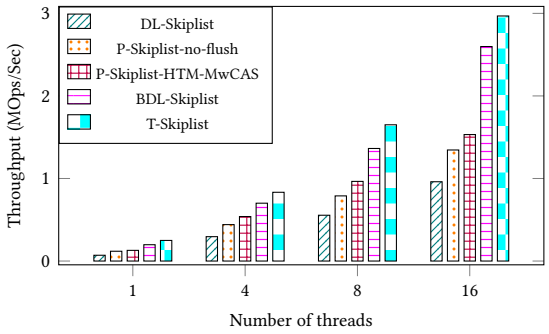

Figure 5: Throughput of persistent lock-free skiplist with uniform workload (read:write = 2:8).

(abandoning durable linearizability), achieving ~1.7× the throughput of DL-Skiplist. P-Skiplist-HTM-MCAS uses HTM to optimize MwCAS, achieving an additional improvement of ~10%. (Note that while both P-Skiplist-no-flush and P-Skiplist-HTM-MwCAS store the entire data structure in NVM, neither is crash consistent.) As a final data point, T-Skiplist, synchronized with MwCAS, stores the entire data structure in DRAM; it outperforms BDL-Skiplist by only ~20%. Given the small transaction footprint and narrow temporal window of vulnerability, optimizing MwCAS with HTM results in very few transaction aborts. In practice, this means that our skiplist preserves most of the advantages (preemption tolerance, low tail latency) of the nonblocking algorithm. Buffering does, however, require extra space in NVM, to store KV pairs and stale copies; we explore this further in Section 5.1.

4.3 Porting eADR-based Structures Back to ADR

Persistent cache (Intel eADR) simplifies persistent programming by ensuring crash consistency without explicit writes-back, and by enabling the use of HTM. At the same time, it allows write-back instructions to be used to optimize cache and memory system performance, potentially increasing throughput and reducing NVM wear-out. In this section, we consider the optimizations employed by **Spash** [62], a state-of-the art persistent hash table designed for eADR machines. Using buffered durability, we show how many of the optimizations can be back-ported to machines with volatile (ADR) caches.

Given the freedom to use writes-back for performance (rather than correctness), Spash tracks its access pattern in a lightweight structure in DRAM, allowing it to distinguish hot and cold KV pairs. It proactively flushes the cold pairs to NVM, freeing space for hot data and improving the cache hit rate. To use NVM bandwidth as efficiently as possible, Spash flushes data only when their size exceeds the NVM-internal access granularity (one "XPLine"—256 B for first-generation Optane). Cold pairs that are smaller than this size are appended to a 256 B thread-local memory chunk, with an indirection pointer placed in the hash-table bucket. The actual data can then be flushed at XPLine granularity, saving bandwidth throughout the memory hierarchy [56] and avoiding write amplification.

Spash employs HTM for low-overhead concurrency control, using a fine-grained structure that avoids transaction aborts due to capacity misses. Specifically, the top level of the hash table consists of a directory containing pointers to *segments*, each of which contains multiple buckets. The segment size is always a multiple of the XPLine size, and buckets are always a multiple of the cache line size. Read and write requests typically access only a few directory entries and segments. When the load factor gets too high, the directory is doubled in size and segments are moved from the old directory to the new. The directory doubling is protected by a global lock but happens quickly. Segments are moved in the background using HTM transactions. Worker threads assist in this process whenever they try to use a directory entry whose segments have not yet been moved.

To accommodate ADR platforms without losing Spash's optimizations and HTM compatibility, we integrate it with our epoch system, resulting in **BD-Spash**. If eADR is detected, the epoch system automatically disables itself, allowing BD-Spash to seamlessly operate on both ADR and eADR machines. BD-Spash stores its directory and buckets in DRAM. Buckets in turn contain pointers to KV pairs in NVM. BD-Spash uses the hotspot detector to decide whether an NVM block should be persisted immediately or deferred: large cold data is persisted at once to optimize cache usage and NVM bandwidth, while small cold data and hot data are tracked by the epoch system for delayed persistence. BD-Spash does not coalesce small cold writes for two reasons. First, compacting these writes in thread-local chunks adds overhead to store indirection pointers in appropriate buckets, and to follow these pointers when reading the data. Compacting also introduces extra memory management costs since stale chunks must be reclaimed periodically. Second, by delaying writes to epoch boundaries, the epoch system naturally coalesces many writes to adjacent KV pairs, even when they occur up to an epoch apart in time.

We compared BD-Spash against Spash and two other persistent hash tables: CCEH [36] and Plush [51]. CCEH (cache-line–conscious extendible hashing) is a fully persistent hash table that guarantees failure atomicity without explicit logging. Like Spash, it employs a directory containing pointers to multi-bucket segments. Updates are synchronized with reader-writer locks; searches are lock free. Each operation performs enough writes-back and fences that post-crash recovery can roll partially completed operations

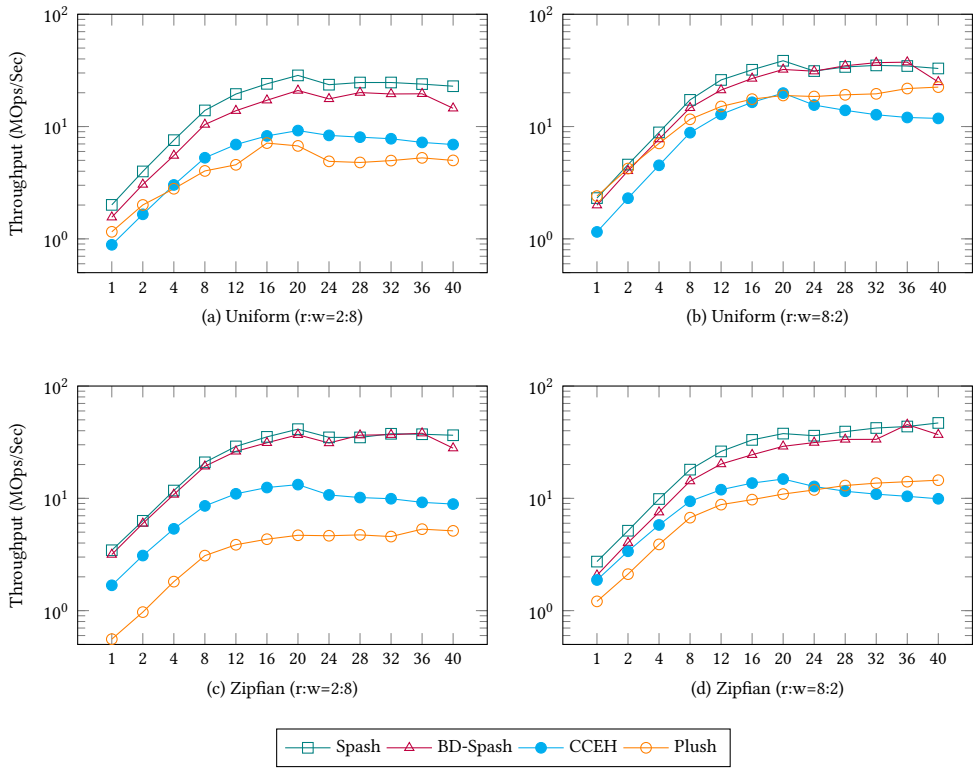

Figure 6: Throughput of persistent hash tables with Uniform (top) or Zipfian (bottom) workloads, characterizing by write-heavy (left) or read-heavy (right). X-axis is the number of threads.

forward or backward and release all locks. Plush adopts a log-structured, layered approach, somewhat akin to an LSM tree [43], with the root level in DRAM and the rest in NVM. Each level is a multiple of the previous level in size. When buckets overflow, data is re-hashed and appended to buckets in the next deeper level. Lookup operations probe a filter at each level. To ensure failure atomicity, Plush persists log entries before a write operation returns.

Performance results appear in Fig. 6. Relative to Spash, the other hash tables pay higher persistence costs for memory management and NVM writes, on the assumption that caches are transient. BD-Spash minimizes this overhead through batched, background persistence performed by its epoch system, essentially matching the throughput of Spash on the write-heavy Zipfian workload and coming close in the other three cases. CCEH and Plush are significantly slower than BD-Spash, due largely to the overhead of strict DL. CCEH's log-free design is optimized for writes, allowing it to outperform Plush for both uniform and Zipfian write-heavy workloads. The advantage disappears in the read-heavy case, particularly when threads are spread across the sockets of the machine. Despite its use of DRAM in the initial layer of the structure, Plush suffers from expensive logging on the critical path in every subsequent layer; with a skewed workload, this logging induces contention that leads to the lower Plush performance in Fig. 6(c). If logging is disabled (results not shown), Plush throughput improves by 1.8×, but this of course forfeits crash consistency.

5 Discussion

The API of Table 2 makes it straightforward to adapt volatile or persistent data structures to buffered durability. Certain guidelines should be followed, however, to guarantee BDL and to fully leverage HTM. Beyond avoiding instructions conflicting with HTM (e.g., non-temporal stores and flush), programmers should also avoid any operations that might implicitly invoke these instructions. NVM allocators, for example, typically rely on flush instructions to avoid permanent memory leaks, and compiler intrinsics such as store_stream use non-temporal instructions. Some standard library routines (e.g., memcpy and memset in glibc) may internally switch to non-temporal accesses under certain conditions (large copy sizes, alignment, certain CPU models). If a program experiences unexpectedly high abort rates, profiling tools like Perf [15] can help identify the culprit instructions.

Because NVM allocation in our strategy is performed outside of transactions and manually managed, extra care is needed to preserve BDL. First, any preallocated but unused NVM block must have its epoch set to invalid. If an operation sets the block's epoch but is later interrupted, that epoch must be reset to invalid upon restart. Second, the epoch number should be assigned *before* the operation's linearization point; otherwise, other threads may see the operation without being able to determine whether it belongs to the same epoch or a newer one, making it impossible to decide if they should abort or continue. Note that in the event of a crash,

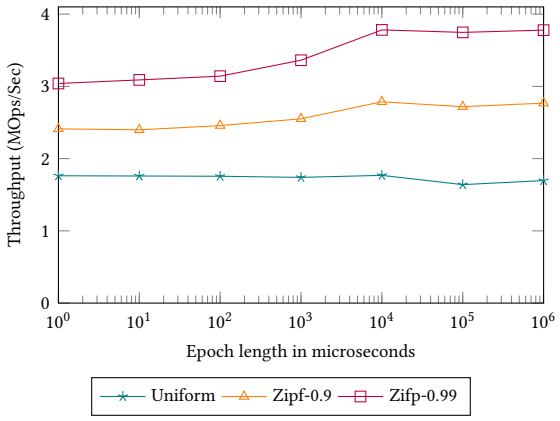

Figure 7: Throughput of single-threaded PHTM-vEB with different epoch lengths and workload distributions. The workload size is 2^{22}, with 80% writes and 20% reads.

any block that has been assigned an epoch in a not-yet-completed operation will be in an epoch that is discarded in recovery, allowing the block to be reclaimed.

5.1 Epoch length

Epoch length determines the frequency of data persistence, potentially influencing system performance via (1) transaction abort rate, (2) cache utilization, (3) NVM space consumption, and (4) NVM bandwidth usage. Background flushes can invalidate active transactions whose read/write sets include evicted data. They can also reduce cache efficiency and increase NVM bandwidth consumption by forcing data out from cache to NVM, necessitating reloads when data are accessed again. Intuitively, increasing the epoch length might be expected to reduce transaction aborts and cache misses caused by background flushes, and to save NVM bandwidth by serving reads and writes directly from the cache, thereby improving performance. At the same time, longer epochs might exacerbate memory usage, depending on workload characteristics. In BDL, in-place updates to data from prior epochs are prohibited; thus, any update to data from epoch x in a later epoch y ($y > x$) requires allocating a new block, leaving the old block intact for crash recovery. The old block cannot be reclaimed until epoch $y + 2$, when the epoch system ensures that the new block has persisted. Consequently, longer epochs may lead to more out-of-place updates when workloads exhibit poor locality, leading to increased memory consumption, poorer cache utilization, and higher memory management overhead, all of which may degrade performance. We consider these tradeoffs in more detail below.

Transaction abort rate. Our PHTM-vEB tree is particularly susceptible to background flushes, because full operations are enclosed in HTM transactions, incurring a large footprint in both time and space. To measure sensitivity to epoch length, we conducted experiments with 1–16 threads, epoch lengths from 1 μs all the way to 10 s (a window of potential data loss that most users would consider unacceptable), a data set of 1–67 M records, and both uniform and Zipfian workloads. In all cases, background flushes caused fewer than 2% of transactions to abort. We conclude that aborts induced by the epoch system are not a significant concern.

Cache utilization. Caches mitigate the latency of memory, significantly enhancing performance when data accesses exhibit strong temporal locality. Flush instructions, however, can expose this latency on the critical path by invalidating frequently reused cache lines. Our experiments indicate that for uniform workloads with working sets exceeding cache capacity, background data persistence imposes minimal overhead, as random accesses derive little benefit from caching. For highly skewed workloads, however, when most accesses target a small range of data that can be cached, background persistence incurs noticeable overhead. As shown in Fig. 7, increasing the epoch length from 1 μ to 10 ms yields performance improvements of 16.7% and 26.7% for Zipfian workloads with constants of 0.9 and 0.99, respectively (the 0.99 constant indicating greater skew). However, further increases in epoch length yield diminishing returns due to increased memory usage, leading to higher cache eviction rates and greater memory management overhead (more on this below).

NVM Space consumption. In a BDL data structure, up to three copies of each record may coexist: a fully persisted copy (from at least two epochs ago), a copy pending persistence (updated in the previous epoch), and an actively modified copy (from the current epoch). Consequently, the overall NVM footprint of a BDL structure may depend on both the epoch length and the frequency of data accesses and updates. For a fixed epoch length, uniform workloads tend to incur higher space consumption because data is accessed with equal probability, leading to more out-of-place updates and fewer in-place modifications. Likewise, for a given workload distribution, longer epochs result in increased space usage because stale data blocks are retained longer, delaying memory recycling and expanding the memory footprint. We illustrate these trends using the PHTM-vEB structure, measuring NVM usage with epoch lengths ranging from 1 μs to 10 s under both uniform and Zipfian distributions. As shown in Fig. 8, uniform workloads have higher space consumption due to frequent out-of-place updates, while longer epochs exacerbate space usage primarily because buffered NVM allocations and deletions remain unconfirmed by the epoch system for an extended period. At the same time, with the exception of the extreme 10 s case, the variations in Fig. 8 are relatively modest, suggesting that BDL structures can be expected to display reasonable performance and space consumption across a wide range of use cases and configurations.

NVM bandwidth usage. Data is persisted to NVM in two ways: controlled epoch-based flushes and uncontrolled cache eviction. Epoch flushes are crucial for crash consistency, and represent a predictable trade-off between write frequency and size: shorter epochs buffer less data, leading to smaller but more frequent writes; longer epochs buffer more data for larger, less frequent writes. In contrast, cache evictions introduce unnecessary access amplification. Read amplification occurs when evicted data is later accessed within the same epoch, forcing a reload from NVM. Write amplification occurs when data is evicted before the epoch ends, preventing multiple updates to the same location from being coalesced in the cache and resulting in redundant writes to NVM. To mitigate amplification,

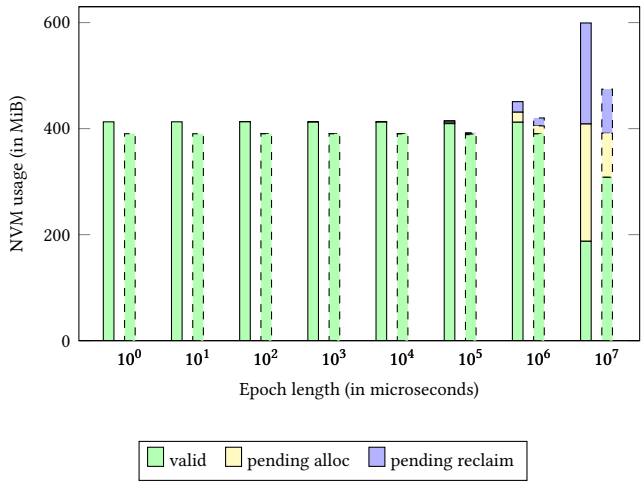

Figure 8: NVM space consumption of PHTM-vEB under uniform (left bars with solid border) and zipfian workloads (right bars with dashed border). Key space is 2^{24}, with a single thread performing 50% insert and 50% remove operations.

the volume of data buffered within an epoch should not exceed the available cache capacity. This volume can be modeled [17] as a function of throughput and epoch length, a relationship that holds regardless of the data structure type. Our evaluation shows that even with the highest-throughput data structures (PHTM-vEB on 20 threads), an epoch length of 100 ms generates 43 MiB of buffered data. This amount fits within the 48 MiB of combined L1, L2 and L3 cache on our test machine.

Overall, our experiments indicate that BDL data structures offer stable performance with only a modest increase in NVM space for duplicate data (old copies and pending deletions), over a wide range of epoch lengths. Thus, performance is robust with respect to epoch length, and fine-tuning this parameter is not essential. In practice, epoch lengths between 10 ms and 100 ms yield excellent performance, efficient cache utilization, and only a moderate increase in NVM usage.

5.2 Recovery

BDL data structures recover to a pre-crash state by scanning the NVM heap that stores data blocks. The recovery procedure first retrieves the global epoch number (persisted at each epoch transition) and then compares it with the epoch number of each data block. Specifically, data blocks that either (1) are valid (e.g., marked as ALLOCATED) and are at least two epochs older than the global epoch, or (2) were deleted (marked as DELETED with a valid delete epoch number) within the past two epochs but whose deletion has not been persisted by the epoch system, are recovered; all other blocks are reclaimed by the allocator.

Recovery of a BDL data structure is rapid because sequential NVM scanning delivers high bandwidth, and the reconstruction of indexing structures is performed in DRAM, which also offers high bandwidth. For example, using a single thread, scanning a 500 MiB heap containing 10 million records takes about 163 ms, while rebuilding the indexing structures requires roughly 2.9 s for the PHTM-vEB tree, 37.3 seconds for the BDL-Skiplist, and 5.4 s for BD-Spash. With 20 threads, heap scanning is reduced to under 10 ms, and the reconstruction times drop to 0.2 s, 4.6 s, and 0.39 s, respectively. When scaling to a 5 GB data set containing 100 million records, even if with the slowest BDL-Skiplist, recovery only takes 50 s when using 20 threads.

6 Conclusion

Designing efficient concurrent data structures is inherently challenging, and HTM has emerged as a promising means of combining simplicity and performance. Meanwhile, nonvolatile memory opens the door to durable data structures, allowing in-memory structures to survive system crashes consistently. In a reversal of conventional wisdom, our work demonstrates that HTM and NVM can be combined, in a straightforward, general, and efficient way, using *buffered durable linearizability* (BDL), achieving both atomicity and durability at reasonable levels of programmer effort. More specifically, we have used BDL to (1) add persistence to HTM-based volatile data structures, (2) reduce the overhead of concurrency control in already persistent structures, and (3) allow structures designed for persistent caches to be back-ported to systems with transient cache while preserving performance optimizations. As data set sizes continue to increase and as technologies like CXL make effectively nonvolatile data-center memory ubiquitous, persistent indexing structures are likely to be ever more important. Buffered durability, combined with HTM, can make these structures more efficient and easier to synchronize.

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] Ahmed Abulila, Izzat El Hajj, Myoungsoo Jung, and Nam Sung Kim. 2022. ASAP: Architecture Support for Asynchronous Persistence. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 306–319.

[2] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. 2018. Bztree: A High-Performance Latch-Free Range Index For Non-Volatile Memory. *Proceedings of the VLDB Endowment* 11, 5 (2018), 553–565.

[3] Gal Assa, Andreia Correia, Pedro Ramalhete, Valerio Schiavoni, and Pascal Felber. 2023. Tl4x: Buffered Durable Transactions on Disk As Fast As In Memory. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 245–259.

[4] Lawrence Benson, Leon Papke, and Tilmann Rabl. 2022. PerMA-bench: Benchmarking Persistent Memory Access. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2463–2476.

[5] Nathan G Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. 2010. A Practical Concurrent Binary Search Tree. *ACM Sigplan Notices* 45, 5 (2010), 257–268.

[6] Trevor Brown. 2017. A Template for Implementing Fast Lock-free Trees Using HTM. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*. 293–302.

[7] Miao Cai, Chance C Coats, and Jian Huang. 2020. Hoop: Efficient Hardware-assisted Out-of-place Update for Non-volatile Memory. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 584–596.

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

[9] Zixian Cai, Stephen M Blackburn, and Michael D Bond. 2021. Understanding And Utilizing Hardware Transactional Memory Capacity. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management*. 1–14.

[10] Daniel Castro, Paolo Romano, and João Barreto. 2019. Hardware Transactional Memory Meets Memory Persistency. *J. Parallel and Distrib. Comput.* 130 (2019), 63–79.

[11] 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.

[12] Zhangyu Chen, Yu Hua, Bo Ding, and Pengfei Zuo. 2020. Lock-free Concurrent Level Hashing for Persistent Memory. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 799–812. https://www.usenix.org/conference/atc20/presentation/chen

[13] CXL Consortium. 2025. Compute Express Link. https://computeexpresslink.org/.

[14] Tudor David, Aleksandar Dragojevic, Rachid Guerraoui, and Igor Zablotchi. 2018. {Log-Free} Concurrent Data Structures. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 373–386.

[15] Arnaldo Carvalho De Melo. 2010. The New Linux 'perf' Tools. In *Slides from Linux Kongress*, Vol. 18. 1–42.

[16] Dana Drachsler, Martin Vechev, and Eran Yahav. 2014. Practical Concurrent Binary Search Trees Via Logical Ordering. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 343–356.

[17] Mingzhe Du and Michael L. Scott. 2024. Buffered Persistence in B+ Trees. *Proceedings of the ACM on Management of Data* 2, 6 (Dec. 2024). Presented at the ACM SIGMOD/PODS Intl. Conf. on Management of Data, Berlin, Germany, June 2025.

[18] 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 *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 377–392.

[19] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. 2018. A Persistent Lock-free Queue for Non-volatile Memory. *ACM SIGPLAN Notices* 53, 1 (2018), 28–40.

[20] Kaan Genç, Michael D Bond, and Guoqing Harry Xu. 2020. Crafty: Efficient, HTM-compatible Persistent Transactions. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 59–74.

[21] Bhavishya Goel, Ruben Titos-Gil, Anurag Negi, Sally A McKee, and Per Stenstrom. 2014. Performance and Energy Analysis of The Restricted Transactional Memory Implementation on Haswell. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 615–624.

[22] 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.

[23] Maurice P Herlihy and Jeannette M Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 3 (1990), 463–492.

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

[25] Joseph Izraelevitz, Hammurabi Mendes, and Michael L Scott. 2016. Linearizability of Persistent Memory Objects Under A Full-system-crash Failure Model. In *Distributed Computing: 30th International Symposium, DISC 2016, Paris, France, September 27-29, 2016. Proceedings 30*. Springer, 313–327.

[26] Jungi Jeong, Jaewan Hong, Seungryoul Maeng, Changhee Jung, and Youngjin Kwon. 2020. Unbounded Hardware Transactional Memory for a Hybrid DRAM/NVM Memory System. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 525–538.

[27] Jungi Jeong, Jianping Zeng, and Changhee Jung. 2022. Capri: Compiler and Architecture Support for Whole-system Persistence. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*. 71–83.

[28] Mohammad Khalaji, Trevor Brown, Khuzaima Daudjee, and Vitaly Aksenov. 2024. Practical Hardware Transactional vEB Trees. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 215–228.

[29] Wook-Hee Kim, R Madhava Krishnan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. 2021. PACTree: A High Performance Persistent Range Index Using PAC Guidelines. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 424–439.

[30] Konrad Kułakowski. 2014. A Concurrent van Emde Boas Array As A Fast and Simple Concurrent Dynamic Set Alternative. *Concurrency and Computation: Practice and Experience* 26, 2 (2014), 360–379.

[31] Konrad Kułakowski. 2015. Dynamic Concurrent van Emde Boas Array. *arXiv preprint arXiv:1509.06948* (2015), 18 pages.

[32] 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.

[33] Mengxing Liu, Jiankai Xing, Kang Chen, and Yongwei Wu. 2019. Building Scalable NVM-based B+ Tree with HTM. In *Proceedings of the 48th International Conference on Parallel Processing*. 1–10.

[34] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2020. Dash: Scalable Hashing on Persistent Memory. *Proceedings of the VLDB Endowment* 13, 8 (April 2020), 1147–1161.

[35] 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 Conference on File and Storage Technologies (FAST 21)*. 1–16.

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

[37] 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 *Intl. Symp. on Distributed Computing (DISC)*. Vienna, Austria, 37:1–37:16.

[38] Tri M Nguyen and David Wentzlaff. 2018. PiCL: A Software-transparent, Persistent Cache Log for Nonvolatile Main Memory. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 507–519.

[39] Liangxu Nie, Shengan Zheng, Bowen Zhang, Jinyan Xu, and Linpeng Huang. 2023. Heart: a Scalable, High-performance ART for Persistent Memory. In *2023 IEEE 41st International Conference on Computer Design (ICCD)*. IEEE, 487–490.

[40] Matheus Almeida Ogleari, Ethan L Miller, and Jishen Zhao. 2018. Steal But No Force: Efficient Hardware Undo+ Redo Logging for Persistent Memory Systems. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 336–349.

[41] Or Ostrovsky and Adam Morrison. 2020. Scaling Concurrent Queues By Using HTM To Profit From Failed Atomic Operations. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 89–101.

[42] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-tree for Storage Class Memory. In *Proceedings of the 2016 International Conference on Management of Data*. 371–386.

[43] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica* 33, 4 (June 1996), 351–385. doi:10.1007/s002360050048

[44] SeongJae Park, Paul E McKenney, Laurent Dufour, and Heon Y Yeom. 2020. An HTM-based Update-side Synchronization for RCU On NUMA Systems. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–15.

[45] Matej Pavlovic, Alex Kogan, Virendra J Marathe, and Tim Harris. 2018. Brief Announcement: Persistent Multi-Word Compare-and-Swap. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*. 37–39.

[46] Ivy B Peng, Maya B Gokhale, and Eric W Green. 2019. System Evaluation of The Intel Optane Byte-addressable NVM. In *Proceedings of the International Symposium on Memory Systems*. 304–315.

[47] Dimitrios Siakavaras, Konstantinos Nikas, Georgios Goumas, and Nectarios Koziris. 2017. RCU-HTM: Combining RCU with HTM To Implement Highly Efficient Concurrent Binary Search Trees. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 1–13.

[48] Anubhav Srivastava and Trevor Brown. 2022. Elimination (a, b)-Trees With Fast, Durable Updates. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 416–430.

[49] Ryan Stutsman, Justin Levandoski, and Darko Makreshanski. 2015. To lock, Swap, or Elide: On the Interplay of Hardware Transactional Memory and Lock-free Indexing. *Proceedings of the VLDB Endowment* 8, 11 (2015), 1298–1309.

[50] Peter van Emde Boas. 1975. Preserving Order in a Forest in Less Than Logarithmic Time. In *16th Annual Symp. on Foundations of Computer Science (FOCS)*. Berkeley, CA, 75–84. doi:10.1109/SFCS.1975.26

[51] Lukas Vogel, Alexander Van Renen, Satoshi Imamura, Jana Giceva, Thomas Neumann, and Alfons Kemper. 2022. Plush: A Write-optimized Persistent Log-structured Hash-table. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2895–2907.

[52] Chao Wang, Junliang Hu, Tsun-Yu Yang, Yuhong Liang, and Ming-Chang Yang. 2023. {SEPH}: Scalable, Efficient, and Predictable Hashing on Persistent Memory. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 479–495.

[53] 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.

[54] Tianzheng Wang, Justin Levandoski, and Per-Ake Larson. 2018. Easy Lock-Free Indexing in Non-Volatile Memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 461–472.

[55] 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 *Proceedings of the 50th International Conference on Parallel Processing*. 1–11.

[56] Lingfeng Xiang, Xingsheng Zhao, Jia Rao, Song Jiang, and Hong Jiang. 2022. Characterizing The Performance of Intel Optane Persistent Memory: A Close Look At Its On-dimm Buffering. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 488–505.

[57] Yi Xu, Joseph Izraelevitz, and Steven Swanson. 2021. Clobber-NVM: Log Less, Re-execute More. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 346–359.

[58] 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 Conference on File and Storage Technologies (FAST 15)*. 167–181.

[59] Jifei Yi, Mingkai Dong, Fangnuo Wu, and Haibo Chen. 2022. {HTMFS}: Strong Consistency Comes for Free with Hardware Transactional Memory in Persistent Memory File Systems. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. 17–34.

[60] Jianping Zeng, Jungi Jeong, and Changhee Jung. 2023. Persistent Processor Architecture. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 1075–1091.

[61] Bowen Zhang, Shengan Zheng, Liangxu Nie, Zhenlin Qi, Hongyi Chen, Linpeng Huang, and Hong Mei. 2024. Revisiting PM-based B-Tree with Persistent CPU Cache. *IEEE Transactions on Parallel and Distributed Systems* 35, 5 (May 2024), 796–813.

[62] Bowen Zhang, Shengan Zheng, Liangxu Nie, Zhenlin Qi, Linpeng Huang, and Hong Mei. 2024. Exploiting Persistent CPU Cache for Scalable Persistent Hash Index. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 3851–3864.

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

[64] Weihua Zhang, Xin Wang, Shiyu Ji, Ziyun Wei, Zhaoguo Wang, and Haibo Chen. 2017. Eunomia: Scaling Concurrent Index Structures Under Contention Using HTM. *IEEE Transactions on Parallel and Distributed Systems* 29, 8 (2017), 1837–1850.

[65] Yijie Zhong, Zhirong Shen, Zixiang Yu, and Jiwu Shu. 2023. Redesigning High-Performance LSM-based Key-value Stores With Persistent CPU Caches. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 1098–1111.

[66] Taiyu Zhou, Yajuan Du, Fan Yang, Xiaojian Liao, and Youyou Lu. 2022. Efficient Atomic Durability on eADR-enabled Persistent Memory. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 124–134.

[67] 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.

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

[69] Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. 2019. Efficient Lock-free Durable Sets. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–26.