BzTree: A High-Performance Latch-free Range Index for Non-Volatile Memory

Joy Arulraj[1]* Justin Levandoski[2] Umar Farooq Minhas[3] Per-Ake Larson[4]

[1]Carnegie Mellon University, [2,3]Microsoft Research, [4]University of Waterloo
[1]jarulraj@cs.cmu.edu, [2]justin.levandoski@microsoft.com,[3]ufminhas@microsoft.com, [4]plarson@uwaterloo.ca

ABSTRACT

Storing a database (rows and indexes) entirely in non-volatile memory (NVM) potentially enables both high performance and fast recovery. To fully exploit parallelism on modern CPUs, modern main-memory databases use latch-free (lock-free) index structures, e.g. Bw-tree or skip lists. To achieve high performance NVM-resident indexes also need to be latch-free. This paper describes the design of the BzTree, a latch-free B-tree index designed for NVM. The BzTree uses a persistent multi-word compare-and-swap operation (PMwCAS) as a core building block, enabling an index design that has several important advantages compared with competing index structures such as the Bw-tree. First, the BzTree is latch-free yet simple to implement. Second, the BzTree is fast - showing up to 2x higher throughput than the Bw-tree in our experiments. Third, the BzTree does not require any special-purpose recovery code. Recovery is near-instantaneous and only involves rolling back (or forward) any PMwCAS operations that were in-flight during failure. Our end-to-end recovery experiments of BzTree report an average recovery time of 145 μs. Finally, the same BzTree implementation runs seamlessly on both volatile RAM and NVM, which greatly reduces the cost of code maintenance.

PVLDB Reference Format:
Joy Arulraj, Justin Levandoski, Umar Farooq Minhas,Per-Ake Larson. BzTree: A High-Performance Latch-free Range Index for Non-Volatile Memory. *PVLDB*, 11(5): 553 - 565, 2018.
DOI: https://doi.org/10.1145/3164135.3164147

1 Introduction

Multi-threaded concurrency is one of the keys to unlocking high performance in main-memory databases. To achieve concurrency on modern CPUs, several systems – both research and commercial – implement latch-free index structures to avoid bottlenecks inherent in latching (locking) protocols. For instance, MemSQL uses latch-free skip-lists [30], while Microsoft's Hekaton main-memory OLTP engine uses the Bw-tree [6], a latch-free B+Tree.

The algorithms for latch-free index designs are often complex. They rely on atomic CPU hardware primitives such as compare-and-swap (CAS) to atomically modify index state. These atomic instructions are limited to a single word, and non-trivial data structures – such as a latch-free B+Tree – usually require multi-word updates, e.g., to handle operations like node splits and merges. These operations have to be broken up into multiple steps, thereby exposing intermediate states to other threads. As a result, the algorithms must handle subtle race conditions that may occur when intermediate states are exposed. In addition, some designs sacrifice performance to achieve latch-freedom. An example is the Bw-tree [22] that uses a mapping table to map logical page identifiers to physical pointers. Nodes in the Bw-tree store logical pointers and must dereference the mapping table on each node access during traversal of the index. Such indirection leads to degraded performance on modern CPUs.

Storing a main-memory database on byte-addressable non-volatile memory (NVM) further complicates implementation of latch-free indexes. NVM devices are becoming available in the form of NVDIMM [27, 35], Intel 3D XPoint [5], and STT-MRAM [14]. NVM provides close-to-DRAM performance and can be accessed by normal `load` and `store` instructions. Storing both records *and* indexes in NVM enables almost *instant* recovery, requiring only a small amount of work before the database is online and active. Several systems have taken this approach [3, 29, 34, 38], and is also the architecture we assume in this paper.

The added complexity in implementing latch-free indexes in NVM is mainly caused by the fact that CAS and other atomic hardware instructions do not persist their updates to NVM automatically and atomically. An update only modifies the target word in the processor cache and does not automatically update the target word in NVM. In case of a power failure, the volatile cache content is lost and the data in NVM may be left in an inconsistent state. Hence, we need a persistence protocol to ensure that an index or other data structure recovers correctly after a system crash.

In this paper, we propose the BzTree, a high-performance latch-free B+Tree design for main-memory databases that has the following benefits.

Reduced complexity. The BzTree implementation makes use of PMwCAS: a high-performance, multi-word, compare-and-swap operation that also provides persistence guarantees when used on NVM [37]. The PMwCAS operation is implemented in software and require no special hardware support other than a CAS (or equivalent) instruction. It is itself latch-free and either atomically installs all new values or fails the operation without exposing intermediate state. Using PMwCAS to build a latch-free index has two major advantages. First, the PMwCAS guarantees that all multi-word updates are atomic, thus avoiding the need to handle complex race conditions that result from exposing intermediate state during multi-word operations.

*Work performed while at Microsoft Research.

Proceedings of the VLDB Endowment, Vol. 11, No. 5

DOI: https://doi.org/10.1145/3164135.3164147

Second, PMwCAS allows the BzTree to avoid logical-to-physical indirection used, for example, in the Bw-tree [22]. The BzTree stores direct memory pointers in both index and leaf nodes.

High performance. Using the YCSB workload on volatile RAM, we show that the BzTree outperforms the Bw-tree. This demonstrates that the BzTree outperforms a state-of-the-art index designed for DRAM-based systems. Given its portability, we also experimentally demonstrate that the penalty for running the BzTree on NVM is low: on realistic workloads, the overhead of persistence is 8% on average. We also show that use of PMwCAS exhibits negligible contention even for larger multi-word operations. Even for highly skewed YCSB access patterns, the failure rate for updating multiple words across multiple BzTree nodes is only 0.2% on average.

Seamless portability to NVM. The same BzTree implementation can run on both volatile DRAM and on NVM without *any* code changes. PMwCAS guarantees that upon success of an update (in this case to B+Tree nodes), the operation will be durable on NVM and persist across failures. Remarkably, recovery is handled entirely by the PMwCAS library without any BzTree specific recovery code.

The rest of this paper is organized as follows. In Sections 2 and 3, we present the necessary background on the BzTree along with an overview of its architecture. Section 4 presents the BzTree node layout and single-node updates, while Section 5 covers structure modifications. Durability and recoverability on NVM are covered in Section 6. We present our experimental evaluation in Section 7, while Section 8 covers related work. Finally, we conclude the paper in Section 9.

2 Background and Overview

2.1 System Model and NVM

We assume a system model with a single-level store where NVM is attached directly to the memory bus. This model was also adopted by several recent NVM based systems [3, 28, 34, 36, 38]. We assume that indexes and base data reside in NVM. The system may also contain DRAM which is used as working storage.

NVM devices, such as NVDIMM products [35, 27] behave like DRAM but data stored on these devices is persistent and survives across power failures. Unlike hard disk drives (HDDs) or solid state drives (SSDs), data in NVM is accessible through normal `load` and `store` instructions. NVDIMMs are DRAM whose data content is saved to flash storage on power failure, so their performance characteristics are equivalent to that of DRAM. At the time of this writing, Intel's 3D XPoint DIMMs are not yet available but they are expected to have larger capacity than DRAM DIMMs, somewhat higher read latency, and lower bandwidth [10, 5]. NVDIMMs based on STT-MRAM technology are already available [7]; their performance is on par with DRAM but their capacity is very low.

When an application issues a `store` to a location on NVM, the store lands in the volatile CPU caches. To ensure the durability of the store, we must flush it from the CPU caches using a Cache Line Write Back (CLWB) or Cache Line FLUSH instruction (CLFLUSH) on Intel processors [15]. Both instructions flush the target cache line to memory but CLFLUSH also evicts the cache line.

2.2 Latch-Free Memory Safety

Latch-free data structure implementations require a mechanism to manage memory lifetime and garbage collection; since there are no locks protecting memory deallocation, the system must ensure no thread can dereference a block of memory before it is freed. The BzTree uses a high-performance epoch-based recycling scheme [23]. A thread joins the current epoch before each operation it performs on the index to protect the memory it accesses from reclamation. It exits the epoch when it finishes its operation. When all the threads that joined an epoch $\mathcal{E}$ have completed and exited, the garbage collector reclaims the memory occupied by the descriptors deallocated in $\mathcal{E}$. This ensures that no thread can possibly dereference a pointer after its memory is reclaimed.

PMWCAS STATUS	UNDECIDED
PMWCAS SIZE	3 SUB-OPERATIONS

TARGET WORD'S ADDRESS	EXPECTED OLD VALUE	NEW VALUE	DIRTY BIT	MEMORY RECYCLING POLICY
ADDRESS–1	VALUE O	VALUE N	0	NONE
ADDRESS–2	POINTER X	POINTER Y	1	FREE ONE
ADDRESS–3	POINTER Q	POINTER R	1	FREE ONE

Figure 1: PMwCAS Descriptor Table – Contents of the descriptor table used by threads to share information about the PMwCAS operation.

2.3 Persistent Multi-Word CAS

The BzTree relies on an efficient and persistent multi-word compare-and-swap operation, named PMwCAS, to update state in a latch-free and persistent manner. The design is based on a volatile version by Harris et al [11], which we enhance to guarantee persistence on NVM (details in [37]). The approach uses a descriptor to track metadata for the operation (details described later); these descriptors are pooled and eventually reused. The API for the PMwCAS is:

- `AllocateDescriptor(callback = default)`: Allocate a descriptor that will be used throughout the PMwCAS operation. The user can provide a custom callback function for recycling memory pointed to by the words in the PMwCAS operation.
- `Descriptor::AddWord(address, expected, desired)`: Specify a word to be modified. The caller provides the address of the word, the expected value and the desired value.
- `Descriptor::ReserveEntry(addr, expected, policy)`: Similar to `AddWord` except the new value is left unspecified; returns a pointer to the `new_value` field so it can be filled in later. Memory referenced by `old_value`/`new_value` will be recycled according to the specified policy.
- `Descriptor::RemoveWord(address)`: Remove the word previously specified as part of the PMwCAS.
- `PMwCAS(descriptor)`: Execute the PMwCAS and return true if succeeded.
- `Discard(descriptor)`: Cancel the PMwCAS (only valid before calling PMwCAS). No specified word will be modified.

The API is identical for both volatile and persistent MWCAS. Under the hood, PMwCAS provides all the needed persistence guarantees, without additional actions by the application.

To use PMwCAS, the application first allocates a descriptor and invokes the `AddWord` or `ReserveEntry` method once for each word to be modified. It can use `RemoveWord` to remove a previously specified word if needed. `AddWord` and `ReserveEntry` ensure that target addresses are unique and return an error if they are not. Calling PMwCAS executes the operation, while `Discard` aborts it. A failed PMwCAS will leave all target words unchanged. This behavior is guaranteed across a power failure when operating on NVM.

2.3.1 Durability

When running on NVM, the PMwCAS provides durability guarantees through the use of instructions to selectively flush or write back a cache line, e.g., via the cache line write-back (CLWB) or cache line flush (CLFLUSH without write-back) instructions on Intel processors [16]. These instructions are carefully placed to ensure linearizable reads and writes and also guarantee correct recovery in case of a crash or power failure. This is achieved by using a single

dirty bit on all modified words that are observable by other threads during the PMwCAS. For example, each modification that installs a descriptor address (or target value) sets a dirty bit to signify that the value is volatile, and that a reader must flush the value and unset the bit before proceeding. This protocol ensures that any dependent writes are guaranteed that the value read will survive power failure.

2.3.2 Execution

Internally, PMwCAS makes use of a *descriptor* that stores all the information needed to complete the operation. Figure 1 depicts an example descriptor for three target words. A descriptor contains, for each target word, (1) the target word's address, (2) the expected value to compare against, (3) the new value, (4) the dirty bit, and (5) a memory recycling policy. The policy field indicates whether the new and old values are pointers to memory objects and, if so, which objects are to be freed on the successful completion (or failure) of the operation. The descriptor also contains a status word tracking the operation's progress. The PMwCAS operation itself is latch-free; the descriptor contains enough information for any thread to help complete (or roll back) the operation. The operation consists of two phases.

Phase 1. This phase attempts to install a pointer to the descriptor in each target address using a *double-compare single-swap* (RDCSS) operation [11]. RDCSS applies change to a target word only if the values of two words (including the one being changed) match their specified expected values. That is, RDCSS requires an additional "expected" value to compare against (but not modify) compared to a regular CAS. RDCSS is necessary to guard against subtle race conditions and maintain a linearizable sequence of operations on the same word. Specifically, we must guard against the installation of a descriptor for a completed PMwCAS (p_1) that might inadvertently overwrite the result of another PMwCAS (p_2), where p_2 should occur after p_1 (details in [37]).

A descriptor pointer in a word indicates that a PMwCAS is underway. Any thread that encounters a descriptor pointer helps complete the operation before proceeding with its own work, making PMwCAS cooperative (typical for latch-free operations). We use one high order bit (in addition to the dirty bit) in the target word to signify whether it is a descriptor or regular value. Descriptor pointer installation proceeds in a target address order to avoid deadlocks between two competing PMwCAS operations that might concurrently overlap.

Upon completing Phase 1, a thread persists the target words whose dirty bit is set. To ensure correct recovery, this must be done before updating the descriptor's status field and advancing to Phase 2. We update status using CAS to either Succeeded or Failed (with the dirty bit set) depending on whether Phase 1 succeeded. We then persist the status field and clear its dirty bit. Persisting the status field "commits" the operation, ensuring its effects survive even across power failures.

Phase 2. If Phase 1 succeeds, the PMwCAS is guaranteed to succeed, even if a failure occurs – recovery will roll forward with the new values recorded in the descriptor. Phase 2 installs the final values (with the dirty bit set) in the target words, replacing the pointers to the descriptor. Since the final values are installed one by one, it is possible that a crash in the middle of Phase 2 leaves some target fields with new values, while others point to the descriptor. Another thread might have observed some of the newly installed values and make dependent actions (e.g., performing a PMwCAS of its own) based on the read. Rolling back in this case might cause data inconsistencies. Therefore, it is crucial to persist status before entering Phase 2. The recovery routine (covered next) can then rely on the status field of the descriptor to decide if it should roll forward or backward. If the PMwCAS fails in Phase 1, Phase 2 becomes a rollback procedure by installing the old values (with the dirty bit set) in all target words containing a descriptor pointer.

Recovery. Due to the two-phase execution of PMwCAS, a target address may contain a descriptor pointer or normal value after a crash. Correct recovery requires that the descriptor be persisted before entering Phase 1. The dirty bit in the status field is cleared because the caller has not started to install descriptor pointers in the target fields; any failure that might occur before this point does not affect data consistency upon recovery.

The PMwCAS descriptors are pooled in a memory location known to recovery. Crash recovery then proceeds by scanning the descriptor pool. If a descriptor's status field signifies success, the operation is rolled forward by applying the target values in the descriptor; if the status signifies failure it is rolled back by applying the old values. Uninitialized descriptors are simply ignored. Therefore, recovery time is determined by the number of in-progress PMwCAS operations during the crash; this is usually on the order of number of threads, meaning very fast recovery. In fact, in an end-to-end recovery experiment for the BzTree, we measured an average recovery time of 145 μs when running a write-intensive workload with 48 threads.

Memory management. Since the PMwCAS is latch-free, descriptor memory lifetime is managed by the epoch-based recycling scheme described in Section 2.2. This ensures that no thread can possibly dereference a descriptor pointer after its memory is reclaimed and reused by another PMwCAS. If any of the 8-byte expected or target values are pointers to larger memory objects, these objects can also be managed by the same memory reclamation scheme. Each word in the descriptor is marked with a memory recycling policy that denotes whether and what memory to free on completion of the operation. For instance, if a PMwCAS succeeds, the user may want memory behind the expected (old) value to be freed once the descriptor is deemed safe to recycle. Section 6 discusses the details of the interplay between PMwCAS and memory reclamation.

3 BzTree Architecture and Design

3.1 Architecture

The BzTree is a high-performance main-memory B+Tree. Internal nodes store search keys and pointers to child nodes. Leaf nodes store keys and either record pointers or actual payload values. Keys can be variable or fixed length. Our experiments assume leaf nodes store 8-byte record pointers as payloads (common in main-memory databases [6]), though we also discuss how to handle full variable-length payloads. The BzTree is a range access method that supports standard atomic key-value operations (insert, read, update, delete, range scan). Typical of most access methods, it can be deployed as a stand-alone key-value store, or embedded in a database engine to support ACID transactions, where concurrency control takes place outside of the access method as is common in most systems (e.g., within a lock manager) [12, 23].

Persistence Modes. A salient feature of the BzTree is that its design works for both volatile and persistent environments. In volatile mode, BzTree nodes are stored in volatile DRAM. Content is lost after a system failure. This mode is appropriate for use in existing main-memory system designs (e.g., Microsoft Hekaton [6]) that already contain recovery infrastructure to recover indexes. In durable mode, both internal and leaf nodes are stored in NVM. The BzTree guarantees that all updates are persistent and the index can recover quickly to a correct state after a failure. For disaster recovery (media failure), the BzTree must rely on common solutions like database replication.

Metadata. Besides nodes, there are only two other 64-bit values used by the BzTree:

- *Root pointer.* This is a 64-bit pointer to the root node of the index. When running in persistence mode, this value is persisted in a known location in order to find the index upon restart.
- *Global index epoch.* When running in persistence mode, the BzTree is associated with an *index epoch* number. This value is drawn from a global counter (one per index) that is initially zero for a new index and incremented *only* when the BzTree restarts after a crash. This value is persisted in a known location, and is used for recovery purposes and to detect in-flight operations (e.g., space allocations within nodes) during a crash. We elaborate on the use of this value in Sections 4 and 6.

3.2 Complexity and Performance

The BzTree design addresses implementation complexities and performance drawbacks of state-of-the-art latch-free range indexes.

Implementation complexities. State-of-the-art range index designs usually rely on atomic primitives to update state. This is relatively straightforward for single-word updates. For example, the Bw-tree [22] updates a node using a single-word CAS to install a pointer to a delta record within a mapping table. Likewise, designs like the MassTree [25] use a CAS on a status word to arbitrate node updates. The implementation becomes more complex when handling multi-location updates, such as node splits and merges that grow (or shrink) an index. The Bw-tree breaks multi-node operations into steps that can be installed with a single atomic CAS; similar approaches are taken by other high-performance indexes to limit latching across nodes. These multi-step operations expose intermediate state to threads that concurrently access the index. This means the implementation must have special logic in place to allow a thread to (a) recognize when it is accessing an incomplete index (e.g., seeing an in-progress split or node delete) and (b) take cooperative action to help complete an in-progress operation. This logic leads to code "bloat" and subtle race conditions that are difficult to debug [24].

As we will see, the BzTree uses the PMwCAS primitive to update index state. We show that this approach performs well even when updating multiple nodes atomically. The BzTree thus avoids the subtle race conditions for more complex multi-node operations. In fact, using cyclomatic complexity analysis[1], we show that the BzTree design is at least half as complex as the Bw-tree and MassTree [25], two state-of-the-art index designs.

Performance considerations. Some latch-free designs such as the Bw-tree rely on indirection through a mapping table to isolate updates (and node reorganizations) to a single location. Bw-tree nodes store logical node pointers, which are indexes into the mapping table storing the physical node pointers. This approach comes with a tradeoff. While it avoids propagation of pointer changes up the index, e.g. to parent nodes, it requires an extra pointer dereference when accessing each node. This effectively doubles the amount of pointer dereferences during index traversal, leading to reduced performance, as we show in our experimental evaluation (Section 7).

The BzTree does not rely on indirection to achieve latch-freedom. Interior index nodes store direct pointers to child nodes to avoid costly extra pointer dereferences during traversal. As we show later in Section 7, this translates into higher performance when compared to the state-of-the-art in latch-free index design.

4 BzTree Nodes

In this section, we begin by describing the BzTree node organization and then discuss how the BzTree supports latch-free reads and updates on these nodes. We then describe node consolidation: an operation that reorganizes a node to reclaim dead space and speed up search. We defer discussion of multi-node operations such as splits and merges until Section 5.

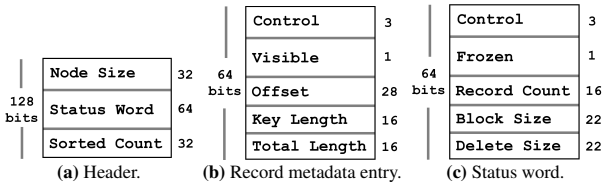


Figure 2: Node layout and details for the BzTree.

4.1 Node Layout

The BzTree node representation follows a typical slotted-page layout, where fixed-size metadata grows "downward" into the node, and variable-length storage (key and data) grow "upward." Specifically, a node consists of: (1) a fixed-size header, (2) an array of fixed-size record metadata entries, (3) free space that buffers updates to the node, and (4) a record storage block that stores variable-length keys and payloads. All fixed-sized metadata is packed into 64-bit aligned words so that it can easily be updated in a latch-free manner using PMwCAS.

Header. The header is located at the beginning of a node and consists of three fields as depicted in Figure 2a: (1) a `node size` field (32 bits) that stores the size of the entire node, (2) a `status word` field (64 bits) that stores metadata used for coordinating updates to a node (content discussed later in this section), and (3) a `sorted count` field (32 bits), representing the last index in the record metadata array in sorted order; any entries beyond this point might be unsorted and represent new records added to the node.

Record metadata array. Figure 2b depicts an entry in the record metadata array that consists of: (1) flag bits (4 bits) that are broken into PMwCAS `control bits`[2] (3 bits) used as internal metadata for the PMwCAS (e.g., to mark dirty words that require a flush) along with a `visible` flag (1 bit) used to mark a record as visible, (2) an `offset` value (28 bits) points to the full record entry in the key-value storage block, (3) a `key length` field (16 bits) stores the variable-length key size, and (4) a `total length` field (16 bits) stores the total length of the record block; subtracting `key length` from this value provides the record payload size.

Free space. Free space is used to absorb modifications to a node such as record inserts. This free space sits between the fixed-size record metadata array and the record storage block. The record metadata array grows "downward" into this space, while the data storage block grows "upward." However, internal index nodes do not contain free space; as we will discuss later, these nodes are search-optimized and thus do not buffer updates, as doing so results in degraded binary search performance.

Record storage block. Entries in this block consist of contiguous key-payload pairs. Keys are variable-length byte strings. Payloads in internal BzTree nodes are fixed-length (8-byte) child node pointers. In this paper, we assume payloads stored in leaf nodes are 8-byte record pointers (as is common in main-memory databases [6]). However, the BzTree also supports storing full variable-length payloads within leaf nodes. We discuss how to update both types of payloads later in this section.

Status word. The status word, depicted in Figure 2c, is a 64-bit value that stores node metadata that changes during an update.

[1]Cyclomatic complexity is a quantitative measure of the number of linearly independent paths through source code.

[2]PMwCAS relies on these bits to function property. Due to space limitations, we do not detail these bits. Please see [37] for a description.

For leaf nodes, this word contains the following fields: (1) PMwCAS `control bits` (3 bits) used to atomically update the word, (2) a `frozen` flag (1 bit) that signals that the node is immutable, (3) a `record count` field (16 bits) that stores the total number of entries in the record metadata array, (4) a `block size` field (22 bits) storing the number of bytes occupied by the record storage block at the end of the node, and (5) a `delete size` field (22 bits) that stores the amount of logically deleted space on the node, which is useful for deciding when to merge or reorganize the node. Status words for *internal nodes* only contain the first two fields; this is because we do not perform singleton updates on internal nodes and thus do not need the other fields. We opt to replace internal nodes wholesale (e.g., when adding or deleting a record) for search performance reasons.

Internal and Leaf Node Differences. Besides status word format, internal and leaf nodes differ in that internal nodes are immutable once created, while leaf nodes are not. Internal nodes only store records in sorted order by key (for fast binary search) and do not contain free space. Leaf nodes, on the other hand, contain free space in order to buffer inserts (and updates if the leaf nodes store full record payloads). This means that leaf nodes consist of both sorted records (records present during node creation) and unsorted records (records added to the page incrementally). We chose this approach because the vast majority of updates in a B+Tree occur at the leaf level, thus it is important to have leaf nodes quickly absorb record updates "in place". On the other hand, internal index nodes are read-mostly and change less frequently, thus can tolerate wholesale replacement, e.g., when adding a new key as a result of a node split. In our experience, keeping internal index nodes search-optimized leads to better performance than an alternative approach that organizes internal nodes with both sorted and unsorted key space [22].

4.2 Leaf Node Operations

This section describes the latch-free read and update operations on BzTree leaf nodes. For writes, the basic idea is to employ the PMwCAS to manipulate the page and record metadata atomically in a latch-free manner, for both reserving space (in the case of copying variable length data into the page) and making the update "visible" to concurrent threads accessing the page. Readers access pages uncontested; they are not blocked by writers. Table 1 summarizes the PMwCAS operations associated with all the tree operations.

4.2.1 Inserts

New records are added to the free space available in the node. To insert a new record r, a thread first reads the frozen bit. If it is set, this means the page is immutable and may no longer be part of the index (e.g., due to a concurrent node delete). In this case the thread must re-traverse the index to find the new incarnation of the "live" leaf node. Otherwise, the thread reserves space for r in both the record metadata array and record storage block. This is done by performing a 2-word PMwCAS on the following fields: (1) the node's `status` word to atomically increment the `record count` field by one and add the size of r to the `block size` value, and (2) the record metadata array entry to flip the `offset` field's high-order bit and set the rest of its bits equal to the global index epoch[3]. If this PMwCAS succeeds, the reservation is a success. The `offset` field is overridden during this phase to remember the allocation's index epoch. We refer to this value as the `allocation epoch` and use it for recovery purposes (explained later). We steal the high-order bit to signal whether the value is an allocation epoch (set) or actual record offset (unset).

[3]Note that setting this field atomically along with the reservation is safe, since it will only succeed if the space allocation succeeds.

Table 1: PMwCAS Summary Table – The size of the PMwCAS operations associated with different node and structure modification operations.

Tree Operation	PMwCAS Size
NODE OPERATIONS	
Insert [Allocation, Completion]	2, 2
Delete	2
Update [Record Pointer, Inlined Payload]	3, 2
Node Consolidation	2
SMOs	
Node Split [Preparation, Installation]	1, 3
Node Merge [Preparation, Installation]	2, 3

The insert proceeds by copying the contents of r to the storage block and updating the fields in the corresponding record metadata entry, initializing the `visible` flag to 0 (invisible). Once the copy completes, the thread flushes r (using CLWB or CLFLUSH) if the index must ensure persistence. It then reads the status word value s to again check the frozen bit, aborting and retrying if the page became frozen (e.g., due to a concurrent structure modification). Otherwise, the record is made visible by performing a 2-word PMwCAS on (1) the 64-bit record metadata entry to set the `visible` bit and also set the `offset` field to the actual record block offset (with its high-order bit unset) and (2) the status word, setting it to s (the same value initially read) to detect conflict with a concurrent thread trying to set the frozen bit. If the PMwCAS succeeds, the insert is a success. Otherwise, the thread re-reads the status word (ensuring the frozen bit is unset) and retries the PMwCAS.

Concurrency issues. The BzTree must be able to detect concurrent inserts of the same key to enforce, for instance, unique key constraints. We implement an optimistic protocol to detect concurrent key operations as follows. When an insert operation first accesses a node, it searches the sorted key space for its key and aborts if the key is present. Otherwise, it continues its search by scanning the unsorted key space. If it sees any record with an unset `visible` flag and an `allocation epoch` value equal to the current *global index epoch*, this means it has encountered an in-progress insert that *may* be for the same key. An entry with an unset `visible` flag and an `allocation epoch` *not* equal to the *global index epoch* means it is either deleted or its allocation was in-progress during a crash from a previous incarnation of the index and can be ignored (details in Section 6.3). Instead of waiting for the in-progress insert to become visible, the thread sets an internal *recheck* flag to remember to re-scan the unsorted key space and continues with its insert. The *recheck* flag is also set if the thread loses a PMwCAS to reserve space for its insert since the concurrent reservation may be for the same key. Prior to setting its own visibility bit, the thread re-scans the unsorted key space if the *recheck* flag is set and examines all prior entries before its own position. Upon encountering a duplicate key, the thread zeroes out its entry in the record storage block and sets its `offset` value to zero; these two actions signify a failed operation that will be ignored by subsequent searches. If the thread encounters an in-progress operation during its scan, it must wait for the record to become visible, since this represents an operation that serialized behind the insert that *may* contain a duplicate key.

4.2.2 Delete

To delete a record, a thread performs a 2-word PMwCAS on (1) a record's metadata entry to unset its `visible` bit and set its `offset` value to zero, signifying a deleted record and (2) the node `status word` to increment the `delete size` field by the size of the target record. If the PMwCAS fails due to a concurrent delete or conflict on

the status word, the thread retries the delete. If the failure is due to a concurrent operation that set the `frozen` bit on the node, the delete must re-traverse the index to retry on a mutable leaf node. Incrementing `delete size` allows the BzTree to determine when to delete or consolidate a node (Section 5).

4.2.3 Update

There are two methods to update an existing record, depending on whether a leaf node stores record pointers or full payloads.

- **Record pointers**. If leaf nodes contain record pointers and the user wishes to update a record in-place, the BzTree is passive and the update thread can simply traverse the pointer to access the record memory directly. If the update requires swapping in a *new* record pointer, this can be done in place within the record storage block. To do this, a thread reads both (a) the record metadata entry m to ensure it is not deleted and (b) the status word s to ensure the node is not frozen. It then performs a 3-word `PMwCAS` consisting of (1) the 64-bit pointer in the storage block to install the new pointer, (2) the record's metadata entry, setting it to m (the same value as it read) to detect conflict with a competing delete trying to modify the word, and (3) the status word, setting it to s (the same value it read) to detect conflict with a competing flip of the frozen bit.
- **Inline payloads**. If leaf nodes store full payloads, the update follows the same protocol as an insert by (1) allocating space in the metadata array and record storage block and (2) writing a (key, `update_payload`) record into the record block that describes the update. The `update_payload` can be either a full payload replacement or a "byte diff" describing only the part(s) of the payload that have changed. Unlike inserts, we treat concurrent updates to the same key as a natural race, supporting the "last writer wins" protocol. This means there is no need to detect concurrent updates to the same key.

4.2.4 Upsert

The BzTree supports the upsert operation common in most key-value stores. If the record exists in the leaf node, the thread performs an update to that record. If the record does not exist, the thread performs an insert. In this case if the insert fails due to another concurrent insert, the operation can retry to perform an update.

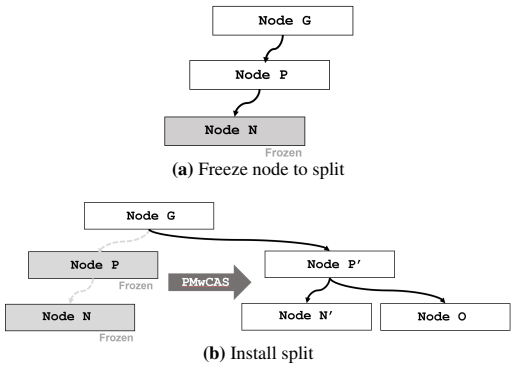
4.2.5 Reads

BzTree update operations do not block readers. A reader traverses the index to the target leaf node. If the leaf node stores record pointers, a thread first performs a binary search on the sorted key space. If it does not find its search key (either the key does not exist or was deleted in the sorted space), it performs a sequential scan on the unsorted key space. If the key is found, it returns the record to the user. If leaf nodes store full record payloads, the search first scans the unsorted key space starting from the most recent entry, as recent update records will represent the latest payload for a record. If the key is not found, the search continues to the sorted key space.

A read simply returns the most recent record it finds on the node that matches its search key. It ignores all concurrent update activity on the node by disregarding both the `frozen` bit and any in-progress record operations (unset `visible` bits). These concurrent operations are treated as natural races, since (a) any record-level concurrency must be handled outside the BzTree and (b) the frozen bit does not matter to reads, as it is used by operations attempting to reorganize the node to serialize with updates.

4.2.6 Range Scans

The BzTree supports range scans as follows. A user opens a scan iterator by specifying a `begin_key` and an optional `end_key` (null if



(a) Freeze node to split

(b) Install split

Figure 3: Node split in the BzTree.

open-ended) defining the range they wish to scan. The scan then proceeds one leaf node at a time until termination. It begins by entering an epoch to ensure memory stability and uses the `begin_key` to find the initial leaf node. When entering a page, the iterator constructs a response array that lists the valid records (i.e., visible and not deleted) on the node in sorted order. In essence, the response array is a snapshot copy of the node's valid records in its record storage block. After copying the snapshot, the iterator exits its epoch so as to not hold back memory garbage collection. It then services record-at-a-time `get_next` requests out of its snapshot. Once it exhausts the response array, the iterator proceeds to the next leaf node by entering a new epoch and traversing the tree using a "greater than" search on the largest key in the response array; this value represents the high boundary key of the previous leaf node and will allow the traversal to find the next leaf node position in the scan. This process repeats until the iterator can no longer satisfy the user-provided range boundaries, or the user terminates the iterator.

4.3 Leaf Node Consolidation

Eventually a leaf node's search performance and effective space utilization degrade due to side effects of inserts or deletes. Search degrades due to (a) the need to sequentially scan the unsorted key space (in the case of many inserts) and/or (b) a number of deletes adding to the "dead space" within the sorted key space, thereby inflating the cost of binary search. The BzTree will occasionally consolidate (reorganize) a leaf node to increase search performance and eliminate dead space. Consolidation is triggered when free space reaches a minimum threshold, or the amount of logically deleted space on the node is greater than a configurable threshold.

To perform consolidation of a node $\mathcal{N}$, a thread first performs a single-word `PMwCAS` on the $\mathcal{N}$'s status word to set its frozen flag. This prevents any ongoing updates from completing and ensures the consolidation process sees a consistent snapshot of $\mathcal{N}$'s records. The process then scans $\mathcal{N}$ to locate pointers to all live records on the page – ignoring deleted and invisible records – and calculates the space needed to allocate a fresh node (the size of all valid records plus free space). If this space is beyond a configurable max page size, the process invokes a node split (covered in Section 5). Otherwise, it allocates memory for a new node $\mathcal{N}'$ along with some free space to buffer new node updates. It then initializes the header and copies over all live records from $\mathcal{N}$ to $\mathcal{N}'$ in key-sequential order. Now, $\mathcal{N}'$ contains all sorted records and is ready to replace $\mathcal{N}$.

Making $\mathcal{N}'$ visible in the index requires "swapping out" a pointer to $\mathcal{N}$ at its parent node $\mathcal{P}$ to replace it with a pointer to $\mathcal{N}'$. To do this, the thread uses its path stack (a stack recording node pointers during traversal) to find a pointer to $\mathcal{P}$. If this pointer represents a frozen page, the thread must re-traverse the index to find the valid parent. It then finds the record r in $\mathcal{P}$ that stores the child pointer to

$\mathcal{N}$ and performs an in-place update using a 2-word PMwCAS on (1) the 64-bit child pointer in r to install the pointer to $\mathcal{N}'$ and (2) $\mathcal{P}$'s status word to detect a concurrent page freeze. If this PMwCAS succeeds, $\mathcal{N}'$ is now live in the index and $\mathcal{N}$ can be garbage collected. However, $\mathcal{N}$ cannot be immediately freed, since this process is latch-free and other threads may still have pointers to $\mathcal{N}$. As discussed in Section 2, the BzTree handles this case by using an epoch-based garbage collection approach to safely free memory.

Concurrency during consolidation. Freezing a node prior to consolidation will cause any in-progress updates on that node to fail, as they will detect the set frozen bit when attempting a PMwCAS on the status word. The failed operations will then retry by re-traversing the tree to find a new "live" leaf node. If they again land on a frozen node, this is a signal to help along to complete the consolidation instead of "spinning" by continuously re-traversing the index hoping for a live node. In this case, each thread will start its own consolidate process and attempt to install it at the parent. This effectively makes threads race to install a consolidated node, though one will ultimately win. Afterward, each thread resumes its original operation.

4.4 Internal Node Operations

Updates to existing records on internal nodes are performed in place following the protocol discussed in the previous section for installing a new child pointer. To maintain search optimality of internal nodes, record inserts and deletes (e.g., part of splitting or deleting a child node) create a completely new version of an internal node. In other words, an insert or delete in an internal node immediately triggers a consolidation. This process is identical to the leaf node consolidation steps just discussed: a new node will be created (except with one record added or removed), and its pointer will be installed at the parent.

5 Structure Modifications

We now describe the latch-free algorithms used in the BzTree for structure modification operations (SMOs). Like single-node updates, the basic idea for SMOs is to employ the PMwCAS to update page state atomically and in a latch-free manner. This involves manipulating metadata like frozen bits, as well as manipulating search pointers within index nodes to point to new page versions (e.g., split pages).

We begin with a presentation of the node split and node merge algorithms. We then discuss the interplay between the algorithms when commingling structural changes and data changes. We also explain why threads concurrently accessing the tree are guaranteed to not observe inconsistencies, which simplifies both implementation and reasoning about correctness.

5.1 Prioritizing Structure Modifications

Triggering SMOs in the BzTree relies on a simple deterministic policy. A split is triggered once a node size passes a configurable `max_size` threshold (e.g., 4KB). Likewise, a node delete/merge is triggered once a node's size falls below a configurable `min_size`.

If an update thread encounters a node in need of an SMO, it temporarily suspends its operation to perform the SMO before continuing its operation (we do not force readers to perform SMOs). Given that SMOs are relatively heavyweight, prioritizing them over (lightweight) single-record operations is important. Otherwise, in a latch-free race, single-record operations would always win and effectively starve SMOs.

5.2 Node Split

Node splits are broken into two phases (1) a preparation phase that allocates and initializes new nodes with the SMO changes and (2) an installation phase that atomically installs the new nodes in the index. We now describe the split details with the aid of Figure 3.

Preparation. To split a node $\mathcal{N}$, we first perform a PMwCAS on its status word to set the `frozen` bit, as depicted in Figure 3a. We then scan $\mathcal{N}$ to find all valid records and calculate a separator key k that provides a balanced split. We then allocate and initialize three new nodes. (1) A new version of $\mathcal{N}$ (call it $\mathcal{N}'$) that contains all valid records with keys less than or equal to k, (2) a new sibling node $\mathcal{O}$ that contains all valid records with keys greater than k, and (3) a new version of $\mathcal{N}$'s parent node $\mathcal{P}$ (call it $\mathcal{P}'$) that replaces the child pointer of $\mathcal{N}$ with a pointer to $\mathcal{N}'$ and adds a new search record consisting of key k and a pointer to the new child $\mathcal{O}$. All nodes are consolidated (search-optimized) and store sorted records.

Installation. Installation of a split involves "swapping out" $\mathcal{P}$ to replace it with $\mathcal{P}'$, thereby making the new split nodes $\mathcal{N}'$ and $\mathcal{O}$ visible in the index. Figure 3b depicts this process. The installation is atomic and involves using a 3-word PMwCAS to modify the following words (1) the status word of $\mathcal{P}$ to set its frozen bit, failure to set the bit means it conflicts with another update to $\mathcal{P}$, (2) the 64-bit child pointer to $\mathcal{P}$ at its parent $\mathcal{G}$ ($\mathcal{N}$'s grandparent) to swap in the new pointer to $\mathcal{P}'$, and (3) $\mathcal{G}$'s status word to detect a concurrent page freeze. If the PMwCAS succeeds, the split is complete, and the old nodes $\mathcal{P}$ and $\mathcal{N}$ are sent to the epoch-protected garbage collector. On failure, a thread retries the split, and the memory for nodes $\mathcal{N}'$, $\mathcal{P}'$, and $\mathcal{O}$ can be deallocated immediately since they were never seen by another thread.

5.3 Node Merge

The BzTree performs node merges in a latch-free manner similar to node splits. Before triggering a delete of a node $\mathcal{N}$, we first find a sibling that will absorb $\mathcal{N}$'s existing records. We chose $\mathcal{N}$'s left sibling $\mathcal{L}$ if (1) it shares a common parent[4] $\mathcal{P}$ and (2) is small enough to absorb $\mathcal{N}$'s records without subsequently triggering a split (defeating the purpose of a merge). Otherwise, we look at $\mathcal{N}$'s right sibling $\mathcal{R}$, verifying it has enough space to absorb $\mathcal{N}$'s records without a split. If neither $\mathcal{R}$ nor $\mathcal{L}$ satisfy the merge constraints, we allow $\mathcal{N}$ to be underfull until these constraints are met. In the remainder of this section, we assume $\mathcal{N}$ merges with its sibling $\mathcal{L}$.

Preparation. To initiate the delete, we first perform a PMwCAS on the status word of both $\mathcal{L}$ and $\mathcal{N}$ to set their `frozen` bit. We then allocate and initialize two new nodes: (1) a new version of the left sibling $\mathcal{L}'$ containing its own valid records and all of $\mathcal{N}'s$ valid records, and (2) a new version of $\mathcal{N}$ and $\mathcal{L}$'s parent $\mathcal{P}'$ that replaces the child pointer of $\mathcal{L}$ with a pointer to $\mathcal{L}'$ and removes the search record containing the separator key between $\mathcal{L}$ and $\mathcal{N}$ along with the child pointer to $\mathcal{N}$.

Installation. Installation of the node delete and merge involves installing the new version of $\mathcal{P}'$ in the index that makes the merged child node $\mathcal{L}'$ visible and removes $\mathcal{N}$ and $\mathcal{L}$. This operation is identical to that of node split that replaces the parent $\mathcal{P}$ with $\mathcal{P}'$ by both freezing $\mathcal{P}$ as well as updating its parent $\mathcal{G}$ to install the new child pointer to $\mathcal{P}'$.

5.4 Interplay Between Algorithms

The BzTree offloads the handling of ACID transactions to a higher software layer of the system. This could, for instance, be a logical concurrency control component in a decoupled database system [22]. The index itself is responsible for correctly serializing conflicting data and structural changes. We now describe how BzTree ensures that threads do not observe the effects of in-progress changes.

[4]We chose to avoid merges that cross parent nodes in order to minimize the number of modified nodes.

Co-operative PMwCAS. B+Tree implementations typically rely on latches for preventing threads from observing changes performed by concurrent threads. The BzTree instead employs PMwCAS to accomplish this. As described in Section 2.3, we employ a latch-free PMwCAS library. The PMwCAS operation is cooperative, in that any thread (reader or writer) that encounters an in-progress PMwCAS will first help along to complete the operation before continuing with its own. This policy effectively serializes PMwCAS operations that might conflict. It also ensures the atomicity of operations within the BzTree. Since all updates to the index are performed using PMwCAS, updates will either succeed uncontested, or the PMwCAS help-along protocol will arbitrate conflict and abort some conflicting operations.

Record operations and structure modifications. BzTree employs the status word to correctly serialize conflicting data and structural changes that might conflict with each other. For instance, an in-progress consolidate or SMO will first set the frozen bit within a node. This causes all in-flight record-level operations to fail their PMwCAS due to conflict on the status word. These record operations will then retry and either see (a) the frozen version of a node that requires maintenance, for which it will attempt to complete or (b) a new (unfrozen) version of the node that is ready for record updates.

Serializing structure modifications. The BzTree uses a cooperative approach for serializing conflicting SMOs. Consider a node deletion operation. To delete node $\mathcal{N}$, the BzTree first checks if its left sibling $\mathcal{L}$ is alive. If it observes that $\mathcal{L}$ is frozen, then it detects that another structural change is in progress. In this case the BzTree serializes the deletion of $\mathcal{N}$ (if still needed) after that of $\mathcal{L}$.

6 BzTree Durability and Recovery

In this section, we illustrate how BzTree ensures recoverability of the tree across system failures using PMwCAS. BzTree stores the tree either on DRAM when used in volatile mode, or on NVM when used in durable mode. In volatile mode, the BzTree does not flush the state of the tree to durable storage. However, when used in durable mode, it persists the tree on NVM to preserve it across system failures. The BzTree does *not* need to employ a specific recovery algorithm. It instead relies on the recovery algorithms of a persistent memory allocator and the PMwCAS library to avoid persistent memory leaks and ensure recoverability, respectively. We now describe these algorithms in detail.

6.1 Persistent Memory Allocation

A classic volatile memory allocator with an `allocate` and `free` interface does not ensure correct recovery when used on NVM. If the allocator marks a memory chunk as being in use (due to `allocate`), and the application (e.g., BzTree) fails to install the allocated chunk on NVM before a crash, then this causes a persistent memory leak. In this state, the memory chunk is "homeless" in that it can neither be seen by the application nor by the memory allocator after a crash.

While creating a safe and correct persistent memory allocator is outside the scope of this paper, there have been many proposals. We assume availability of a three-stage allocator [31] that provides the following states: (1) `allocate`, (2) `activated`, and (3) `free`. The application first requests the allocation of a memory chunk. The allocator updates the chunk's meta-data to indicate that it has been `allocated` and returns it to the application. During recovery after a system failure, the allocator reclaims all `allocated` memory chunks. To retain the ownership of the memory chunk even after a failure, the application must separately request that the allocator `activate` the memory chunk. At this point in time, the application owns the memory chunk and is responsible for its lifetime, including any cleanup after a failure.

The application must carefully interact with the allocator in the activation process, through an interface (provided by the allocator) that is similar to `posix_memalign` which accepts a reference of the target location for storing the address of the allocated memory. This design is employed by many existing NVM systems [17, 28, 36, 31]. The application owns the memory only after the allocator has successfully persisted the address of the newly allocated memory in the provided reference.

6.2 Durability

There are two cases by which the BzTree handles durability of index data.

- **Variable-length data**. Newly inserted records as well as new node memory (allocated as part of a consolidate, split, or delete/merge) represents variable-length data in the BzTree. To ensure durability, the BzTree flushes all variable-length data before it can be read by other threads. That is, newly inserted record memory on a node is flushed before the atomic flip of its `visible` bit. Likewise, new node memory is flushed before it is "linked into" the index using a PMwCAS. This flush-before-visible protocol ensures that variable-length data in the BzTree is durable when it becomes readable to concurrent threads.
- **Word-size data**. The durability of word-size modifications is handled by the PMwCAS operation. As mentioned in Section 2.3, the PMwCAS ensures durability of all words it modifies upon acknowledging success. Thus, modifications like changing the node status word and reserving and updating a record's metadata entry are guaranteed to be durable when modified using the PMwCAS. In addition, all modifications performed by the PMwCAS are guaranteed to be durable to concurrent readers.

The BzTree avoids inconsistencies arising from write-after-read dependencies. That is, it guarantees that a thread *cannot* read a volatile modification made by another thread. Otherwise, any action taken after the read (such as a dependent write) might not survive across a crash and lead to an inconsistent index. As mentioned above, the flush-before-visible protocol ensures this property for variable-length modifications to the BzTree. Likewise, the PMwCAS ensures this property for word-sized modifications.

6.3 Recovery

Memory lifetime. The PMwCAS library maintains a pool of descriptors at a well-defined location on NVM. Each word descriptor contains a field specifying a memory recycling policy. This policy defines how the memory pointed to by the old value and new value fields should be handled when the PMwCAS operation concludes. The PMwCAS library supports two memory recycling policies: `NONE` and `FREE-ONE`. With the former policy, there is no need for recycling memory. The BzTree uses this policy for modifying non-pointer values, such as the status word in nodes. With the latter policy, the PMwCAS library frees the memory pointed to by the old (or new) value depending on whether the PMwCAS operation succeeds (or fails)[5]. The BzTree uses this policy when allocating and installing a new node in the tree. To *activate* the node memory, BzTree provides a memory reference to the descriptor word responsible for holding a pointer to the node memory. This ensures an atomic transfer of the activated memory pointer to the descriptor. The memory lifetime is then handled by the PMwCAS library. In case of a failure, the node's memory is reclaimed by the recovery algorithm. This obviates the need for BzTree to implement its own memory recycling mechanism.

[5]Memory lifetime safety is managed by the epoch-based mechanism described in Section 2.3

Recovery steps. During recovery from a system failure, the allocator first runs its recovery algorithm to reclaim memory chunks that have been reserved but not yet activated. Then, the PMwCAS library executes its recovery algorithm to ensure that the effects of all successfully completed PMwCAS operations are persisted. As covered in Section 2.3, upon restart after a crash, any in-flight PMwCAS operations marked as succeeded will roll forward, otherwise they will roll back. For operations involving memory pointer swaps, the PMwCAS will ensure that allocated and active memory dereferenced by its descriptors will be correctly handled according to the provided memory recycling policy.

Aborted space allocations. While PMwCAS recovery can handle recovery of 64-bit word modifications, including pointer swaps and node memory allocations, it cannot handle recovery of dangling record space allocations within a node. As detailed in 4.2, an insert[6] is broken into two atomic parts: ❶ record space allocation and ❷ record initialization (copying key bytes and populating metadata) and making the record visible. The BzTree must be able to detect and recover failed inserts that allocated space within a node in ❶, but crashed during ❷ before a record was fully populated and made visible. The BzTree uses the `allocation epoch` for this purpose (as described in Section 4.2.1, this value is temporarily stored in the `offset` field until ❷ completes). Since this field is populated atomically during ❶, any subsequent failure before completion of ❷ will be detected after recovery increments the *global index epoch*. Doing so will invalidate any searches – such as those done by inserts checking for duplicate keys – that encounter an allocation from a previous epoch. This dangling node space will be reclaimed when the node is rebuilt during consolidation or a structure modification.

7 Evaluation

7.1 Experimental Setup

7.1.1 Environment

We implemented the BzTree in approximately 3,000 lines of C++ code, using the PMwCAS library to ensure atomicity and durability of tree updates [37]. This library employs the Win32 native `InterlockedCompareExchange64` to perform CAS. NVM devices based on new material technologies (e.g., Intel 3D XPoint) are not yet commercially available. We instead target flash-backed NVDIMMs. These NVDIMMs are DRAM whose data content is saved to flash storage on power failure. We conduct experiments on a workstation running Windows Server 2012 on an Intel Xeon E7-8890 CPU (at 2.2GHz) with 24 physical cores.

7.1.2 Evaluation Workloads

YCSB Benchmark: The Yahoo! Cloud Serving benchmark (YCSB) approximates typical large-scale cloud service [4]. We construct a set of workload mixtures that are based on YCSB. Both BzTree and Bw-tree are treated as standalone key-value record stores that accept read (`Get`), write (`Insert/Delete/Update/Upsert`), and range scan operations. We vary the amount of read and write operations using four different workload mixtures:

- **Read-Only:** 100% *reads*
- **Read-Mostly:** 90% *reads*, 10% *writes*
- **Read-Heavy:** 75% *reads*, 25% *writes*
- **Balanced:** 50% *reads*, 50% *writes*

By default, the write operations in the workload mixtures are `Upserts`. We configure the distribution of the keys accessed by the index operations to be based on the following distributions:

- **Random:** 64-bit integers from a Uniform distribution.
- **Zipfian:** 64-bit integers from a Zipfian distribution.
- **Monotonic:** 64-bit monotonically increasing integer.

We generate a skewed workload using the Zipfian distribution. Unless mentioned otherwise, our primary performance metric is throughput measured in *operations per second*. We use 48 worker threads equal to the number of logical cores on our experiment machine. We use 8-byte keys and values, and configure the default page size for both BzTree and Bw-tree to be 1 KB. In all our experiments, we prefill the index with 1M records. We observed similar trends when the index is prefilled with ten million records. The BzTree, by default, assumes that keys are variable length and uses the offset field in the record metadata entry to dereference keys (there is no fixed-length optimization). We execute all the workloads three times under each setting and report the average throughput.

7.2 Design Complexity

As minimized complexity is one of our primary goals, we begin by quantifying the BzTree design complexity compared to the Bw-tree. The Bw-tree's latch-free tree algorithms make use of a single-word CAS [22]. Its complexity stems from the fact that multi-word updates temporarily leave the index in an inconsistent state that other threads must detect and handle. The BzTree instead uses PMwCAS to atomically install changes at multiple tree locations, and this reduces its complexity considerably. Consider the node split algorithm: if the node split operation propagates only up to the great-grandparent node, it involves atomic updates to 5 tree locations. With a single-word CAS approach, the developer must explicitly handle many of the 2^5 (32) possible intermediate states that could be exposed to concurrent threads.[7] In contrast, with PMwCAS, the developer only needs to reason about 2 tree states: the initial state with none of the locations mutated, and the final state with all the five locations successfully mutated. More broadly, PMwCAS shrinks the state space associated with mutating k tree locations from 2^k states to k states.

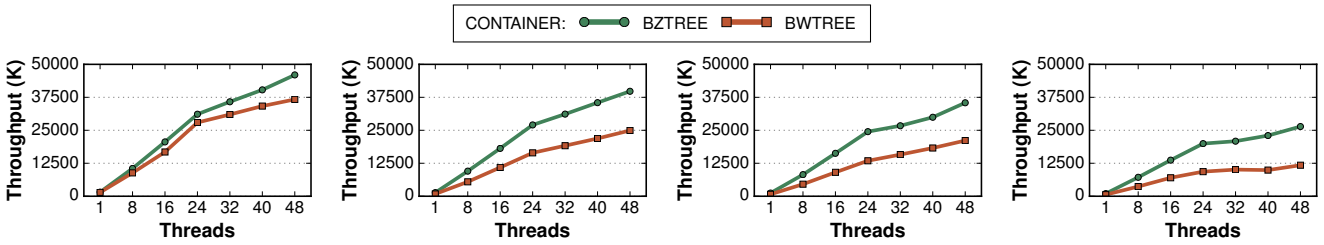
To quantify the reduction in design complexity, we measure the lines of code (LOC) of the node split algorithm in BzTree and Bw-tree. While the Bw-tree implementation contains 750 LOC, the one in BzTree only contains 200 LOC. This is because the BzTree implementation handles fewer tree states. A relative reduction in LOC does not necessarily imply a more maintainable data structure. We therefore measured the *cyclomatic complexity* (CC) of these algorithms. CC is a quantitative measure of the number of linearly independent paths through the function's source code, and represents the function's complexity. Higher values of CC, therefore, correspond to more complex functions that are harder to debug and maintain. CC of the node split algorithms in BzTree and Bw-tree are 7 and 12, respectively. This demonstrates that PMwCAS reduces the design complexity of BzTree's algorithms.

7.3 Runtime Performance

We now provide an analysis of the runtime performance of BzTree compared to the Bw-tree on different workload mixtures and key access distributions. For each configuration, we scale up the number of worker threads. The worker threads process tree operations in a closed loop. These experiments are run with both indexes in volatile DRAM mode to (a) showcase the peak performance of the BzTree (we study durability in the next section) and (b) provide a fair comparison to the Bw-tree since its design targets volatile DRAM with no straightforward extension to NVM.

[6] And update if leafs contain full record payloads.

[7] This is because each tree location can either be updated or not at a given point in time. Although some of these intermediate states might never be observed in practice, we note that the state space grows exponentially.



(a) Read-only Workload **(b)** Read-mostly Workload **(c)** Read-heavy Workload **(d)** Balanced Workload

Figure 4: Random Key Access Distribution – The throughput of BzTree and Bw-tree for the YCSB benchmark with different workload mixtures.

Random Key Access Distribution: We first consider the results on the read-only workload with random key distribution shown in Figure 4a. These results provide an upper bound on the tree's performance because none of the operations modify the tree. The most notable observation from this experiment is that BzTree delivers 28% higher throughput than Bw-tree. This is primarily because BzTree employs raw pointers to inter-link tree nodes meaning readers do not have to use indirection to locate child nodes.

The benefits of BzTree algorithms are more prominent on the write-intensive workloads in Figures 4c and 4d, where BzTree's throughput is 1.7× higher and 2.4× higher than that of Bw-tree, respectively. We attribute this gap to the reduction in algorithm complexity and BzTree's ability to perform in-place updates.

Zipfian Key Access Distribution: Figures 5a and 5b present the throughput of BzTree and Bw-tree on different workloads with the Zipfian key distribution. The benefits of BzTree's reader-friendly algorithms are prominent on the read-only workload, where BzTree outperforms Bw-tree by 33%. By skipping the layer of indirection through the mapping table, readers can traverse BzTree faster than Bw-tree. Unlike Bw-tree, BzTree supports inlined updates in leaf nodes and always keeps the interior nodes consolidated. This reduces pointer chasing, thereby enabling a faster read path.

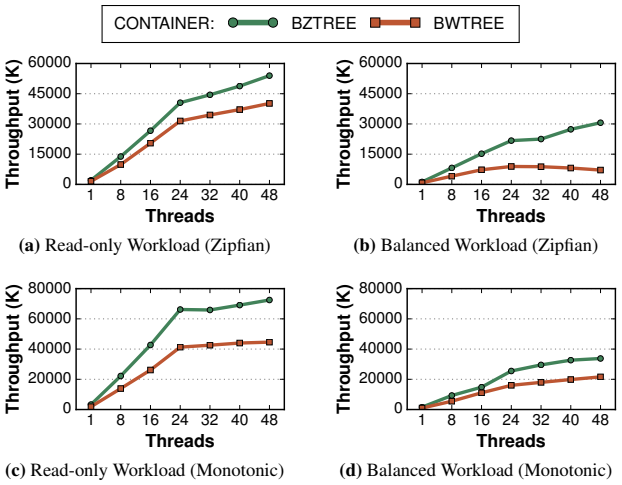
On the balanced workload in Figure 5b, BzTree's throughput is 4.3× higher than that of Bw-tree. Since most of key accesses are directed to a few leaf nodes with the Zipfian key distribution, the in-place update design of BzTree reduces the need for frequent node splits in comparison to Bw-tree. This shrinks the amount of work performed by writers in BzTree, since the split operations take more time to complete compared to single record writes.

Monotonic Key Access Distribution: The performance of Bw-tree and BzTree on workloads with the monotonic key distribution is shown in Figures 5c and 5d. We observe that on the balanced workload, BzTree and Bw-tree deliver 34 M and 21 M operations per second, respectively. The benefits of processor caching are prominent on this workload since keys are monotonically increasing. This is a pathological configuration for concurrent writers, since they always contend on the same node. It is an approximate upper bound on the worst-case behavior of BzTree's latch-free algorithms on write-intensive workloads.

7.4 Durability

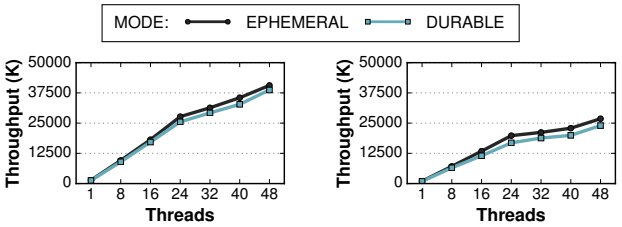
We now examine the cost of persistence by measuring the runtime performance of BzTree in volatile and durable modes on different workload mixtures based on the random key access distribution. As shown in Figures 6a and 6b, the persistence overhead is 5% and 12% on the read-mostly and balanced workloads, respectively. We attribute the small drop in throughput to the overhead of using `PMwCAS` in durable mode as opposed to volatile version [37].

In durable mode, the BzTree additionally uses the `CLFLUSH` instruction to write back the modified tree contents to NVM. Since



(a) Read-only Workload (Zipfian) **(b)** Balanced Workload (Zipfian)
(c) Read-only Workload (Monotonic) **(d)** Balanced Workload (Monotonic)

Figure 5: Zipfian and Monotonic Key Access Distributions – The throughput of BzTree and Bw-tree for the YCSB benchmark.



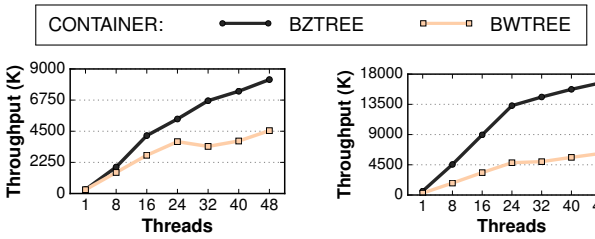
(a) Read-mostly Workload **(b)** Balanced Workload

Figure 6: Cost of Persistence – The throughput of BzTree for the YCSB benchmark in volatile and durable modes across different workload mixtures based on the random key access distribution.

`CLFLUSH` invalidates the line from the cache, this results in compulsory cache miss when the same data is accessed after the line has been flushed. Future processors will support the `CLWB` instruction, which unlike `CLFLUSH`, does not invalidate the line and instead only transitions it to a non-modified state [15]. We expect that such a lightweight cache-line flushing instruction will further increase BzTree's throughput in durable mode by improving its caching behavior. This experiment illustrates that the same BzTree implementation can be used for indexes in both DRAM and NVM with a moderate cost to seamlessly support persistence.

7.5 Scan Performance

We next examine the performance of BzTree and Bw-tree on different workload mixtures of the YCSB benchmark containing range scan operations. In this experiment, we configure the scan predicate's key range so that the scan operation starts from a uniformly



(a) Read-mostly Workload **(b)** Balanced Workload

Figure 7: Scan Performance – The throughput of BzTree and Bw-tree on different workload mixtures containing range scan operations.

random starting offset, and returns at most 10 matching records. The most notable observation from the results shown in Figure 7 is that BzTree scales better than Bw-tree. On the read-mostly workload with range scan operations, as shown in Figure 7a, the BzTree's throughput with 48 worker threads is 28.5× that of its single-threaded performance. In contrast, Bw-tree's throughput with 48 worker threads is 16.4× that of its single-threaded performance. This is mainly due to less pointer chasing and memory accesses in the BzTree. The Bw-tree must always perform delta updates to pages (new memory prepended to a node representing an update), even for 8-byte payload changes. This causes the scan to perform pointer chases over delta chains, e.g., when constructing a page snapshot to service `get-next` requests.

The BzTree's throughput on the read-mostly workload is 1.8× higher than that of Bw-tree. We attribute this to the reduction in indirection overhead of range scan operation, that forms 90% of the read-mostly workload. On the balanced workload, as shown in Figure 7b, the BzTree outperforms Bw-tree by 2.7×. This performance gap is realized by virtue of the reduction in algorithm complexity and BzTree's ability to perform in-place updates, compared to the Bw-tree's usage of delta updates. We observe that the absolute throughput of BzTree on the balanced workload is 2.1× higher than that on the read-mostly workload. This is because range scan is more expensive than the write operation, and the latter operation is more often executed in the balanced workload.

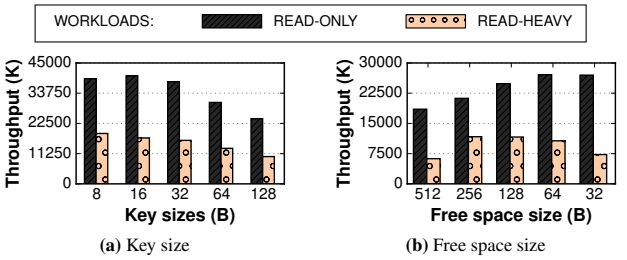
7.6 PMWCAS Statistics

To perform record updates and install structure modifications, the BzTree uses the `PMwCAS` operation with the word count varying from 1 to 3. We analyzed the failure frequency of `PMwCAS` operations in the BzTree across varying degrees of contention on the balanced workload. We observe an increase of 0.02% to 0.12% in the fraction of failed `PMwCAS` operations going from 8 to 48 threads. This is primarily because multiple worker threads concurrently attempt to split the same leaf node and only one thread succeeds. A key takeaway is that on all configurations, the overall fraction of failed `PMwCAS` operations remains less than 0.2%.

7.7 Sensitivity Analysis

We now analyze how the key size and unsorted free space size affects the runtime performance of BzTree on the YCSB benchmark. We ran the read-only and read-heavy workloads based on the random key distribution.

Key Size: In this experiment, we fix the page size to be 1 KB, and vary the key size from 8 B to 128 B. The key observation from the results in Figure 8a is that the throughput drops by 39% when we increase the key size on the read-only workload. We attribute this to more expensive key comparisons in case of longer keys, both in the interior and leaf nodes. The performance impact of key size is more prominent on the read-heavy workload where we observe a throughput drop of 46%. This is because with longer keys, the



(a) Key size **(b)** Free space size

Figure 8: 'Impact of Key Size and Free Space Size – The throughput of BzTree while running the YCSB benchmark under different key size and free space size settings.

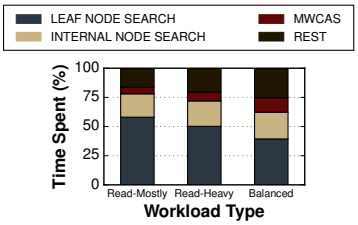


Figure 9: Execution Time Breakdown – The time that BzTree spends in its internal components when running the balanced workload.

leaf nodes are filled faster with fewer keys, and this leads to more frequent node splits that negatively affects throughput.

Free Space Size: Lastly, we examine the impact of the size of the free space on the BzTree's performance. We fix the page size to be 1 KB, and vary the free space size from 512 B to 32 B. The BzTree uses the remaining space in the leaf node to store the sorted keys, as described in Section 4.1. Figure 8b shows that the throughput increases by 45% when we decrease the free space size. This is because readers need to perform fewer key comparisons in leaf nodes as the free space can only contain fewer keys. Reducing the free space in this manner, however, increases the frequency of node split operations and reduces space utilization. This is illustrated on the read-heavy workload, where BzTree delivers its peak throughput when the free space size is 128 B. We attribute this to more node splits and key comparisons under smaller and larger free space size settings, respectively.

7.8 Memory Footprint

We next compare the peak memory footprint of BzTree and Bw-tree data structures while running the balanced workload in the YCSB benchmark. We observe that BzTree's footprint is 1.6× smaller than that of Bw-tree for trees whose sizes range from 100 K to 10 M keys. For instance, when we prefill the index with 10 M keys, the peak memory footprint of BzTree and Bw-tree are 228 MB and 365 MB, respectively. We attribute this to the compact node layout of BzTree and its ability to buffer updates in place.

7.9 Execution Time Breakdown

In this experiment, we analyze the time that BzTree spends in its internal components during execution. We examine the balanced workload in the YCSB benchmark with uniform key access distribution. We use profiling tools available in Windows Server to track the cycles executed within the different components of BzTree [26]. We start this profiling after prefilling the index. The cycles are classified into four categories: (1) leaf node search, (2) internal node search, (3) `PMwCAS`, and (4) other miscellaneous components. This last category includes the time spent in copying data and performing tasks such as garbage collection.

The most notable result for this experiment, as shown in Figure 9, is that even on the balanced workload, BzTree only spends 12% of its time on performing PMwCAS operations. This is because it spends the bulk of the time on traversing the tree and searching the leaf and internal nodes. We observe that the proportion of the time that BzTree spends on searching nodes increases from 61% to 78% when the workload is not write-intensive. This explains why the BzTree optimizations are more beneficial for the balanced workload.

8 Related Work

BzTree's design benefited from prior work on latch-free indexes, persistent indexes, and MWCAS frameworks.

Latch-free Indexes. Modern devices with multi-core processors and high-capacity memory mandate highly concurrent indexes. This gave rise to the design of Bw-tree, a latch-free index built on top of the single-word CAS primitive [22]. Although such an index delivers high performance, it is challenging to design, debug, and extend. The developers must carefully design every SWCAS-based latch-free algorithm so that each atomic action leaves the tree in a valid intermediate state for other threads [8, 33]. We employ the stronger MWCAS primitive to simplify BzTree's design.

Other state-of-the-art data structures include ART and MassTree. ART is a trie-based data structure that employs an adaptive node structure and adopts an optimistic lock coupling synchronization algorithm [21, 20]. MassTree is a hybrid cache-conscious B-tree/trie data structure that eschews traditional latch coupling [25]. Unlike BzTree, its synchronization algorithm relies on clever use of atomic operations and hand-over-hand latching. With this approach, developers need to keep track of the latches being held along different control flow paths in order to release the correct set of locks. This increases the design complexity of the data structure. For example, we found that the cyclomatic complexity of the node split algorithm in MassTree is 19, which is more than two times that of the BzTree complexity of 7 (Section 7.2). In addition, MassTree and ART, by design, do not ensure durability on NVM.

Persistent Indexes. The advent of NVM triggered the development of different persistent indexes [2, 3, 28, 34, 38]. Write atomic B+Tree adopts a redo-only logging algorithm for ensuring durability [3]. In contrast, NV-Tree [38] employs an append-only update strategy and re-constructs internal index nodes during recovery. However, it requires the internal nodes to be stored in consecutive memory blocks. FPTree [28] is a hybrid DRAM-NVM index that keeps the internal nodes in volatile memory and stores the leaf nodes on NVM, requiring the overhead of a partial index rebuild during recovery. It exploits hardware transactional memory and fine-grained locks to handle concurrent internal and leaf node accesses.

The BzTree design differs from these approaches in three ways:

- *The BzTree does not require custom recovery code*. Prior persistent NVM index designs employ sophisticated logging and/or recovery algorithms. These logging algorithms record all the tree updates to persistent storage in order to achieve persistence, and periodically write out checkpoints to speed up recovery. For example, the FPTree's node split algorithm requires the writer to log information about the node being split and the newly allocated leaf node. Depending on when the crash occurs within the algorithm, the writer either rolls the split operation forward or backward. Developing correct logging and recovery algorithms are challenging and contributes to the increased design complexity of these data structures. In addition, existing designs often require the reconstruction of internal index nodes after a system failure. In contrast, BzTree does not require a tree-level recovery algorithm. As discussed in Section 6.3, fixed size-word updates and memory allocations are recoverable through the general-purpose PMwCAS primitive, while "dangling" space allocations within a node are detected using the global index epoch.
- *The BzTree design works seamlessly across both volatile and persistent environments.* To our knowledge, the BzTree is the only index design flexible enough to function and perform well in both environments. As demonstrated in Section 7, the BzTree performs better than a state-of-the art B+Tree in volatile DRAM and takes a modest 8% performance hit to guarantee persistence on NVM.
- *Latch-free design.* The BzTree is the only design that is latch-free and highly concurrent, while also ensuring persistence guarantees on NVM.

Multi-Word CAS. A multi-word CAS instruction (MWCAS) simplifies latch-free programming of high performance data structures as exemplified in BzTree. A general-purpose MWCAS implementation is not available in hardware. Prior work has focused on realizing MWCAS in software using the hardware-provided SWCAS [1, 9, 13, 18]. The PMwCAS library we use is based on the volatile MWCAS primitive proposed by Harris et al. [11]. MWCAS and transactional memory systems are similar in that they require either all or none of the sub-operations to succeed [13]. Software transactional memory systems have limited adoption due to high performance overhead [32]. In contrast, hardware transactional memory (HTM) [39, 19] exhibits lower performance overhead and can help simplify latch-free design. However, HTM suffers from spurious transaction aborts either due to transaction size or because of CPU cache associativity [24]. The PMwCAS library does not use HTM.

9 Conclusion

It is challenging to design, debug, and extend latch-free indexing structures. This is because "traditional" latch-free designs rely on a single-word CAS instruction that requires the developer to carefully stage every atomic action so that each action leaves the tree in an intermediate state that is recognizable to concurrent accessors. Upcoming NVM environments will only make this task more difficult due to durability guarantees and the interplay with volatile CPU caches. With the BzTree design we demonstrate that using PMwCAS, a multi-word compare-and-swap with durability guarantees, helps reduce index design complexity tremendously. Our experimental evaluation shows that even though PMwCAS is computationally more expensive than a hardware-based single-word CAS, the simplicity that we gain by using PMwCAS improves not only the maintainability but also the performance of the BzTree. A cyclomatic complexity analysis shows that the BzTree is at least half as complex as state-of-the-art main-memory index designs (the Bw-tree and MassTree). Another benefit of the BzTree design is its flexibility: the same design can be used on both volatile DRAM and NVM, with a roughly 8% overhead to ensure persistence. In addition, existing B+Tree implementations that achieve durability on NVM often employ complex algorithms for ensuring recoverability. The BzTree, on the other hand, does not rely on custom recovery techniques: it simply relies on the general-purpose PMwCAS to roll forward (or back) its in-flight word modifications before becoming online and active. This allows for near-instantaneous recovery of the BzTree index and is a defining feature of its design.

Acknowledgements

We would like to thank Donald Kossmann, Phil Bernstein, and David Lomet for their feedback that helped improve this work.

10 References

[1] J. H. Anderson, S. Ramamurthy, and R. Jain. Implementing wait-free objects on priority-based systems. *PODC*, pages 229–238, 1997.

[2] S. Chen, P. B. Gibbons, and S. Nath. Rethinking database algorithms for phase change memory. In *CIDR*, pages 21–31, 2011.

[3] S. Chen and Q. Jin. Persistent b+-trees in non-volatile main memory. *PVLDB*, 8(7):786–797, 2015.

[4] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, pages 143–154, 2010.

[5] R. Crooke and M. Durcan. A revolutionary breakthrough in memory technology. *Intel 3D XPoint launch keynote*, 2015.

[6] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL Server's Memory-optimized OLTP Engine. In *SIGMOD*, pages 1243–1254, 2013.

[7] Everspin. DDR3 DRAM compatible MRAM: Spin torque technology. `https://www.everspin.com/ddr3-dram-compatible-mram-spin-torque-technology-0`, 2017.

[8] K. Fraser. *Practical lock-freedom*. PhD thesis, University of Cambridge, 2004.

[9] M. Greenwald. Two-handed emulation: How to build non-blocking implementations of complex data-structures using DCAS. *PODC*, pages 260–269, 2002.

[10] J. Handy. Understanding the Intel/Micron 3D XPoint memory. `http://www.snia.org/sites/default/files/SDC15_presentations/persistant_mem/JimHandy_Understanding_the-Intel.pdf`, 2015. Storage Developer Conference.

[11] T. L. Harris, K. Fraser, and I. A. Pratt. A practical multi-word compare-and-swap operation. *DISC*, pages 265–279, 2002.

[12] J. M. Hellerstein, M. Stonebraker, and J. R. Hamilton. Architecture of a Database System. *Foundations and Trends in Databases*, 1(2):141–259, 2007.

[13] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM TOPLAS*, 15(5):745–770, Nov. 1993.

[14] M. Hosomi et al. A novel nonvolatile memory with spin torque transfer magnetization switching: spin-ram. *IEEE International Electron Devices Meeting (IEDM)*, pages 459–462, 2005.

[15] Intel. Intel Architecture Instruction Set Extensions Programming Reference. `https://software.intel.com/sites/default/files/managed/0d/53/319433-022.pdf`, 2017.

[16] Intel Corporation. Intel®64 and IA-32 architectures software developer's manuals. 2016.

[17] Intel Corporation. NVM library. `http://www.pmem.io`, 2016.

[18] A. Israeli and L. Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. *PODC*, pages 151–160, 1994.

[19] C. Jacobi, T. Slegel, and D. Greiner. Transactional memory architecture and implementation for IBM System Z. *MICRO*, pages 25–36, 2012.

[20] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: ARTful indexing for main-memory databases. *ICDE*, pages 38–49, 2013.

[21] V. Leis, F. Scheibner, A. Kemper, and T. Neumann. The ART of practical synchronization. *DaMoN*, pages 3:1–3:8, 2016.

[22] J. Levandoski, D. Lomet, and S. Sengupta. The Bw-Tree: A B-tree for new hardware platforms. *ICDE*, pages 302–313, 2013.

[23] J. Levandoski, D. B. Lomet, S. Sengupta, R. Stutsman, and R. Wang. High performance transactions in deuteronomy. In *CIDR*, 2015.

[24] D. Makreshanski, J. Levandoski, and R. Stutsman. To lock, swap, or elide: On the interplay of hardware transactional memory and lock-free indexing. *PVLDB*, 8(11):1298–1309, 2015.

[25] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. *EuroSys*, pages 183–196, 2012.

[26] Microsoft. Profiling Tools. `https://docs.microsoft.com/en-us/visualstudio/profiling/profiling-feature-tour`.

[27] Netlist. Netlist storage class memory: `http://www.netlist.com/products/Storage-Class-Memory/HybriDIMM/default.aspx`, 2017.

[28] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner. FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory. *SIGMOD*, pages 371–386, 2016.

[29] I. Oukid, W. Lehner, T. Kissinger, T. Willhalm, and P. Bumbulis. Instant recovery for main memory databases. In *CIDR*, 2015.

[30] A. Prout. The Story Behind MemSQL's Skiplist Indexes. `http://blog.memsql.com/the-story-behind-memsqls-skiplist-indexes/`, 2014.

[31] D. Schwalb, T. Berning, M. Faust, M. Dreseler, and H. Plattner. nvm malloc: Memory allocation for NVRAM. In *ADMS*, pages 61–72, 2015.

[32] N. Shavit and D. Touitou. Software transactional memory. *PODC*, pages 204–213, 1995.

[33] H. Sundell and P. Tsigas. Lock-free deques and doubly linked lists. *JPDC*, 68(7):1008–1020, July 2008.

[34] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *FAST*, pages 61–75, 2011.

[35] Viking. Viking technology memory and storage: `http://www.vikingtechnology.com/uploads/embedded_overview.pdf`, 2017.

[36] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: lightweight persistent memory. In R. Gupta and T. C. Mowry, editors, *ASPLOS*, pages 91–104. ACM, 2011.

[37] T. Wang, J. Levandoski, and P. A. Larson. Easy lock-free indexing in non-volatile memory. Technical report, Microsoft Research, 2017.

[38] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He. NV-Tree: Reducing consistency cost for NVM-based single level systems. *FAST*, pages 167–181, 2015.

[39] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance evaluation of Intel® transactional synchronization extensions for high-performance computing. *SC*, pages 19:1–19:11, 2013.