Elimination (a,b)-trees with fast, durable updates

Anubhav Srivastava
anubhav.srivastava@uwaterloo.ca
University of Waterloo
Waterloo, Canada

Trevor Brown*
trevor.brown@uwaterloo.ca
University of Waterloo
Waterloo, Canada

Abstract

Many concurrent dictionary implementations are designed and optimized for read-mostly workloads with uniformly distributed keys, and often perform poorly on update-heavy workloads. In this work, we first present a concurrent (a,b)-tree, the OCC-ABtree, which outperforms its fastest competitor by up to 2x on uniform update-heavy workloads, and is competitive on other workloads. We then turn our attention to *skewed* update-heavy workloads (which feature many inserts/deletes on the same key) and introduce the Elim-ABtree, which features a new optimization called publishing elimination. In publishing elimination, concurrent inserts and deletes to a key are reordered to *eliminate* them. This reduces the number of writes in the data structure. The Elim-ABtree achieves up to 2.5x the performance of its fastest competitor (including the OCC-ABtree). The OCC-ABtree and Elim-ABtree are linearizable. We also introduce durable linearizable versions[1] for systems with Intel Optane DCPMM non-volatile main memory that are nearly as fast.

CCS Concepts: • **Theory of computation → Concurrent algorithms**.

Keywords: Concurrent data structures, optimistic concurrency, elimination, B-trees

1 Introduction

The (ordered) dictionary is one of the most fundamental abstract data types. It stores a set of keys, each of which has an associated value, and provides operations to insert a key and value, remove a key, and find the value associated with a key. Sometimes dictionaries also support predecessor, successor, and range query operations.

Concurrent dictionary implementations in the literature typically focus on maximizing performance under low contention read-mostly workloads, with less attention paid to performance under update-heavy workloads and high contention workloads. In this paper, we study the question of how to scale these challenging workloads, ideally without sacrificing performance in the read-mostly workload. Update-heavy workloads are particularly difficult to scale when there is a lot of memory contention. To generate high contention, we study Zipfian access distributions, in which the frequency of a key being accessed is inversely proportional to its rank. That is, the kth most frequent key is requested with probability proportional to $1/k^s$, where s is a parameter controlling the skew of the distribution.

The advantages of concurrent B-trees over binary search trees, including better cache locality, are well known. Our new data structures presented in this paper are (a,b)-trees, which are a variant of B-trees that allow between a and b keys per node (for $a \leq b/2$). Our trees are based on the (non-concurrent) *relaxed* (a,b)-tree of Larsen and Fagerberg [35]. Relaxed (a,b)-trees are more concurrency friendly than B-trees. They break insert and delete operations, and any subsequent rebalancing, into multiple *sub-operations* (each of which modifies at most four nodes). As long as each *sub-operation* is atomic, the relaxed (a,b)-tree's structure and balance properties are maintained. Implementing these sub-operations atomically requires less synchronization than implementing traditional (sequential) B-tree operations atomically, since B-tree operations must sometimes rebalance an entire root-to-leaf path.

Relaxed (a,b)-trees have been implemented in a concurrent setting before [5, 11, 12], but the overheads of existing implementations are high, and they perform poorly in update-heavy workloads. Our first new data structure, an optimistic concurrency control (a,b)-tree (OCC-ABtree), uses mostly known techniques to avoid the main sources of overhead in those implementations: unnecessary node copying and key sorting in leaves, and various overheads introduced by lock-free synchronization primitives.

The main challenge of creating a *concurrent* relaxed (a,b)-tree is guaranteeing that sub-operations occur atomically, and that searches are correct. The OCC-ABtree uses fine-grained versioned locks to achieve the former, and version based validation in leaf nodes for the latter. This locking technique is somewhat similar to OPTIK [22] and the optimistic

*Corresponding author
[1]Technically, we show that our data structures satisfy a stronger correctness condition called *strict linearizability* [3].

PPoPP '22, April 2–6, 2022, Seoul, Republic of Korea

ACM ISBN 978-1-4503-9204-4/22/04.
https://doi.org/10.1145/3503221.3508441

validation of the AVL tree of Bronson et al. [10]. As our experiments show, the OCC-ABtree outperforms many state-of-the-art data structures on *both* read-mostly and update-heavy workloads. However, like its competitors, its performance degrades as contention increases.

To optimize for high-contention workloads, we take inspiration from another data structure that tackles extremely high contention: *elimination stacks* [32]. In an elimination stack, whenever a thread experiences contention while accessing the stack, it attempts to synchronize *directly* with another thread performing the opposite operation (push/pop) to complete both operations *without* accessing the stack.

Our second new data structure, the Elim-ABtree, uses a new type of elimination called *publishing elimination*. This is a primary contribution in this work. In publishing elimination, threads that modify a leaf place a *record* of their modification in the leaf itself. Other threads can then use this record to return from their operation without having to modify the data structure. In traditional elimination, *pairs* of threads rendezvous and eliminate each others' operations. In publishing elimination, *many* threads can use a *single record* in a leaf to eliminate their own operations. The Elim-ABtree is significantly faster than the OCC-ABtree (and prior work) in high contention workloads.

Publishing elimination is especially enticing in systems with Intel Optane (DCPMM) persistent main memory, because fewer *flushes* and high-latency *fence* instructions are needed. We present durably linearizable [34] implementations of the OCC-ABtree and Elim-ABtree. This requires minor modifications to the code to add flushing and fencing as appropriate to ensure that each update appears to occur atomically in persistent memory. The resulting persistent trees are only slightly slower than their volatile counterparts, offering persistence at nearly the speed of in-memory computing.

Contributions. (1) We present two novel algorithms: OCC-ABtree and Elim-ABtree which outperform the state-of-the-art in many workloads. (2) We introduce a novel publishing elimination algorithm that is optimized for our data structures. (3) We add persistence to our trees, and present experiments that show the overhead of persistence is low. (4) Our algorithms have strong theoretical properties: deadlock-freedom, linearizability (for the volatile trees) and durable linearizability (for the persistent trees), and can be modified to guarantee logarithmic height bounds with some overhead.

2 Related work

We briefly survey the state-of-the-art in concurrent ordered dictionaries and contrast with our techniques. We experimentally compare with the bolded data structures.

Binary search trees. Ellen et al. [26] introduced the first lock-free external BST. Searches are implemented the same way as in a sequential BST. An update operation searches for a target node to modify, then synchronizes by *flagging* or *marking* nodes to indicate that they will be modified. Other updates that encounter these flags or marks will *help* the operation complete, guaranteeing lock-free progress. Natarajan and Mittal [45] improved upon this design by flagging/-marking *edges* instead of nodes, and reducing the amount of memory allocated per update operation (**NM14**).

Bronson et al. [10] propose a partially external balanced BST (**BCCO10**) that uses optimistic concurrency control to synchronize threads. They introduce a complex hand-over-hand version number based validation technique to implement fast searches. Our synchronization technique for updates is somewhat similar to BCCO10, but our searches avoid the complexity of Bronson's hand-over-hand validation transactions. BCCO10 has previously been shown to be the fastest concurrent BST in search-dominated workloads [6].

David et al. [22] propose a set of rules for optimizing concurrent data structures. They apply these rules to design a straightforward, efficient lock-based external BST (**DGT15**).

Brown et al. [13] introduced wait-free synchronization primitives (`LLX` and `SCX`), used them to implement a *template* for lock-free trees, and used the template to produce a (balanced) chromatic tree [14]. Several other concurrent BST algorithms have also been proposed [33, 46, 49].

B-tree variants. Brown used the aforementioned template to design a lock-free (a,b)-tree (**LF-ABtree**) [11], based on the same relaxed (a,b)-tree as our OCC-ABtree [35]. Update operations take a read-copy-update approach: inserting or deleting a key involves replacing a tree node with a new copy. The LF-ABtree has been shown to be substantially faster than NM14 and BCCO10, which are among the fastest BSTs [15]. As our experiments show, our trees significantly outperform the LF-ABtree in many workloads.

Braginsky and Petrank introduced the first lock-free, linearizable B+tree [9]. Each node contains a lock-free linked list of entries, implemented using arrays. Each entry is a key-value pair stored in the same word.

The Bw-Tree is a lock-free variant of a B+tree that is designed to achieve high performance under realistic workloads [40]. Many of the design decisions made in the Bw-Tree are focused on workloads that do not fit in memory, and incur significant overhead. Our experiments include an optimized variant of the Bw-Tree called the **OpenBw-Tree** [56].

The BzTree [7] simplifies the implementation of the Bw-Tree by using a multi-word compare-and-swap (MwCAS), and results in the paper suggest it is faster than the BwTree. Guerraroui et al. introduced a faster MwCAS algorithm and used it to accelerate the BzTree. The BzTree can also be made persistent by using a persistent MwCAS.

Concurrent tries. Tries are an alternative to B-trees for implementing concurrent (ordered) dictionaries. The Masstree [43] and the Adaptive Radix Tree with optimistic lock coupling

(ART) [37, 38] both use optimistic concurrency control techniques. In both, operations are accelerated using SIMD instructions. However, they are not strictly comparison-based, and they require the programmer to serialize keys to be binary comparable. This extra data marshalling is tedious and can add overhead. Additionally, the shape and depth of the trees are determined by the key distribution, not by the number of keys they contain.[2] We compare with ART with optimistic lock coupling in an extended technical report [53]. ART with optimistic lock coupling has been shown to be faster than Masstree [38].

Distribution/contention aware data structures. There has been also some work on data structures that are designed to accommodate non-uniform distributions. The concurrent interpolation search tree (**C-IST**) of Brown et al. [15] provides doubly-logarithmic runtime for smooth distributions. However, its updates are slow.

The splay tree [52] is a popular sequential data structure that adapts to non-uniform distributions. After searching for a key, the splay tree performs rotations to move the node containing the key to the root. This reduces future access time for searches on the same key but also introduces a point of contention at the root, which makes the splay tree unsuitable for concurrent use. The **CBTree** [2] is a concurrent splay tree-like data structure which uses counting to perform splaying only after a significant number of searches/updates have accessed a node, effectively amortizing the cost of the splay over many operations.

The **Splay-List** [4] is a concurrent variant of a SkipList [48] that splays by increasing the height of frequently-accessed keys. Like the CBTree, it uses a counter-based approach to amortize the cost of the splaying.

The contention adapting search tree (**CATree**) [50] is a variant of an external search tree with binary internal nodes. Each external node is a sequential dictionary data structure, protected by a lock. AVL trees were used as the CATree's sequential dictionary in the authors' experiments (as well as our own). The authors approximate contention at each external node by measuring how often a lock is already acquired when a thread attempts to acquire it. When sufficient contention is detected at a node, the sequential data structure is split into two and an internal node connecting them is linked into the tree. Similarly, two adjacent sequential data structures are combined if neither is under contention.

General approaches. There are several universal constructions for transforming sequential data structures into concurrent ones. These come in lock-based, lock-free, wait-free, and even NUMA-aware variants [18, 27, 28]. Though they are simple to use, these constructions either require a copy of the data structure per thread or NUMA node (which is not practical for large data structures) or have a single global bottleneck on updates (e.g. an update log or state object).

Transactional memory makes it relatively easy to produce concurrent implementations of data structures, but it has significant drawbacks. Hardware transactional memory (HTM) is not universally available, and software transactional memory (STM) adds substantial overhead. Furthermore, transactional memory is optimized for low-contention workloads. In the high-contention scenarios we study in this paper, almost all transactions would abort (or serialize) because of data conflicts. We performed some formative experiments comparing our trees with analogous trees implemented using HTM, STM, and a hybrid of the two (HyTM, [20]), and found that, while the fastest of these implementations was close in performance to our trees under very low contention, performance degraded drastically under high contention. We omitted these experiments, as they are only tangentially related to this work.

Elimination. Elimination was first introduced for use in concurrent stacks by Shavit and Touitou in [51], but this implementation was not linearizable. The first linearizable implementation of elimination in stacks was provided by Hendler et al. [32]. Hendler et al. coordinate the threads using an elimination array that stores ongoing operations' descriptors. Without loss of generality, suppose a thread t is performing a push. t first attempts to modify the data structure directly. If it fails, t selects a random slot in the elimination array. If this slot contains a descriptor for a pop, t attempts to eliminate both operations. Otherwise, if the slot is empty, it writes its own descriptor and waits a set amount of time to be eliminated. Note that it is possible for multiple push-pop pairs to be eliminated at once (at different indices in the elimination array). This is key to the scalability of the algorithm. Braginsky et al. applied a similar approach to achieve elimination in priority queues [8].

Combining. A different approach to tackling high contention workload is combining, in which a *combiner* thread aggregates and performs the operations of many concurrent threads on the data structure. Drachsler-Cohen and Petrank provide an insightful summary of combining techniques [25]. Flat combining [31] is one of the most popular techniques. In flat combining, each thread attempting to update the data structure adds a record of its operation to a global list. Threads compete to become the combiner by acquiring a global lock. The combiner scans the entire list of operations, then performs them in some order.

Flat combining introduces higher latency compared to our publishing elimination technique. Threads must *wait* for the combiner to complete their operations one-by-one, and the wait can be quite long for operations near the end of the list.

Recently, Drachsler-Cohen and Petrank created a variant of flat combining called local combining on-demand and

[2]Height bounds in a trie are logarithmic in the size of the *universe*. Even with path compression, some key distributions can result in deep tries.

demonstrated it on a linked list [25]. They perform flat combining at each node in the list. We tested our trees with a similar technique: We augmented each leaf node with an MCS queue [44] and used the queues to perform flat combining. We found that this approach was much slower than our publishing elimination technique, in which threads do not have to wait for a combiner.

Persistent concurrent trees. Venkataraman et al. introduced the CDDS-tree, a persistent concurrent B-tree. However, the pseudocode contains a global version number which is a scalability bottleneck [54]. Yang et al. created the NV-Tree, a persistent B-link tree that outperforms the CDDS B-tree by up to 12x [58]. The NV-Tree rebuilds *all* of its internal nodes if *any* internal node becomes too full. This can be extremely slow for large trees but occurs less than 1% of the time in their workloads. Additionally, the NV-Tree only persists leaf nodes since the entire tree can be recovered from them after a crash. This makes the recovery procedure slow, but avoids some flushes during normal execution.

The **FPTree** is another persistent concurrent B-tree [47]. It includes a number of optimizations that make it scale better than the NV-Tree. Each leaf node includes a one-byte hash of each of its keys, known as a fingerprint. The fingerprints are scanned prior to probing the keys themselves, which limits the average number of key comparisons to 1. This can have a large impact when key comparisons are costly (for example, if the keys are strings). Like the NV-Tree, the FPTree uses unsorted leaves and only persists leaf nodes.

The **RNTree** is a persistent concurrent B+tree that uses transactional memory and an indirection array with pointers to key-value pairs in each leaf node [41]. The indirection array makes binary searching for a key possible, with the drawback that inserts might require shifting every key-value pointer in the indirection array.

Finally, there are a number of general transformations for making concurrent dictionaries persistent.

RECIPE [36] provides general advice on how to make three categories of data structures persistent: those whose updates occur atomically, those whose updates fix inconsistent state, and those whose updates do not fix inconsistent state. The OCC-ABtree is closest to the third category. RECIPE offers only a vague idea of how one might transform such a data structure. In particular they instruct the data structure designer to: "Add [a] mechanism to allow [updates] to detect permanent inconsistencies. Add [a] helper mechanism to allow [updates] to fix inconsistencies." Both of these seem to require deep knowledge of the data structure. They also introduce fences after each store, whereas we carefully avoid fences where possible in the OCC-ABtree.

The transformations in NVTraverse [29] and Mirror [30] both provide durable linearizability, but target non-blocking data structures (and so are not applicable to the trees in this paper). Montage [57] is another transformation which guarantees a weaker correctness condition known as buffered durable linearizability.

3 OCC-ABtree

Semantics. The OCC-ABtree implements the following *dictionary* operations.

- `find(k)`: If a key-value pair with key `k` is present, return the associated value. Otherwise, return ⊥.
- `insert(k, v)`: If a key-value pair with key `k` is present, return the associated value. Otherwise, insert the key-value pair `<k,v>` and return ⊥.
- `delete(k)`: If a key-value pair with key `k` is present, delete it and return the associated value. Otherwise, return ⊥.

Range queries for the trees we present could be added using the techniques described in [5].

The OCC-ABtree consists of an `entry` pointer to a *sentinel node* that is never removed. This sentinel node has no keys and just one child pointer, which points to the root of the tree. The pseudocode for the data structures used in the OCC-ABtree and selected operations are presented below.

3.1 Data structures

The OCC-ABtree has three types of nodes: leaf nodes, internal nodes and tagged internal nodes. Leaf nodes store keys and values in their `keys` and `vals` arrays. We say an entry in the `keys` array is **empty** if it is ⊥. An empty key has no associated value. The keys in a leaf are *unsorted* and there can be empty entries between keys. This results in much faster updates since inserts and deletes do not need to shift other keys in the node.

Internal nodes contain k child pointers (between 2 and 11, in our implementation), and $k-1$ *routing* keys (that are used to guide searches to the appropriate leaf) in a *sorted* array. Once an internal node is created, its routing keys are *never* changed, but its child pointers can change. To add or remove a key in an internal node, one must *replace* the internal node. This happens relatively infrequently.

A `TaggedInternal` node (or simply **tagged node**) conceptually represents a temporary height imbalance in the tree. A tagged node is created when a key/value must be inserted into a node but the node is full. The node is split, and the two halves are joined by a tagged node. Tagged nodes are not part of any other operation, and thus always have exactly two children. Tagged nodes are eventually removed from the tree when the `fixTagged` rebalancing step is invoked.

Each node has a `lock` field. We use MCS locks as our lock implementation [23, 44]. In MCS locks, threads waiting for the lock join a queue and spin on a *local* bit (meaning they scale well across multiple NUMA nodes). In our trees, a thread only modifies a node if it holds its lock. Leaf nodes have an additional version field, `ver`, that records how many times the leaf has changed and whether it is currently being

```
// K is key type, V is value type
abstract type Node
  keys      : K[MAX_SIZE]
  lock      : MCSLock
  size      : int
  marked    : bool

type Leaf inherits Node
  vals      : V[MAX_SIZE]
  ver       : int

type Internal inherits Node
  ptrs      : Node[MAX_SIZE]

type TaggedInternal inherits Internal

// The result of a search
type PathInfo
  gp     : Node // grandparent
  p      : Node // parent
  pIdx : int    // index of parent in grandparent
  n      : Node // node
  nIdx : int    // index of node in parent

type RetCode is SUCCESS or FAILURE or RETRY

// Sentinel node: points to root
entry    : Internal
MIN_SIZE = 2, MAX_SIZE = 11
```

Figure 1. OCC-ABtree data structures

```
<RetCode, V> searchLeaf(leaf, key)
RETRY:
  ver1 = leaf.ver
  if ver1 is odd
    goto RETRY

  val = ⊥
  for keyIndex = 0 up to MAX_SIZE - 1
    if leaf.keys[keyIndex] = key
      val = leaf.vals[keyIndex]
      break
  ver2 = leaf.ver
  if ver1 ≠ ver2 goto RETRY
  if val = ⊥ return <FAILURE, ⊥>
  else return <SUCCESS, val>

PathInfo search(key, target)
  gp = NULL, p = NULL, pIdx = 0, n = entry, nIdx = 0
  while n is not Leaf
    if n = target break
    gp = p, p = n, pIdx = nIdx, nIdx = 0
    while nIdx < node.size-1 and key ≥ node.keys[nIdx]
      nIdx++
    n = n.ptrs[nIdx]
  return PathInfo(gp, p, pIdx, n, nIdx)

V find(key)
  path = search(key, NULL)
  rc, val = searchLeaf(path.n, key)
  return val
```

Figure 2. OCC-ABtree search operations

changed. After acquiring a leaf's lock, a thread increments the version before making any changes to the leaf and increments the version again once it has completed its changes, and finally releases the lock. Thus the version is even if the leaf is not being modified and odd if it is being modified. The version is used by searches to determine whether any modifications occurred while reading the keys of a leaf[3].

Nodes also contain a `marked` bit, which is set when a node is unlinked from the tree so that updates can easily tell whether a node is in the tree. Marked nodes are never unmarked.

The `PathInfo` structure is returned by `search` and contains the node at which the search terminated, the node's parent and grandparent, the index of the node in the parent's `ptrs` array, and the index of the parent in the grandparent's `ptrs` array.

3.2 Operations

All operations invoke a common `search` procedure, which takes a `key` and optionally a target node as its arguments, and searches the tree, starting at the root, looking for `key`. At each internal node, `search` determines which child pointer it should follow by traversing the (sorted) routing keys sequentially. Once `search` reaches a leaf (or the target node), it returns a `PathInfo` object as described in Section 3.1.

`searchLeaf` is similar to the classical double-collect snapshot algorithm [1]. It reads the leaf's version, reads its keys and values, then re-reads the leaf's version to verify that the leaf did not change while the keys and values were being read. If the leaf *did* change, then `searchLeaf` retries. If the key is found, `searchLeaf` returns `<SUCCESS, val>`, otherwise, it returns `<FAILURE, ⊥>`. Note that `search` and `searchLeaf` do not acquire locks. This allows for greater concurrency since internal nodes can be updated *while* searches are traversing through them.

The `find(key)` operation simply invokes `search` and `searchLeaf`, and returns `val`. `find` operations in the OCC-ABtree never have to restart, unlike in other trees.

Delete. The update operations are perhaps best understood with reference to Figure 3. In a `delete(key)` operation, a thread first invokes `search(key, target)` and `searchLeaf`. If it does not find `key`, then `delete` returns ⊥. Otherwise, it locks the leaf and deletes the key by setting it to ⊥, and returns the associated value (Figure 3(1)). If `key` was deleted by another thread between `search` and acquiring the lock, `delete` returns ⊥. If deleting the key makes the node smaller than the minimum size a, `delete` invokes `fixUnderfull` to remove the underfull node by merging it with a sibling (Figure 3(2)).

Insert. In an `insert(key, val)` operation, a thread first invokes `search(key, target)` and `searchLeaf`. If it finds the `key`, then `insert` returns the associated value (Figure 3(3)). Otherwise, it locks the leaf and tries to insert `key` (resp., `val`) into an empty slot in the `keys` (resp., `vals`) array. We call this case a **simple insert**. If there is no empty slot, `insert` locks the leaf's parent and *replaces* the pointer to the leaf with a pointer to a new tagged node whose two (newly-created) children contain the leaf's old contents and the inserted key-value pair (Figure 3(4)). We call this case a **splitting insert**.

[3] A leaf's version field could hypothetically wrap around and cause an ABA problem, but at 100 million updates per second, this would take 2900 years for a 64 bit word size.

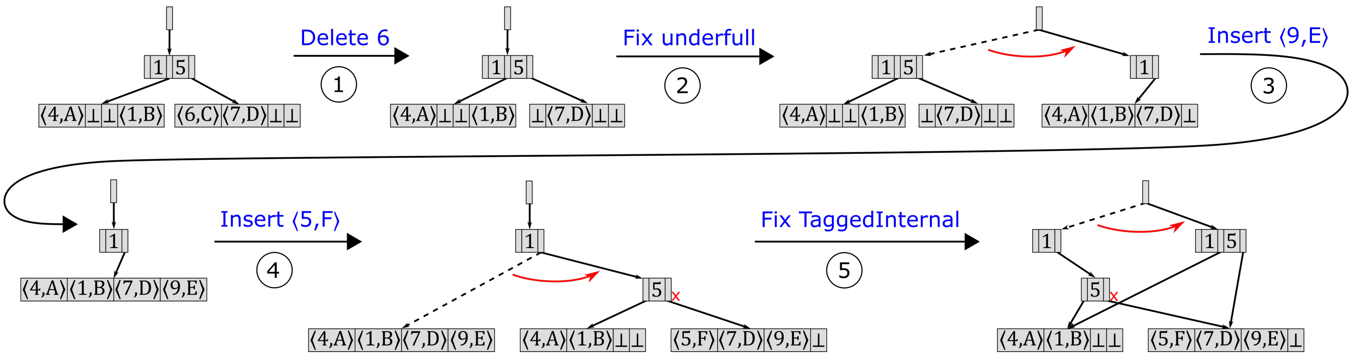

Figure 3. An OCC-ABtree with $a = 2, b = 4$. (1) The key-value pair ⟨6,C⟩ is deleted. This creates an underfull node. (2) The underfull node is merged with its sibling. This leaves the parent underfull, but the parent is the root, which is allowed to remain underfull. (3) ⟨9,E⟩ is inserted into an empty slot (*simple* insert). (4) No empty slot exists for ⟨5,F⟩, so the appropriate leaf is split and a `TaggedInternal` node is created (*splitting* insert). (5) The `TaggedInternal` node is conceptually merged into its parent. We implement this by replacing it with a *new* `Internal` node.

```
V insert(key, val)
RETRY:
  path = search(key, NULL)
  rc, val = searchLeaf(path.n, key)
  if rc = SUCCESS return val

  leaf, parent = path.n, path.p

  Lock leaf
  if leaf.marked
    Unlock leaf and goto RETRY

  // Verify key is not present
  for i = 0 to DEGREE - 1
    if leaf.keys[i] = key
      Unlock leaf and return leaf.vals[i]

  if leaf.size < MAX_SIZE
    // Insert without splitting
    for i = 0 to DEGREE - 1
      if leaf.keys[i] = ⊥
        leaf.ver++          // Start modification
        leaf.keys[i] = key
        leaf.vals[i] = val
        leaf.size++
        leaf.ver++          // End modification
        Unlock leaf and return ⊥
  else
    Lock parent
    if parent.marked
      Unlock leaf and parent and goto RETRY

    N = {contents of leaf} ∪ {key/val}
    newLeaf = TaggedInternal with two children that
         evenly share N
    parent.ptrs[path.nIdx] = newLeaf
    node.marked = true
    Unlock leaf and parent
    fixTagged(newLeaf)
    return ⊥
```

Figure 4. OCC-ABtree insert operation

The pointer change, and hence the insert of `key`, is atomic. The insert then invokes `fixTagged` to get remove the tagged node (Figure 3(5)).

Rebalancing. `fixTagged` attempts to remove a tagged node. It first searches for the tagged node, returning if it is unable to find it. (This case only occurs if another thread has already removed the tagged node). If `fixTagged` finds the target node, it tries to get rid of it by creating a copy c of its parent,

```
V delete(key)
RETRY:
  path = search(key, NULL)
  rc, val = searchLeaf(path.n, key)
  if rc = FAILURE
    return ⊥

  leaf = path.n
  Lock leaf
  if leaf is marked
    goto RETRY

  for i = 0 to DEGREE - 1
    if leaf.keys[i] = key
      deletedVal = leaf.vals[i]
      leaf.ver++        // Start modification
      leaf.keys[i] = ⊥
      leaf.size--
      leaf.ver++        // End modification

      if leaf.size < MIN_SIZE
        Unlock leaf
        fixUnderfull(leaf)
  return ⊥
```

Figure 5. OCC-ABtree delete operation

with the tagged node's key and children merged into c, and changing the grandparent to point to c (Figure 3(5)). However, if the merged node would be larger than the maximum allowed size, `fixTagged` instead creates a new node p with two new children, which evenly share the contents of the old tagged node and its parent (Figure 6). The grandparent is then changed to point to p. (p is a tagged node, unless it is the new root, in which case it is simply an internal node).

Now we turn to `fixUnderfull`. `fixUnderfull` fixes a node n which is smaller than the minimum size, unless n is the root/entry node. It does this by either distributing keys evenly between n and its sibling s if doing so does not make one of the new nodes underfull (Figure 8). Otherwise, `fixUnderfull` merges n with s (Figure 3(2)). In this case, the merged node might still be underfull or the parent node might be underfull (if it was of the minimum size before merging its children). Thus, `fixUnderfull` is called on the merged node and its parent. `fixUnderfull` requires

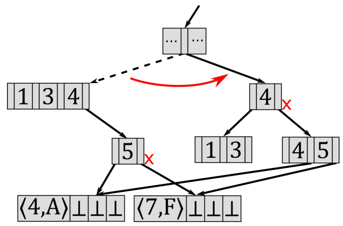

Figure 6. `fixTagged` *split* case (*merge* is in Figure 3)

```
fixTagged(node)
RETRY:
  if node.marked return
  path = search(node.searchKey, node)
  if path.n ≠ node return

  Lock path.n, path.p, and path.gp
  if node, parent or gParent is marked or
     path.p is TaggedInternal
    Release all locks and goto RETRY

  node.marked = true
  path.p.marked = true
  if path.p.size + 1 ≤ MAX_SIZE
    newNode = new Internal containing the keys &
         pointers of node and parent
    path.gp.ptrs[path.pIdx] = newNode
    Release all locks
  else
    // newNode is a TaggedInternal, unless it will be
         the new root (in which case it is Internal)
    newNode = new subtree of three nodes consisting of
         a new Internal that points to two new internal
          nodes which evenly share the keys & pointers
         of node and parent (except for the pointer to
         node)
    path.gp.ptrs[path.pIdx] = newNode
    Release all locks
      fixTagged(newNode)
```

Figure 7. OCC-ABtree fixTagged rebalancing step

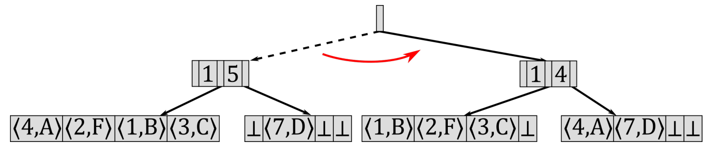

Figure 8. `fixUnderfull` *distribute* case (*merge* is in Figure 3)

```
fixUnderfull(node)
  if node = entry or node = entry.ptrs[0] return

RETRY:
  path = search(node.searchKey, node)
  if path.n ≠ node return

  if path.nIdx = 0
    sIndex = 1 // Sibling is right child
  else
    sIndex = path.nIdx - 1 //
  sibling = parent.ptrs[sIndex]

  Lock node, sibling, path.p, and path.gp
  if node.size ≥ MIN_SIZE return
  if parent.size < MIN_SIZE or
     node, sibling, parent, or gParent is marked or
     node, sibling or parent is TaggedInternal
    Release all locks and goto RETRY

  if node.size + sibling.size ≤ 2 * MIN_SIZE
    newNode, sibling = Distribute keys of node and
         sibling evenly amongst new node and sibling
    newParent = copy of parent plus pointer to newNode
         and key between newNode and sibling
    gParent.ptrs[path.pIdx] = newParent
      Mark node, parent, and sibling
      Release all locks and return
  else
    newNode = Combined keys of node and sibling
    if gParent = entry and parent.size = 2
      entry.ptrs[0] = newNode
      Mark node, parent, and sibling
      Release all locks and return
    else
      newParent = copy of parent with pointer to
           newNode instead of node/sibling
      path.gp.ptrs[path.pIdx] = newParent
      Mark node, parent, and sibling
      Release all locks
      fixUnderfull(newNode)
      fixUnderfull(newParent)
```

Figure 9. OCC-ABtree fixUnderfull rebalancing step

that n is underfull, its parent p is not underfull, and none of n, p, and s are tagged. If these conditions are not satisfied, `fixUnderfull` retries its search.

3.3 Correctness

We give linearization points for all operations and briefly outline the progress argument. Detailed proofs of linearizability appear in the full paper [53].

A `find` is linearized at the second read of the version of the terminal leaf (line 41), *if* the leaf is still in the tree at that time. Otherwise, it is linearized at the time just before the leaf was unlinked. In the latter case, it is guaranteed that the `find` was in progress when the leaf was unlinked.

There are four cases for linearizing `insert(key, val)` operations. A `delete(key)` operation is linearized similarly to the first three cases.

- If an `insert` finds key during its invocation of `search`, then it is linearized similarly to a `find` operation.
- If the `insert` does not find key during its invocation of `search`, but then finds key after it locks the leaf (i.e., the key was inserted between the `search` and the lock acquisition), then it can be linearized any time between the lock acquisition and release.
- If the `insert` performs a simple insert, then it is linearized when it increments the leaf's version for the second time (line 85). This linearization point is chosen to support publishing elimination in the next section.
- Finally, if the `insert` performs a splitting insert, then it is linearized when it changes the child pointer of the parent to point to the new `TaggedInternal` (line 94).

Intuitively, deadlock freedom is guaranteed by locking order: nodes are locked from bottom to top, with ties broken by left-to-right ordering. Note that the relative ordering never changes between two siblings, nor between parent and child.

We have also created a version of the OCC-ABtree with a height bounded by $O(\log(n) + c)$ height, where c is the number of threads currently executing an operation on the tree. However, this version is slightly slower and has more complicated rebalancing logic.

4 Elimination

We now describe a technique for eliminating *dictionary operations* by carefully choosing the linearization order for concurrent insertions and deletions of the *same key*. In the following, we say an insertion or deletion of key is *in progress* after it is invoked and before it returns.

Suppose O is a simple insert(key, val). If a deletion of key is in progress when O is linearized, then the delete can be linearized immediately before O and return $\perp$ (without modifying the data structure). Similarly, if an insertion of key is in progress when O is linearized, then the insert can be linearized immediately after O and return val. Since neither of these operations change the data structure (when linearized in this way), an arbitrary number of insertions and deletions of key can be eliminated, provided they are in progress when O is linearized.

The case where O is a successful delete(key) is similar. A deletion of key that is in progress when O is linearized can be linearized after O (and return $\perp$), and an insertion of key that is in progress when O is linearized can be linearized before O (and return the value removed by O).

4.1 Publishing elimination algorithm

The challenge is now to *detect* insertions and deletions of key that are in progress when O is linearized. We describe a modified version of the OCC-ABtree called the Elim-ABtree, in which each leaf additionally stores a summary, called an ElimRecord, of the last operation O that *modified* it. An ElimRecord contains the following three fields. key (resp. val) stores the key (resp. value) that O inserted or deleted. ver stores a version number that helps an insert or delete determine whether it was in progress when O is linearized.

Concurrent operations use the ElimRecord to eliminate themselves as follows. Recall how a simple insert or successful delete O modifies a leaf l. It first increments the version number of l to an odd value v, then modifies l, then increments l's version number to the even value $v+1$. It linearizes at this second increment. O publishes an ElimRecord rec in l just after the first increment. rec.ver is set to v.[4]

Observe that an insert or delete O' is in progress when O is linearized if the following conditions hold:

C1. O' reads $l.ver$ and sees it is $\leq rec.ver$, and
C2. O' returns after $l.ver > rec.ver$

Let us see how an insert(key, val) decides whether it can eliminate itself. The insert first searches towards a leaf. Once it arrives at a leaf l, it optimistically scans l *once*, looking for key. (In contrast, in the OCC-ABtree, searchLeaf is used to repeatedly scan l until it obtains a consistent snapshot of l's contents.)

[4]For simplicity, we only eliminate simple inserts and successful deletes. Eliminating splitting inserts would be more complicated and they are not as frequent.

```
185 // K is key type, V is value type
186 type ElimRecord {key: K, val: V, ver: int}
187 type Leaf
188   ...
189   rec: ElimRecord
190
191 V insert(key, val)
192   ... // Find leaf and search it once
193   acq, retval = lockOrElim(leaf, key)
194   if not acq
195     return retval
196
197   // Did not eliminate, insert as usual
198   leaf.ver++
199   leaf.rec = <key, val, leaf.ver>
200   ... // Insert key
201   leaf.ver++
202   Unlock leaf and return ⊥
203   ...
204
205 // Returns <true, _> if acquired
206 // Returns <false, val> if eliminated
207 <bool, V> lockOrElim(leaf, key)
208   startVer = leaf.ver
209   while true
210     // Try to eliminate self
211     do
212       ver1 = leaf.ver
213       rec = leaf.rec
214       ver2 = leaf.ver
215     while ver1 is odd or ver1 ≠ ver2
216
217     if startVer ≤ rec.ver and rec.key = key
218       return <false, rec.val>
219
220     // Cannot eliminate, try to lock
221     if leaf.lock.tryLock()
222       return <true, _>
```

Figure 10. Elimination pseudocode

If this single scan is not consistent, then the insert is concurrent with another update, so we try to eliminate it by invoking lockOrElim (Figure 10). lockOrElim either eliminates the insert and returns <false, rec.val> (where rec.val is the value that the insert should return), or acquires the leaf's lock and returns <true, ⊥>. In the latter case, the insert then inserts <key, val> into l and releases the lock (as in the OCC-ABtree).

On the other hand, suppose the scan *was* consistent. If it found key, then no modification is necessary, and the insert returns. Otherwise, it will use lockOrElim to try to lock l so it can insert key. (If the insert experiences contention while acquiring the lock, it might even be eliminated.)

How lockOrElim performs elimination. In lockOrElim, the insert attempts to read a snapshot of the leaf's ElimRecord. To do this, it reads the leaf's version (line 211), then reads the ElimRecord rec, then re-reads the leaf's version (line 215). If the reads of the leaf's version return identical results, and the version is even (indicating the leaf is not being modified), then a snapshot was obtained. Otherwise, lockOrElim tries to obtain a snapshot again.

Once a snapshot is obtained, condition C2 is guaranteed to be satisfied. To see why, note that the leaf's version is even when it is last read at line 214 by the exit condition

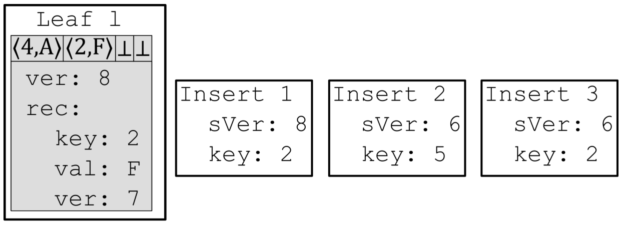

Figure 11. Consider the state of leaf l as shown. $l.rec$ stores the Elim-ABtree of a completed simple insert `insert(2,F)`. Consider three (independent) inserts that are attempting to insert in l and are all at line 217. Insert 1 cannot eliminate itself with `rec` since the version of the leaf it read is greater than `rec.ver`. Insert 2 cannot eliminate itself since its key does not match `rec.key`. Insert 3 can eliminate itself.

of the loop. But, `rec.ver` is always an odd value, thus the version read at line 214 is at least `rec.ver+1`.

At line 217, `lockOrElim` tries to determine whether condition 1 is satisfied. If it is, and `key` matches `rec.key`, then this insert can be eliminated. So, `lockOrElim` returns `<false, rec.val>` and `insert` returns `rec.val` at line 195. Otherwise, `lockOrElim` tries to lock the leaf at line 221. If it acquires the lock, it returns `<true, ⊥>`. If `lockOrElim` fails to acquire the lock, it attempts to eliminate the insert again.

The elimination of deletes is similar, except that eliminated deletes always return ⊥ (not `rec.val`). Figure 11 shows an example of publishing elimination.

Searches could be eliminated. Finally, we note that the `ElimRecord` could also be used to linearize `find`s in high-contention workloads. In some extreme scenarios, this could possibly be useful in preventing `find(key)` from being starved by an endless stream of updates to `key`. We did not observe this in our experiments, since our node size is small enough that searches can typically traverse a leaf in the interval between when one update completes and the next one begins.

5 Persistent trees

In this section we describe the changes to make a persistent version of the OCC-ABtree, the p-OCC-ABtree. The p-OCC-ABtree persists only its keys, values, and pointers. Every update in the p-OCC-ABtree appears to occur atomically in persistent memory. Thus, the recovery procedure for the p-OCC-ABtree is extremely simple: it traverses the tree in persistent memory starting from the root (which is in a known location), and fixes all non-persisted fields (i.e. setting `size` to the actual number of pointers/values in the node, and resetting version, lock state, and the marked bit to their initial values).

The updates in the p-OCC-ABtree require the following cache line flushes. (Below, a flush refers to a `clwb` instruction followed by an `sfence`). For a simple `insert(key, val)`, two flushes must be used: `val` must be flushed after it is written, and `key` must be flushed after it is written. The insert occurs atomically when the `key` reaches persistent memory. Note that if a crash occurs *after* `val` is flushed but before `key` is, `key` is still ⊥ so the key-value pair is *not* logically in the tree. For a successful delete, `key` must be flushed after it is set to ⊥. The delete occurs atomically when the key field is equal to ⊥ in persistent memory.

Recall that splitting inserts and rebalancing steps occur atomically in *volatile* memory by creating a set of new nodes and linking them into the tree by changing a single pointer. We guarantee that these updates appear atomically in *persistent* memory by flushing the new nodes before changing the pointer, then flushing the pointer. The update occurs atomically when the new pointer is flushed.

Operations in the p-OCC-ABtree must only follow *persisted* pointers. To see why, consider the following scenario: a splitting `insert` inserts `key` and `val`, then a `find(key)` operation returns `val`, then a crash occurs before the pointer to the new nodes is persisted. In this case, the recovered tree will *not* contain the inserted key-value pair, so the `find` cannot be linearized. To ensure that all operations only access persisted data, we use the link-and-persist method from [21] (a similar technique is given in [55]). In this technique, whenever an update writes a new pointer p into the tree, p is written with a mark on it to indicate that it has not been persisted. It is then flushed, and the mark is removed. Whenever a thread encounters a marked pointer, it waits until the mark is cleared (hence the pointer is flushed) before following the pointer.

There are two differences in the linearization points of the p-OCC-ABtree. First, splitting inserts must linearize when the new pointer is flushed. Operations cannot access the new key-value pair before this point because the pointer to the tagged node is still marked. The second change is more subtle. In the OCC-ABtree and Elim-ABtree, a simple insert or successful delete O is linearized at the second increment of the version number. In the persistent setting, a crash could occur *before* this increment but *after* `key` has been flushed, so the update will be recovered. To deal with this, any simple insert or successful delete that flushes `key` but has not yet incremented version for the second time when a crash occurs is linearized at the time of the crash. These changes result in a durable linearizable implementation (as proved in the full paper [53]).

The Elim-ABtree can also be made persistent by applying the same changes. We call the resulting tree the p-Elim-ABtree. The change to the linearization point of O does not affect the correctness argument for publishing elimination, since O can only cause the elimination of another operation *after* O has incremented the version for the second time.

6 Experiments

In this section, we compare our trees with other leading dictionary implementations using both a synthetic microbenchmark and the Yahoo! Cloud Serving Benchmark [19], as

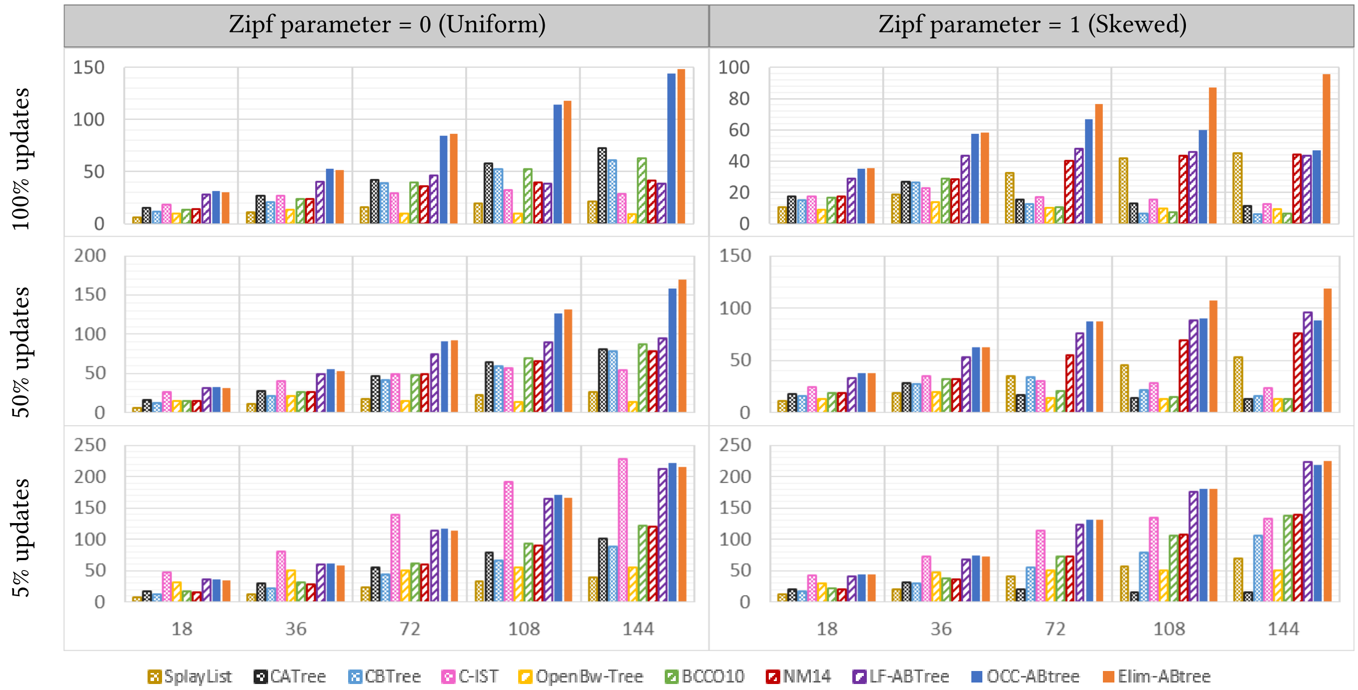

Figure 12. SetBench microbenchmark with 1M keys. x-axis: number of threads. y-axis: operations per μs.

implemented in SetBench (a framework for benchmarking concurrent dictionaries) [15].

See Section 2 for descriptions of the data structures included in our graphs. In the following figures, solid bars represent our trees, striped bars represent data structures that are distribution-naïve (LF-ABtree, BCCO10, NM14, OpenBw-Tree), and checkered bars represent data structures that adapt their structure to the access distribution (CATree, CBTree, SplayList), or try to exploit it to obtain faster searches (C-IST). We also tested DGT15 and the ART with optimistic lock coupling. DGT15 is excluded because it performed strictly worse than NM14, and the ART is excluded because it relies on the keys being binary comparable, but a comparison with both data structures appears in the full paper [53].

System. Our volatile memory experiments (Figure 12, Figure 13) run on a 4-socket Intel Xeon Gold 5220 with 18 cores per socket and 2 hyperthreads (HTs) per core (for a total of 144 hardware threads), and 192GiB of RAM. Our persistent memory experiments (Section 6.3) run on a 2-socket Intel Xeon Gold 5220R CLX with 24 cores per socket and 2 HTs per core (for a total of 96 hardware threads), 192GiB of RAM, and 1536GiB of Intel 3DXPoint NVRAM. In all of our experiments, we pin threads such that the first socket is saturated before the second socket is used, and so on. Additionally, the pinning ensures that all cores on a socket are used before hyperthreading was engaged. The machine runs Ubuntu 20.04.2 LTS. All code is written in C++ and compiled with G++ 7.5.0-3 with compilation options `-std=c++17 -O3`. We use the scalable allocator jemalloc 5.0.1-25. We use `numactl -i all` to interleave pages evenly across all NUMA nodes.

Memory reclamation. All data structures use DEBRA, a variant of epoch-based memory reclamation [16], except the SplayList, FPTree and RNTree (which do not reclaim memory) and the OpenBw-Tree (which uses a different epoch-based reclamation scheme which we were unable to change due to its complexity).

Methodology. Each experiment *run* starts with a prefilling phase, in which a random subset of 8-byte keys and values are inserted into the data structure until the data structure size reaches its expected steady-state size (half of the key range, since the proportions of inserts and deletes are equal in our experiments). After the prefilling phase, n threads are created and started together, and the *measured* phase of the experiment begins. In this phase, each thread repeatedly selects an operation (`insert`, `delete`, `find`) based on the desired update frequency, and selects a key according to a uniform or Zipfian distribution. This continues for 10 seconds, and the total *throughput* (operations completed per second) is recorded. Each experiment is run three times, and our graphs plot the averages of these runs.

Validation. To sanity-check the correctness of the evaluated data structures, each thread keeps track of the sum of keys that it successfully inserts and deletes. At the end of each run, all threads' sums are added to a grand total, and the grand total must match the sum of keys in the data structure.

6.1 SetBench microbenchmark

Read-mostly (5% updates). Performance on read-mostly workloads has been shown to be correlated with short paths to keys, since shorter paths resulted in fewer cache misses (which dominate runtime in read-mostly workloads) [15].

Thus, we expected the (a,b)-trees, OpenBw-Tree, CBTree, and C-IST (all of which use fat nodes containing many pointers) to be the fastest. However, this is only true for the (a,b)-trees. The C-IST, which is heavily optimized for search-only workloads, performs well in the uniform case, but performs much worse in the Zipfian case. The OpenBw-Tree performs poorly in both workloads. However, a short experiment suggests that both the C-IST and OpenBw-Tree perform comparably to the (a,b)-trees with *no* updates. The extent to which just 5% updates affects their read performance is surprising. The BSTs (BCCO10, NM14) have similar performance relative to one another (roughly half that of the (a,b)-trees).

The CBTree and SplayList fell short of our expectations on the Zipfian workload. We expected that splaying would greatly accelerate searches (especially since the splayed key is never removed in a read-mostly workload), but they barely exceed their performance on the uniform workload. The CATree's performance is reasonable on the uniform workload, but is much worse than the other data structures on the Zipfian workload. All of the CATree's operations (even searches) require *locking* a leaf.

Update-heavy (50%, 100% updates). Overall, throughput decreases as the proportion of updates increases (as expected).

On *uniform* update-heavy workloads, the LF-ABtree and the C-IST scale much worse than our trees. The LF-ABtree creates a new copy of a (fat) node every time a key is inserted. The C-IST must completely rebuild the tree after $n/4$ updates, where n is the size of the tree. As a result, both incur high overhead for updates. The other competing trees have better scaling but relatively poor absolute throughput. Our trees are roughly 2x faster than the leading competitor (the CATree) in the uniform 100% workload.

On *skewed* update-heavy workloads, the benefit of publishing elimination becomes clear. The Elim-ABtree is significantly faster than the OCC-ABtree on these workloads, with the gap increasing as the proportion of updates does. At 100% updates, the Elim-ABtree is up to 2.5x as fast as its fastest competitor. The C-IST still scales poorly on these workloads, but the LF-ABtree performs extremely well, outperforming even the OCC-ABtree at 50% updates. At relatively low update rates, the benefit of lock-freedom (i.e., faster threads *helping* slower threads) exceeds the overhead of allocating new nodes for each key inserted. At the highest update rates, the overhead of managing memory dominates the performance of the LF-ABtree.

NM14 scales much better than BCCO10 in these workloads, slightly exceeding the performance of the OCC-ABtree. This is because searches in BCCO10 have to restart many times because of frequent updates along the path to the frequently-accessed keys. A notable outlier in the skewed update-heavy workloads is the SplayList, which had relatively poor read-mostly performance but matches the performance of NM14 and the LF-ABtree on the skewed update-only workload. This may be partially because the SplayList never frees memory (simply marking keys as deleted instead), so reinserting a key that was once in the SplayList requires no memory allocation (which normally adds considerable overhead to the other data structures). This approach is quite efficient in our microbenchmark, but might be less so if the set of keys that are *ever* inserted is much larger than the set of keys that are *typically* in the dictionary.

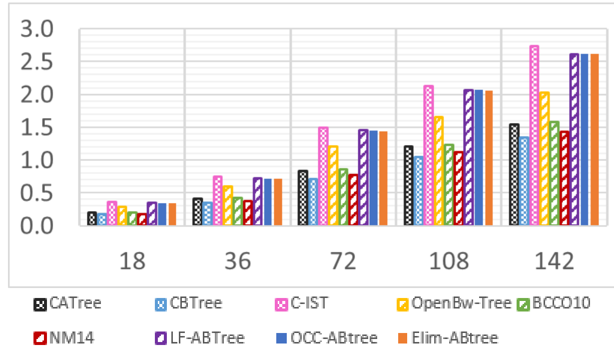

Figure 13. YCSB throughput on Workload A. x-axis: number of threads. y-axis: transactions per μs.

6.2 YCSB

The Yahoo! Cloud Serving Benchmark (YCSB) is a standard tool for benchmarking concurrent database indices [19]. We run the benchmark using the above data structures as the database index. We run Workload A (50% reads, 50% writes, Zipf factor 0.5) from the YCSB standard workloads, with a uniform access distribution and an initial data structure size of 100M (Figure 13). Figure 13 does not contain the SplayList since it does not reclaim memory and consequently caused the system to run out of memory. Note that the writes in the YCSB workload are to the database itself, not the index. That is, a YCSB write simply *reads* the row pointer from the index, then locks the row, updates it, and unlocks it (without modifying the index). As a result, the results are closest to our microbenchmark uniform read-mostly workload.

6.3 Persistence experiments

Of the concurrent persistent trees in Section 2, only the FPTree and RNTree have publicly available implementations that passed our validation scheme (both implementions were from [41]). However, these implementations do not reclaim memory.

Figure 14 shows the results of our microbenchmark on our persistent memory machine. Even with the overhead of reclaiming memory, the p-OCC-ABtree and p-Elim-ABtree outperform both the FPTree and the RNTree on all thread counts. (Results on smaller/larger key ranges and different update percentages were similar). In the uniform case, the FPTree performs similarly to our trees at low thread counts but exhibits extreme negative scaling when running on 2 sockets (96 threads). However, this might be an artifact of this

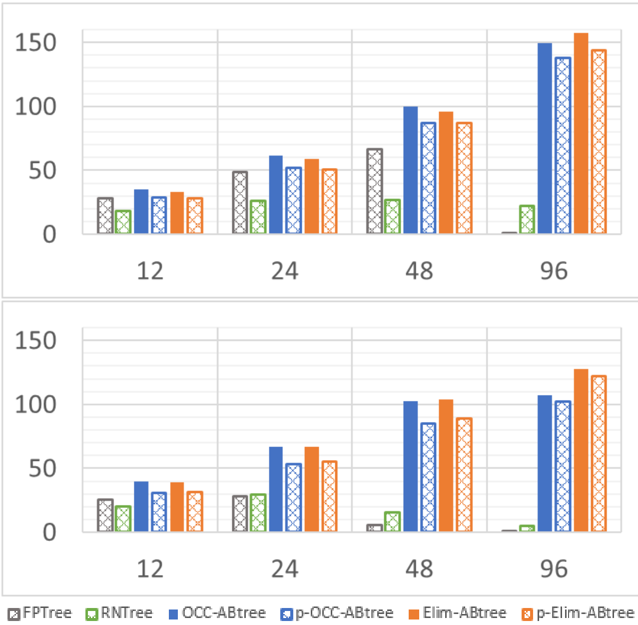

Figure 14. Comparing with other persistent trees: SetBench microbenchmark with 1M keys, 50% updates (25% insert and 25% delete). Left: Uniform access distribution. Right: Zipfian access distribution (with Zipf factor 1). x-axis: number of threads. y-axis: operations per μs.

particular implementation, since the original paper shows better scaling on 2 sockets. The RNTree performs worse than the FPTree on uniform workloads, but slightly better on the Zipfian workload. Both the FPTree and RNTree also exhibit negative scaling in the Zipfian case, even when running on only one socket.

We attempted to compare with an unofficial implementation of the BzTree [39], but encountered failures during validation. The implementors mentioned that the errors might be fixable, but were unable to produce a fix in time for this publication. Table 1 shows the persistence overhead of our trees. Comparing with the overheads listed in the BzTree paper, the overhead of our trees is slightly less than the BzTree's average persistence cost of 5% on a uniform 10%-update workload and 12% on a uniform 50%-update workload.

7 Future work and conclusion

It would be interesting to explore the interaction between publishing elimination and different data structure semantics. Publishing elimination remains correct for some alternative definitions of insert. If insert replaces existing keys but returns no value (instead of simply returning the existing key), publishing elimination does not require any modifications: the thread that successfully modifies the data structure is linearized last.

On the other hand, if insert returns the value it replaces, then publishing elimination would require changes to allow

	Uniform			Zipfian		
Update rate:	**100%**	**50%**	**10%**	**100%**	**50%**	**10%**
p-OCC-ABtree	-16%	-8%	-6%	-6%	-9%	-7%
p-Elim-ABtree	-14%	-9%	-1%	-5%	-5%	-5%

Table 1. Change in throughput upon enabling persistence. 96 threads, 1 million keys.

each insert in a sequence of linearized inserts to communicate its value to the next insert.

Using MCS locks (instead of test-and-test-and-set spinlocks) significantly increased the scalability of the OCC-ABtree. Using NUMA-aware locks like HCLH [42], lock cohorting [24], or NUMA-aware reader-writer locks [17] might also be a simple way of improving performance further.

We have introduced the OCC-ABtree, which provides good performance in both read-mostly and update-heavy workloads, and the Elim-ABtree which uses publishing elimination to further improve performance in high-contention workloads. Finally, we have presented persistent versions of our trees that require only minor modifications and are still highly performant.

Acknowledgments

This work was supported by: the Natural Sciences and Engineering Research Council of Canada (NSERC) Collaborative Research and Development grant: CRDPJ 539431-19, the Canada Foundation for Innovation John R. Evans Leaders Fund with equal support from the Ontario Research Fund CFI Leaders Opportunity Fund: 38512, Waterloo Huawei Joint Innovation Lab project "Scalable Infrastructure for Next Generation Data Management Systems", NSERC Discovery Launch Supplement: DGECR-2019-00048, NSERC Discovery Program under the grants: RGPIN-2019-04227 and RGPIN-04512-2018, and the University of Waterloo. We would also like to thank the reviewers for their insightful comments.

References

[1] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. 1993. Atomic Snapshots of Shared Memory. *J. ACM* 40, 4 (Sept. 1993), 873–890. https://doi.org/10.1145/153724.153741

[2] Yehuda Afek, Haim Kaplan, Boris Korenfeld, Adam Morrison, and Robert E. Tarjan. 2012. CBTree: A Practical Concurrent Self-Adjusting Search Tree. In *Proceedings of the 26th International Conference on Distributed Computing* (Salvador, Brazil) *(DISC'12)*. Springer-Verlag, Berlin, Heidelberg, 1–15. https://doi.org/10.1007/978-3-642-33651-5_1

[3] Marcos K Aguilera and Svend Frølund. 2003. *Strict linearizability and the power of aborting*. Technical Report. HP Labs.

[4] Vitaly Aksenov, Dan Alistarh, Alexandra Drozdova, and Amirkeivan Mohtashami. 2020. The Splay-List: A Distribution-Adaptive Concurrent Skip-List. In *34th International Symposium on Distributed Computing (DISC 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 179)*, Hagit Attiya (Ed.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 3:1–3:18. https://doi.org/10.4230/LIPIcs.DISC.2020.3

[5] Maya Arbel-Raviv and Trevor Brown. 2018. Harnessing Epoch-Based Reclamation for Efficient Range Queries. In *Proceedings of the 23rd ACM*

SIGPLAN Symposium on Principles and Practice of Parallel Programming (Vienna, Austria) *(PPoPP '18)*. Association for Computing Machinery, New York, NY, USA, 14–27. https://doi.org/10.1145/3178487.3178489

[6] Maya Arbel-Raviv, Trevor Brown, and Adam Morrison. 2018. Getting to the Root of Concurrent Binary Search Tree Performance. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 295–306. https://www.usenix.org/conference/atc18/presentation/arbel-raviv

[7] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. 2018. Bztree: A High-Performance Latch-Free Range Index for Non-Volatile Memory. *Proc. VLDB Endow.* 11, 5 (Jan. 2018), 553–565. https://doi.org/10.1145/3164135.3164147

[8] Anastasia Braginsky, Nachshon Cohen, and Erez Petrank. 2016. CBPQ: High Performance Lock-Free Priority Queue. In *Euro-Par 2016: Parallel Processing*, Pierre-François Dutot and Denis Trystram (Eds.). Springer International Publishing, Cham, 460–474.

[9] Anastasia Braginsky and Erez Petrank. 2012. A Lock-Free B+tree. In *Proceedings of the Twenty-Fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures* (Pittsburgh, Pennsylvania, USA) *(SPAA '12)*. Association for Computing Machinery, New York, NY, USA, 58–67. https://doi.org/10.1145/2312005.2312016

[10] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. 2010. A Practical Concurrent Binary Search Tree. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Bangalore, India) *(PPoPP '10)*. Association for Computing Machinery, New York, NY, USA, 257–268. https://doi.org/10.1145/1693453.1693488

[11] Trevor Brown. 2017. *Techniques for Constructing Efficient Lock-free Data Structures*. Ph.D. Dissertation. University of Toronto. arXiv:1807/80693

[12] Trevor Brown. 2017. A Template for Implementing Fast Lock-Free Trees Using HTM. In *Proceedings of the ACM Symposium on Principles of Distributed Computing* (Washington, DC, USA) *(PODC '17)*. Association for Computing Machinery, New York, NY, USA, 293–302. https://doi.org/10.1145/3087801.3087834

[13] Trevor Brown, Faith Ellen, and Eric Ruppert. 2013. Pragmatic Primitives for Non-Blocking Data Structures. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing* (Montréal, Québec, Canada) *(PODC '13)*. Association for Computing Machinery, New York, NY, USA, 13–22. https://doi.org/10.1145/2484239.2484273

[14] Trevor Brown, Faith Ellen, and Eric Ruppert. 2014. A General Technique for Non-Blocking Trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Orlando, Florida, USA) *(PPoPP '14)*. Association for Computing Machinery, New York, NY, USA, 329–342. https://doi.org/10.1145/2555243.2555267

[15] Trevor Brown, Aleksandar Prokopec, and Dan Alistarh. 2020. Non-Blocking Interpolation Search Trees with Doubly-Logarithmic Running Time. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, California) *(PPoPP '20)*. Association for Computing Machinery, New York, NY, USA, 276–291. https://doi.org/10.1145/3332466.3374542

[16] Trevor Alexander Brown. 2015. Reclaiming Memory for Lock-Free Data Structures: There Has to Be a Better Way. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing* (Donostia-San Sebastián, Spain) *(PODC '15)*. Association for Computing Machinery, New York, NY, USA, 261–270. https://doi.org/10.1145/2767386.2767436

[17] Irina Calciu, Dave Dice, Yossi Lev, Victor Luchangco, Virendra J. Marathe, and Nir Shavit. 2013. NUMA-Aware Reader-Writer Locks. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Shenzhen, China) *(PPoPP '13)*. Association for Computing Machinery, New York, NY, USA, 157–166. https://doi.org/10.1145/2442516.2442532

[18] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. 2017. Black-Box Concurrent Data Structures for NUMA Architectures. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 207–221. https://doi.org/10.1145/3037697.3037721

[19] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[20] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. 2006. Hybrid Transactional Memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) *(ASPLOS XII)*. Association for Computing Machinery, New York, NY, USA, 336–346. https://doi.org/10.1145/1168857.1168900

[21] Tudor David, Aleksandar Dragojević, Rachid Guerraoui, and Igor Zablotchi. 2018. Log-Free Concurrent Data Structures. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '18)*. USENIX Association, USA, 373–385.

[22] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2015. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (Istanbul, Turkey) *(ASPLOS '15)*. Association for Computing Machinery, New York, NY, USA, 631–644. https://doi.org/10.1145/2694344.2694359

[23] Dave Dice and Alex Kogan. 2019. Compact NUMA-Aware Locks. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 12, 15 pages. https://doi.org/10.1145/3302424.3303984

[24] David Dice, Virendra J. Marathe, and Nir Shavit. 2012. Lock Cohorting: A General Technique for Designing NUMA Locks. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New Orleans, Louisiana, USA) *(PPoPP '12)*. Association for Computing Machinery, New York, NY, USA, 247–256. https://doi.org/10.1145/2145816.2145848

[25] Dana Drachsler-Cohen and Erez Petrank. 2014. Lcd: Local combining on demand. In *International Conference On Principles Of Distributed Systems*. Springer, Cortina d'Ampezzo, Italy, 355–371.

[26] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. 2010. Non-Blocking Binary Search Trees. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing* (Zurich, Switzerland) *(PODC '10)*. Association for Computing Machinery, New York, NY, USA, 131–140. https://doi.org/10.1145/1835698.1835736

[27] Panagiota Fatourou and Nikolaos D. Kallimanis. 2009. The RedBlue Adaptive Universal Constructions. In *Proceedings of the 23rd International Conference on Distributed Computing* (Elche, Spain) *(DISC'09)*. Springer-Verlag, Berlin, Heidelberg, 127–141.

[28] Panagiota Fatourou and Nikolaos D. Kallimanis. 2011. A Highly-Efficient Wait-Free Universal Construction. In *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures* (San Jose, California, USA) *(SPAA '11)*. Association for Computing Machinery, New York, NY, USA, 325–334. https://doi.org/10.1145/1989493.1989549

[29] 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 (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 377–392. https://doi.org/10.1145/3385412.3386031

[30] Michal Friedman, Erez Petrank, and Pedro Ramalhete. 2021. Mirror: Making Lock-Free Data Structures Persistent. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 1218–1232. https://doi.org/10.1145/3453483.3454105

[31] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. 2010. Flat Combining and the Synchronization-Parallelism Tradeoff. In *Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures* (Thira, Santorini, Greece) *(SPAA '10)*. Association for Computing Machinery, New York, NY, USA, 355–364. https://doi.org/10.1145/1810479.1810540

[32] Danny Hendler, Nir Shavit, and Lena Yerushalmi. 2004. A Scalable Lock-Free Stack Algorithm. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures* (Barcelona, Spain) *(SPAA '04)*. Association for Computing Machinery, New York, NY, USA, 206–215. https://doi.org/10.1145/1007912.1007944

[33] Shane V. Howley and Jeremy Jones. 2012. A Non-Blocking Internal Binary Search Tree. In *Proceedings of the Twenty-Fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures* (Pittsburgh, Pennsylvania, USA) *(SPAA '12)*. Association for Computing Machinery, New York, NY, USA, 161–171. https://doi.org/10.1145/2312005.2312036

[34] Joseph Izraelevitz, Hammurabi Mendes, and Michael Scott. 2016. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In *International Symposium on Distributed Computing*, Vol. 9888. Association for Computing Machinery, Paris, France, 313–327. https://doi.org/10.1007/978-3-662-53426-7_23

[35] Kim S. Larsen and Rolf Fagerberg. 1995. B-Trees with Relaxed Balance. In *Proceedings of the 9th International Symposium on Parallel Processing (IPPS '95)*. IEEE Computer Society, USA, 196–202.

[36] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. Recipe: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 462–477. https://doi.org/10.1145/3341301.3359635

[37] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, Brisbane, QLD, Australia, 38–49.

[38] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of Practical Synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware* (San Francisco, California) *(DaMoN '16)*. Association for Computing Machinery, New York, NY, USA, Article 3, 8 pages. https://doi.org/10.1145/2933349.2933352

[39] Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. 2019. Evaluating Persistent Memory Range Indexes. *Proc. VLDB Endow.* 13, 4 (Dec. 2019), 574–587. https://doi.org/10.14778/3372716.3372728

[40] J. J. Levandoski, D. B. Lomet, and S. Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, Brisbane, QLD, Australia, 302–313. https://doi.org/10.1109/ICDE.2013.6544834

[41] 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* (Kyoto, Japan) *(ICPP 2019)*. Association for Computing Machinery, New York, NY, USA, Article 101, 10 pages. https://doi.org/10.1145/3337821.3337827

[42] Victor Luchangco, Dan Nussbaum, and Nir Shavit. 2006. A Hierarchical CLH Queue Lock. In *Proceedings of the 12th International Conference on Parallel Processing* (Dresden, Germany) *(Euro-Par'06)*. Springer-Verlag, Berlin, Heidelberg, 801–810. https://doi.org/10.1007/11823285_84

[43] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache Craftiness for Fast Multicore Key-Value Storage. In *Proceedings of the 7th ACM European Conference on Computer Systems* (Bern, Switzerland) *(EuroSys '12)*. Association for Computing Machinery, New York, NY, USA, 183–196. https://doi.org/10.1145/2168836.2168855

[44] John M. Mellor-Crummey and Michael L. Scott. 1991. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Trans. Comput. Syst.* 9, 1 (Feb. 1991), 21–65. https://doi.org/10.1145/103727.103729

[45] Aravind Natarajan and Neeraj Mittal. 2014. Fast Concurrent Lock-Free Binary Search Trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Orlando, Florida, USA) *(PPoPP '14)*. Association for Computing Machinery, New York, NY, USA, 317–328. https://doi.org/10.1145/2555243.2555256

[46] Aravind Natarajan, Lee H. Savoie, and Neeraj Mittal. 2013. Concurrent Wait-Free Red Black Trees. In *15th International Symposium on Stabilization, Safety, and Security of Distributed Systems - Volume 8255* (Osaka, Japan) *(SSS 2013)*. Springer-Verlag, Berlin, Heidelberg, 45–60.

[47] 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* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 371–386. https://doi.org/10.1145/2882903.2915251

[48] William Pugh. 1990. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Commun. ACM* 33, 6 (June 1990), 668–676. https://doi.org/10.1145/78973.78977

[49] Arunmoezhi Ramachandran and Neeraj Mittal. 2015. A Fast Lock-Free Internal Binary Search Tree. In *Proceedings of the 2015 International Conference on Distributed Computing and Networking* (Goa, India) *(ICDCN '15)*. Association for Computing Machinery, New York, NY, USA, Article 37, 10 pages. https://doi.org/10.1145/2684464.2684472

[50] K. Sagonas and K. Winblad. 2015. Contention Adapting Search Trees. In *2015 14th International Symposium on Parallel and Distributed Computing*. Association for Computing Machinery, Limassol, Cyprus, 215–224. https://doi.org/10.1109/ISPDC.2015.32

[51] Nir Shavit and Dan Touitou. 1995. Elimination Trees and the Construction of Pools and Stacks: Preliminary Version. In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures* (Santa Barbara, California, USA) *(SPAA '95)*. Association for Computing Machinery, New York, NY, USA, 54–63. https://doi.org/10.1145/215399.215419

[52] Daniel Dominic Sleator and Robert Endre Tarjan. 1985. Self-Adjusting Binary Search Trees. *J. ACM* 32, 3 (jul 1985), 652–686. https://doi.org/10.1145/3828.3835

[53] Anubhav Srivastava and Trevor Brown. 2021. Elimination (a,b)-trees with fast, durable updates. arXiv:2112.15259 [cs.DC]

[54] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. 2011. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies* (San Jose, California) *(FAST'11)*. USENIX Association, USA, 5.

[55] 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, Paris,France, 461–472.

[56] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. 2018. Building a Bw-Tree Takes More Than Just Buzz Words. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA)

(SIGMOD '18). Association for Computing Machinery, New York, NY, USA, 473–488. https://doi.org/10.1145/3183713.3196895

[57] Haosen Wen, Wentao Cai, Mingzhe Du, Louis Jenkins, Benjamin Valpey, and Michael L. Scott. 2021. *A Fast, General System for Buffered Persistent Data Structures.* Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/3472456.3472458

[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 *Proceedings of the 13th USENIX Conference on File and Storage Technologies* (Santa Clara, CA) *(FAST'15)*. USENIX Association, USA, 167–181.

8 Artifact Description

The artifact containing the source code for all algorithms and experiments run in this paper is available at https://doi.org/10.5281/zenodo.5810865.

Note: Sudo permission may be required to execute the following instructions.

1. Install the latest version of Docker on your system. The artifact was tested with the Docker version 20.10.2. (Instructions to install Docker can be found at https://docs.docker.com/get-docker/.)
2. Download the artifact from Zenodo at URL: https://doi.org/10.5281/zenodo.5810865.
3. Load the downloaded docker image:
   `$ sudo docker load -i setbench.tar.gz`
4. Verify that image was loaded:
   `$ sudo docker images`
5. Start a docker container from the loaded image:
   `$ sudo docker run -p 2222:22 -d -privileged -name setbench setbench`
6. Verify that the container is running (you should see a setbench container):
   `$ sudo docker container ls`
7. SSH into the running container with password `root`:
   `$ ssh root@localhost -p 2222`
8. Follow the instructions in `setbench/README.md` to replicate results. Note that you might have to change thread counts in the `run.sh` and `run_persistence_cost.sh` scripts to match the constraints of your system.