

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 versions¹ 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.

*Corresponding author

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

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

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

© 2022 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-9204-4/22/02.

<https://doi.org/10.1145/3503221.3508441>

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 k th 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

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. Nataraajan 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** [55].

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

(**OLC-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. 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 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).

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

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 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 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 [53]. Yang et al. created the NV-Tree, a persistent B-link tree that outperforms the CDDS B-tree by up to 12x [57]. 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 OCC-ABtree). Montage [56] 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 \perp .
- `insert(k, v)`: If a key-value pair with key k is present, return the associated value. Otherwise, insert the key-value pair $\langle k, v \rangle$ and return \perp .
- `delete(k)`: If a key-value pair with key k is present, delete it and return the associated value. Otherwise, return \perp .

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 \perp . 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 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

```

1 // K is key type, V is value type
2 abstract type Node
3   keys      : K[MAX_SIZE]
4   lock      : MCSLock
5   size      : int
6   marked    : bool
7
8 type Leaf inherits Node
9   vals      : V[MAX_SIZE]
10  ver       : int
11
12 type Internal inherits Node
13  ptrs      : Node[MAX_SIZE]
14
15 type TaggedInternal inherits Internal
16
17 // The result of a search
18 type PathInfo
19   gp       : Node // grandparent
20   p        : Node // parent
21   pIdx     : int
22   n        : Node // node
23   nIdx     : int
24
25 type RetCode is SUCCESS or FAILURE or RETRY
26
27 // Sentinel node: points to root
28 entry     : Internal
29 MIN_SIZE = 2, MAX_SIZE = 11

```

Figure 1. OCC-ABtree data structures

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 leaf³.

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, \perp >. Note that search and

³A 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.

```

30 <RetCode, V> searchLeaf(leaf, key)
31 RETRY:
32   ver1 = leaf.ver
33   if ver1 is odd
34     goto RETRY
35
36   val =  $\perp$ 
37   for keyIndex = 0 up to MAX_SIZE - 1
38     if leaf.keys[keyIndex] = key
39       val = leaf.vals[keyIndex]
40       break
41   ver2 = leaf.ver
42   if ver1  $\neq$  ver2 goto RETRY
43   if val =  $\perp$  return <FAILURE,  $\perp$ >
44   else return <SUCCESS, val>
45
46 PathInfo search(key, targetNode)
47   gp = NULL, p = NULL, pIdx = 0, n = entry, nIdx = 0
48   while n is not Leaf
49     if n = targetNode break
50     gp = p, p = n, pIdx = nIdx, nIdx = 0
51     while nIdx < node.size-1 and key  $\geq$  node.keys[nIdx]
52       nIdx++
53     n = n.ptrs[nIdx]
54   return PathInfo(gp, p, pIdx, n, nIdx)
55
56 V find(key)
57   path = search(key, NULL)
58   rc, val = searchLeaf(path.n, key)
59   return val

```

Figure 2. OCC-ABtree search operations

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 \perp . Otherwise, it locks the leaf and deletes the key by setting it to \perp , and returns the associated value (Figure 3(1)). If key was deleted by another thread between search and acquiring the lock, delete returns \perp . 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**. 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)).

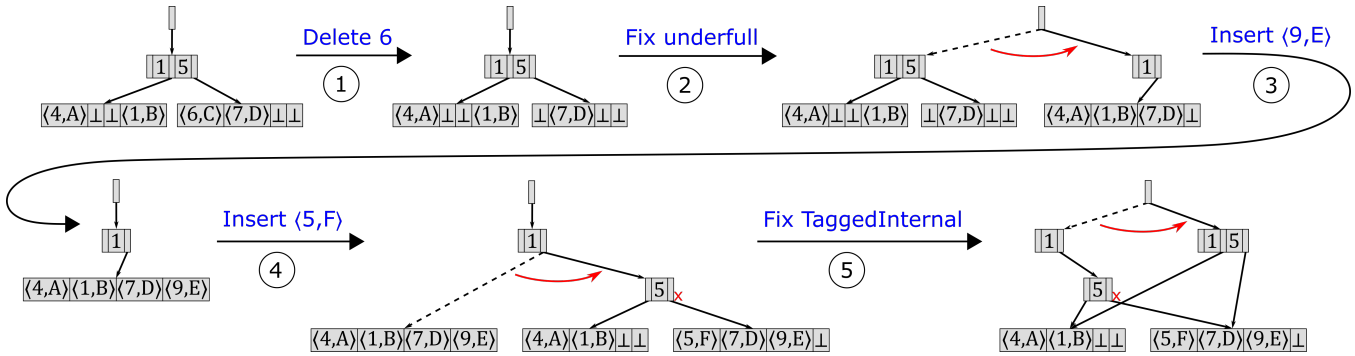


Figure 3. An OCC-ABtree with $a = 2, b = 4$. (1) The key-value pair $\langle 6, C \rangle$ is deleted. This creates an underfull node. (2) The underfull node is merged with its sibling. This leaves the parent underfull, but the parent is the root, which is allowed to remain underfull. (3) $\langle 9, E \rangle$ is inserted into an empty slot (*simple insert*). (4) No empty slot exists for $\langle 5, F \rangle$, 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.

```

60 V insert(key, val)
61 RETRY:
62   path = search(key, NULL)
63   rc, val = searchLeaf(path.n, key)
64   if rc = SUCCESS return val
65
66   leaf, parent = path.n, path.p
67
68   Lock leaf
69   if leaf.marked
70     Unlock leaf and goto RETRY
71
72   // Verify key is not present
73   for i = 0 to DEGREE - 1
74     if leaf.keys[i] = key
75       Unlock leaf and return leaf.vals[i]
76
77   if leaf.size < MAX_NODE_SIZE
78     // Insert without splitting
79     for i = 0 to DEGREE - 1
80       if leaf.keys[i] = ⊥
81         leaf.ver++ // Start modification
82         leaf.keys[i] = key
83         leaf.vals[i] = val
84         leaf.size++
85         leaf.ver++ // End modification
86         Unlock leaf and return ⊥
87   else
88     Lock parent
89     if parent.marked
90       Unlock leaf and parent and goto RETRY
91
92   N = {contents of leaf} ∪ {key/val}
93   newLeaf = TaggedInternal with two children that
94             evenly share N
95   parent.ptrs[path.nIdx] = newLeaf
96   node.marked = true
97   Unlock leaf and parent
98   fixTagged(newLeaf)
99   return ⊥

```

Figure 4. OCC-ABtree insert operation

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, 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

```

99 V delete(key)
100 RETRY:
101   path = search(key, NULL)
102   rc, val = searchLeaf(path.n, key)
103   if rc = FAILURE
104     return ⊥
105
106   leaf = path.n
107   Lock leaf
108   if leaf is marked
109     goto RETRY
110
111   for i = 0 to DEGREE - 1
112     if leaf.keys[i] = key
113       deletedVal = leaf.vals[i]
114       leaf.ver++ // Start modification
115       leaf.keys[i] = ⊥
116       leaf.size--
117       leaf.ver++ // End modification
118
119   if leaf.size < MIN_NODE_SIZE
120     Unlock leaf
121     fixUnderfull(leaf)
122   return ⊥

```

Figure 5. OCC-ABtree delete operation

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 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.

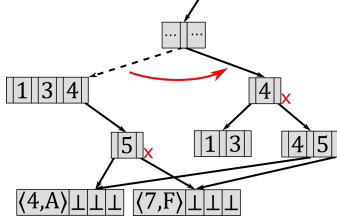


Figure 6. fixTagged split case (merge is in Figure 3)

```

123 fixTagged(node)
124   RETRY:
125     if node.marked return
126     path = search(node.searchKey, node)
127     if path.n ≠ node return
128
129     Lock path.n, path.p, and path.gp
130     if node, parent or gParent is marked or
131     path.p is TaggedInternal
132       Release all locks and goto RETRY
133
134     node.marked = true
135     path.p.marked = true
136     if path.p.size + 1 ≤ MAX_NODE_SIZE
137       newNode = new Internal containing the keys &
138       pointers of node and parent
139       path.gp.ptrs[path.pIdx] = newNode
140       Release all locks
141     else
142       // newNode is a TaggedInternal, unless it will be
143       // the new root (in which case it is Internal)
144       newNode = new subtree of three nodes consisting of
145       a new Internal that points to two new internal
146       nodes which evenly share the keys & pointers
147       of node and parent (except for the pointer to
148       node)
149       path.gp.ptrs[path.pIdx] = newNode
150       Release all locks
151       fixTagged(newNode)

```

Figure 7. OCC-ABtree fixTagged rebalancing step

3.3 Correctness

This section proves that the OCC-ABtree is linearizable. Recall that an algorithm is linearizable if, in every concurrent execution, every operation appears to happen *atomically* at some point between its invocation and its response.

Proving the linearizability of the OCC-ABtree requires a definition linking the *physical* representation of the OCC-ABtree (i.e. the contents of the system’s memory) to the *abstract* dictionary it represents. The operations are then shown to modify the physical state of the tree in a way that is consistent with the abstract semantics described at the beginning of Section 3.

3.3.1 Definitions.

Definition 3.1 (Reachable node). A node is said to be **reachable** if it can be reached by following child pointers from the entry node.

Definition 3.2 (Key in OCC-ABtree). Let l be a reachable leaf. k is **in the OCC-ABtree** if, when l ’s version was last even, k was in l ’s keys array. Furthermore, if k is the i th key in l , the value associated with k is $l.vals[i]$.

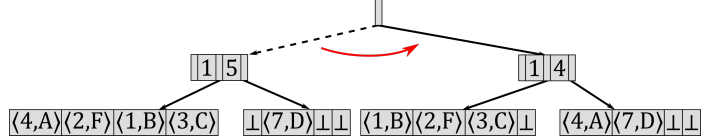


Figure 8. fixUnderfull distribute case (merge is in Figure 3)

```

146 fixUnderfull(node)
147   if node = entry or node = entry.ptrs[0] return
148
149   RETRY:
150     path = search(node.searchKey, node)
151     if path.n ≠ node return
152
153     if path.nIdx = 0
154       sIndex = 1 // Sibling is right child
155     else
156       sIndex = path.nIdx - 1 //
157       sibling = parent.ptrs[sIndex]
158
159     Lock node, sibling, path.p, and path.gp
160     if node.size ≥ MIN_NODE_SIZE return
161     if parent.size < MIN_NODE_SIZE or
162     node, sibling, parent, or gParent is marked or
163     node, sibling or parent is TaggedInternal
164       Release all locks and goto RETRY
165
166     if node.size + sibling.size ≤ 2 * MIN_NODE_SIZE
167       newNode, sibling = Distribute keys of node and
168       sibling evenly amongst new node and sibling
169       newParent = copy of parent plus pointer to newNode
170       and key between newNode and sibling
171       gParent.ptrs[path.pIdx] = newParent
172       Mark node, parent, and sibling
173       Release all locks and return
174     else
175       newNode = Combined keys of node and sibling
176       if gParent = entry and parent.size = 2
177         entry.ptrs[0] = newNode
178         Mark node, parent, and sibling
179         Release all locks and return
180     else
181       newParent = copy of parent with pointer to
182       newNode instead of node/sibling
183       path.gp.ptrs[path.pIdx] = newParent
184       Mark node, parent, and sibling
185       Release all locks
186       fixUnderfull(newNode)
187       fixUnderfull(newParent)

```

Figure 9. OCC-ABtree fixUnderfull rebalancing step

In other words, a key is logically inserted or deleted when a thread increments the version number of the leaf for the second time (making it even).

Definition 3.2 is somewhat counter-intuitive. One might consider the following simpler definition: a key k is in the tree if it is in some leaf’s keys array. Indeed, this alternate definition can also be used to prove that the OCC-ABtree is linearizable. However, Definition 3.2 is necessary for the correctness of publishing elimination (Section 4). Using a consistent definition hopefully makes the correctness argument easier for the reader.

There are two more definitions which are used in the proofs throughout this paper. The **key range** of a node is a half-open subset of the universe of keys (e.g. $[100, 200)$ if the keys are numbers, or $[$ “aardvark”, “apple” $)$ if the keys are strings). Intuitively, the key range of a node is the set of keys that are allowed to appear in the subtree rooted at that node.

Definition 3.3 (Key range). The key range of the entry node is defined to be the universe of keys. Let n be a reachable internal node with key range $[L, R)$. If n has no keys, the key range of its child is also $[L, R)$. Otherwise, suppose n contains keys k_1 to k_m . The key range of n 's leftmost child (pointed to by $n.ptrs[0]$) is $[L, k_1)$, the key range of n 's rightmost child (pointed to by $n.ptrs[m]$) is $[k_m, R)$, and the key range of any middle child pointed to by $n.ptrs[i]$ is $[k_i, k_{i+1})$.

Finally, the OCC-ABtree (along with all other trees in introduced in this paper) is a relaxed (a,b)-tree, as introduced by Larsen and Fagerberg [35]. The relaxed (a,b)-tree is a search tree (as defined below). The most important consequence of the OCC-ABtree being a search tree is that, for any key k in the universe of keys, there is a unique search path for k , and this path passes through every reachable node whose key range contains k . Intuitively, this path is the path an atomic search of k would take. Note that the uniqueness of the path implies that there is a unique reachable leaf in the search tree whose key range contains k .

Definition 3.4 (Search Tree). Suppose n is an internal node in a tree and k is a key in n . A tree is a search tree if

- All keys in the subtrees to the left of k in n are less than k AND
- All keys in the subtrees to the right of k in n are greater than or equal to k

3.3.2 Invariants. Proving an insert is correct requires proving that search finds the correct leaf to insert into. For search to find the correct leaf, the tree must satisfy some structural properties, which are only satisfied if *previous* inserts and deletes were correct. We deal with this cyclical dependency is by assuming a set of invariants about the structure of the tree. These invariants hold for the initial state of the tree, and every modification to the tree preserves all invariants. These invariants can then be used to prove the linearizability of the data structure.

Theorem 3.5 (OCC-ABtree Invariants). *The following invariants are true at every configuration in any execution of the OCC-ABtree.*

1. *The reachable nodes form a relaxed (a,b)-tree.*
2. *The key range of a node that was once reachable is constant.*
3. *A node that is not reachable contains the same keys and values that it contained when it was last reachable and unlocked (i.e. updates do not both unlink and modify a node).*
4. *A key appears at most once in a leaf.*
5. *If a node was once reachable, and is currently unmarked, it is still reachable.*
6. *If a node is unlocked and was once reachable, its size field matches the number of keys it contains.*

7. *The key range of n in search(key, target) contains key.*

Intuitively, invariants 1 to 4 follow from the sequential correctness of the updates together with the guarantee that any node that might be replaced or modified is locked and reachable until the update occurs. The sequential correctness of the updates (i.e. their correctness in a single-threaded execution) can be established by inspection of the pseudocode, so we do not prove it in detail. We briefly explain the (concurrent) correctness of invariants 1 to 4. Invariants 5 and 6 are straightforward from the pseudocode.

Invariant 7 is slightly different from the others, in that it is not a *structural* invariant. Rather, it describes the correctness of one of the operations. The proof is somewhat involved, so it is proved in detail.

Proof. The invariants are satisfied at the initial state of the OCC-ABtree.

1: OCC-ABtree is a relaxed (a,b)-tree. The updates to the tree are the same as those described by Larsen and Fagerberg in [35]. They prove that, if these updates occur atomically, the tree is always a relaxed (a,b)-tree. Thus, the remainder of the proof simply shows that each update affects the tree atomically. This requires proving that for each update:

- There is a single step at which the update appears to take place
- The update is correct

The first condition is simple. Simple inserts and deletes appear when the modified leaf is unlocked, by Definition 3.2. All other updates only change a single pointer of a reachable node (to point to the update's newly created nodes).

For the second condition, assume that the updates are sequentially correct. This is easily verifiable by examining the pseudocode in this paper and comparing it to the pseudocode in [35]. To establish concurrent correctness from sequential correctness, it is sufficient to show that the update occurs on the correct data (i.e. on the correct node and with any preconditions of the sequential code satisfied), the update affects data that is actually in the tree, and the data used to construct the update does not change while the update is being constructed.

An insert(key, val) operation uses the search function to find the leaf in which to insert. By invariant 7, the leaf's key range contains key. By invariant 1 (this invariant), the tree is a relaxed (a,b)-tree and thus a search tree, so there is a unique *reachable* leaf whose key range contains key. Finally, this leaf is reachable if the insert returns, because insert verifies that the leaf is not marked. Thus, the insert occurs in the correct leaf. A similar argument holds for delete.

The sequential code for the rebalancing steps has some preconditions. The fixTagged rebalancing step requires that the node is tagged, but its parent and grandparent node are

not. `fixUnderfull` requires that none of the involved nodes are tagged, the parent node is not underfull, and the target node is underfull. This is explicitly verified in both functions. Thus, the rebalancing steps also act on the correct data.

Each update verifies that all involved nodes are not marked before performing its update. If the node is not marked, it is in the tree until the update itself unlinks the node, by invariant 5. Moreover, any children of the node are also in the tree by Definition 3.1. Thus, the data used to construct the update is actually in the tree.

Finally, the locks acquired by each update guarantee that any data involved in the update is constant until the locks are released.

2: Constant key range. We must examine places where existing nodes are attached to a new parent and ensure that the key range of all descendent nodes remains the same. This happens in `fixTagged` and `fixUnderfull`. In either function, the routing keys surrounding any pointer that is not removed remain the same before and after the update. Thus, the key range of the pointed to node does not change. This holds for leftmost and rightmost children of a node too, since the grandparent’s key range does not change (by this invariant), and the new parent’s key range is the same as the old parent’s.

3: Unreachable nodes contain the same keys and values as they did when they were last reachable and unlocked. Updates that unlink a node first lock it, then unlink it, then unlock it, *without* changing the node’s keys or values.

4: No duplicate key. Insert operations read the whole leaf while it is locked before attempting to insert a key, so a duplicate key is never inserted. The leaf that an insert operation tries to insert into is correct by invariant 7.

`fixUnderfull` does not create duplicate keys when merging two leaves because there is a unique leaf whose key range contains a given key, and any keys in that key range are only present in that leaf (invariant 1). Thus, a key can only be in one of the two leaves and so cannot appear twice in the merged node.

7: Search correctness. The search maintains the invariant that the key range of the node it is currently reading contains the search key. Call this node n . The invariant is satisfied for the entry node, since its key range is the entire key space. Since the routing keys of an internal node partition its key range, there is a unique child whose key range contains the search key.

Let c be the child followed by the search after reading node n . Even if n is not in the tree when the pointer to c is read by the search, c must have been set as a child of n while n was in the tree, since only nodes in the tree are modified (invariant 3). Thus, at the time that n was in the tree and had c as its child, the key range of c contained the search key (Definition 3.3). Since the key range of a node is constant (invariant 2), the key range of c still contains the search key. \square

With these invariants proved, the linearizability of the operations can now be established.

3.3.3 Linearizability of find. The leaf at which `search(key, target)` terminates was, at some point, the unique leaf that might have contained `key`, by invariant 7.

The search only returns if, during an interval when the leaf was unlocked, it either finds the search key and reads its value or it reads the entire leaf and does not find it. In the former case, we know that this key is unique in the leaf (invariant 4). Since the leaf was unlocked for the entire interval and nodes are not modified while they are unlocked, the result value of `find` is correct for the leaf state in that interval.

If the leaf was in the tree at any point in this interval, `find` may linearize at that point and be correct. If the leaf was never in the tree during the unlocked interval, `find` linearizes at the point just before the leaf was unlinked. The leaf must have been marked when it was unlinked; so, by invariant 3, the value returned by `find` is the same as it would be if the `find` occurred atomically just before the node was unlinked.

In this case, we must show that the `find` was concurrent with the point when the leaf was unlinked. Theorem 3.6 implies that the leaf must have been in the tree at some point during `find`’s invocation of `search`. Since (by assumption) the node was not in the tree in the unlocked interval, the `find` must have been concurrent with its unlinking. Note that the search procedure does not actually read the marked bit to check whether a leaf is in the tree; it is only described here for analysis.

Theorem 3.6. *Each node search visits was in the tree at some time during the search.*

Proof. The statement is true for the root. If the root is a leaf, the proof is complete. Otherwise, `search` reads a child pointer from the root. We now show that any child pointer read from a node n which was in the tree at some time during the search points to a child which was also in the tree at some time during the search.

If n is still in the tree at the time the child pointer is read, the child pointed to is also in the tree at that point by Definition 3.2. Thus the child is also in the tree at some point during the search.

Otherwise, n must have been (atomically) unlinked by some update U at time t . The search was concurrent with the unlinking of n since n was in the tree at some point during the search (by assumption) and n was not in the tree when the search read the child pointer.

By invariant 3, the pointers of n point to its children just before it was unlinked at time t . Thus, the child followed by the search procedure was in the tree at some time during the search as well (namely, at t). \square

3.3.4 Linearizability of insert and delete. There are four possible linearization points for an `insert(key, val)` operation. Note that in the final iteration of the `RETRY` loop, the leaf l that the `insert` locks is the *unique* reachable leaf that might contain key since the OCC-ABtree is a search tree (invariant 1), key is in l 's key range (invariant 7), and l is not marked (invariant 5).

An `insert` that succeeds in its search is linearized in the same way as a `find` operation. The return value of the search is the value associated with the key (by the correctness of `find`) and is the correct value to return for `insert`.

An `insert` that finds key in the leaf l after acquiring the l 's lock (and thus does not modify l) may linearize at any point while the l 's lock is held because while l is locked, the key cannot be removed from l , the key's associated value cannot change, and l cannot be unlinked (since unlinking l would require marking it). Since the leaf's version is even, the associated value is the correct return value according to Definition 3.2.

An `insert` that inserts a key-value pair into a non-full leaf l linearizes at its second increment of l 's version (which marks the modification as complete). The key is not in the OCC-ABtree *before* the linearization point since the `insert` read l while it was locked without finding the key, and l is the unique reachable leaf that might contain l . The key is in the OCC-ABtree after the linearization point (according to Definition 3.2) because the key is added to l , l is still reachable, and l 's version is even.

For splitting inserts, searches can observe the change as soon as the pointer to the new subtree is written in the parent, since searches do not read locks on internal nodes. Thus, splitting inserts *must* linearize at the write to the parent node. Suppose a splitting insert writes the new pointer into the parent node p at time t . Let l be the leaf that was split and replaced by a tagged node t with children l_1 and l_2 . The inserted key is not in the OCC-ABtree *before* the write to p since the `insert` reads l while it is locked and does not find the key (and l is the unique reachable leaf that might contain the key). *After* the write to p , the inserted key is in the tree because it is in either l_1 or l_2 , both of which are reachable because p is unmarked and thus reachable (invariant 5). The other keys in l are not affected by splitting inserts since they are placed in one of l_1 or l_2 by the splitting insert.

The returned value of \perp is correct in the above two cases, since the `insert` succeeded. The linearization of deletes and justification of return values is similar to the first three cases above.

3.3.5 Deadlock freedom. 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 $\leq \text{rec.ver}$, and
- C2. O' returns after `l.ver` $>$ `rec.ver`

⁴For 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  $\perp$ 
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  $\neq$  ver2
216
217         if startVer  $\leq$  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

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.)

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, \perp >`. 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),

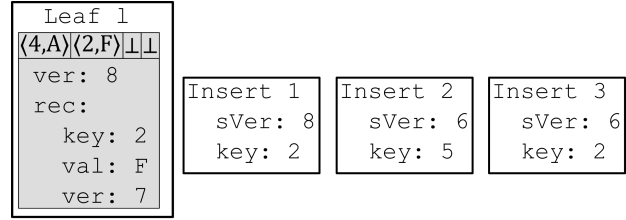


Figure 11. Consider the state of leaf l as shown. $l.rec$ stores the Elim-ABtree of a completed simple insert `insert(2, F)`. Consider three (independent) inserts that are attempting to insert in l and are all at line 217. Insert 1 cannot eliminate itself with `rec` since the version of the leaf it read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

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 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, \perp >`. 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 `\perp` (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 finds 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 \perp so the key-value pair is *not* logically in the tree. For a successful delete, `key` must be flushed after it is set to \perp . The delete occurs atomically when the key field is equal to \perp 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 [54]). 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.

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.

5.1 p-OCC-ABtree Correctness

This section begins by providing a definition to link the physical state of the p-OCC-ABtree to its abstract contents. It then mentions some invariants which hold for the p-OCC-ABtree; these are analogous to the invariants of the OCC-ABtree and can similarly be used to show that the p-OCC-ABtree is strictly-linearizable.

5.1.1 Definitions.

Definition 5.1 (p-Reachable node). A node is **p-reachable** (short for persistently reachable) if it can be reached from the entry node by following child pointers in *persistent* memory.

Definition 5.2 (Recovering). The system is said to be **recovering** from the time when a crash occurs until the time when the recovery procedure returns.

In strict linearizability, every operation that is concurrent with a crash must either be linearized before the crash or be removed from the execution. Any simple insert or successful delete that has flushed a key will be recovered (and thus cannot be removed from the execution). These operations must therefore be linearized before the crash, even if they have not yet incremented the version for the second time. This is reflected in the definition below, and in the changes to the linearization points in the following section.

Definition 5.3. Let l be a p-reachable node. A key k (not equal to \perp) is **in the p-OCC-ABtree** if either

1. The system is recovering and k is in l 's keys array OR
2. The system is not recovering and k was in l 's keys array when l 's version (in *volatile* memory) was last even

Furthermore, if key k is the i th key in l , the value associated with k is `l.vals[i]`.

If the system is not recovering, Definition 5.3 is similar to the definition of a key being in the OCC-ABtree. That is, keys and values are logically added or removed from the tree when the version number is incremented to an even number. If the system is recovering, however, every key in a p-reachable node is in the tree (the version is ignored).

5.1.2 Invariants.

Theorem 5.4 (p-OCC-ABtree Invariants). *The p-OCC-ABtree satisfies the following invariants, which are analogous to the OCC-ABtree invariants:*

1. *The p-reachable nodes form a relaxed (a,b)-tree.*
2. *The key range of a node that was once p-reachable is constant.*
3. *A node that is not p-reachable contains the same keys and values that it did when it was last p-reachable and unlocked (i.e. updates do not both unlink and modify a node).*
4. *A key appears at most once in a leaf.*

5. If a node was once p -reachable, and is currently unmarked, it is p -reachable.
6. If a node is unlocked and was once reachable, its size field matches the number of keys it contains.
7. The key range of n in $\text{search}(\text{key}, \text{target})$ contains key.

Proof. The proofs of most of these invariants are similar to the proofs in Section 3.3. The proof for invariant 1 requires an additional explanation of why every node used by an update was once p -reachable.

Proof of invariant 1. Recall that to prove the p -OCC-ABtree is a relaxed (a,b)-tree, it suffices show that for each update:

- There is a single step at which the update appears to take place
- The update is correct

The first condition holds for the reasons laid out in the previous section on atomic updates.

The second condition is largely the same as the proof in the OCC-ABtree. However, that proof uses invariant 5, which requires showing that the nodes traversed in the search were all reachable at some time. This was trivial in the case of the OCC-ABtree (because the nodes are reached by following child pointers), but is not trivial in the p -OCC-ABtree, which uses p -reachability.

We will show that every node traversed by search in the p -OCC-ABtree was p -reachable at some time. Assume that every node traversed by a search until node n is p -reachable. If n is the entry node, it is p -reachable by definition.

Otherwise, the search reached n by following an unmarked pointer from a node p . We will show that there exists a time t when p was p -reachable and contained the unmarked pointer to n . If p was p -reachable when it read the unmarked pointer to n , t is the time of the read. Otherwise, invariant 3 guarantees that p 's pointers have not been modified since it was last p -reachable. Thus, when p was last p -reachable, it contained an unmarked pointer to n . In this case, t is the time when p was last reachable.

Finally, notice that there are two ways p could contain an unmarked pointer to n . The first way is if p contained a pointer to n when it was created. In this case, since p was flushed before being linked into the tree, its pointer to n is persisted. Otherwise, the pointer was first introduced to p by an update U as a marked pointer. This update must have flushed the pointer before unmarking it. In either case, the unmarked pointer was in persistent memory by time t .

At time t , p was p -reachable and contained a pointer to n in persistent memory. Thus, n was p -reachable at time t .

The remainder of the proof of is similar to the proof for the OCC-ABtree. \square

5.1.3 Strict linearizability. The p -OCC-ABtree has slightly different linearization points than the OCC-ABtree, to deal with the different definition of when a key is in the tree.

Operations which do not modify the tree are linearized as in the OCC-ABtree. Splitting inserts in the p -OCC-ABtree are linearized when the pointer to the new nodes is flushed to persistent memory (instead of it is written to volatile memory).

Simple inserts and successful deletes linearize differently depending on whether or not they are interrupted by a crash. When not interrupted by a crash, simple inserts and successful deletes have the same linearization points as they did in the OCC-ABtree: the increment of the leaf's version (in volatile memory) to an even number. Recall that this linearization point is chosen to support publishing elimination.

To see why we cannot linearize the same way when interrupted by a crash, consider the following scenario. Suppose a simple insert or successful delete that flushes key (thus making its change persistent) but does increment the version to an even number before a crash. The recovery procedure would recover this key-value pair, even though the operation was not linearized.

To solve this problem, we linearize these operations *at the crash*.

Theorem 5.5. *The p -OCC-ABtree is strictly-linearizable.*

Proof. Note that these linearization points all occur after an operation's invocation and before its response or crash. We must show that performing each operation at its linearization point (and returning the appropriate value) correctly affects the contents of the abstract dictionary according to Definition 5.3. Let E be an arbitrary execution of the p -OCC-ABtree. We prove that E is strictly-linearizable by induction.

Suppose the prefix of E up to the beginning of the i th era of operations (including the recovery procedure after the $i - 1$ th crash, if $i > 1$) is strictly-linearizable, and that the p -OCC-ABtree satisfies all invariants. We show that the prefix up to the beginning of the $i + 1$ th era of operations is strictly-linearizable and the p -OCC-ABtree recovered after the i th crash satisfies all invariants.

We do this by breaking up the execution fragment from the beginning of the i th era of operations to the beginning of the $i + 1$ th era of operations into three parts: the execution fragment before the i th crash, the i th crash, and the recovery after the i th crash. We show that each fragment is strictly-linearizable by showing that the operations correctly modify the abstract dictionary. Note that the concatenation of strictly-linearizable execution fragments is strictly-linearizable, by the *locality* property of strict linearizability.

Before the i th crash. We consider the tree operations performed from the beginning of the i th era until (but not including) the i th crash. The linearizability arguments for these operations is analogous to the arguments established for linearizability in the OCC-ABtree: the linearization points used in this section are all analogous to the OCC-ABtree's linearization points, the p -OCC-ABtree satisfies analogous

invariants, and the definition of a key being in the p-OCC-ABtree is Definition 5.3.2 (which is analogous to the OCC-ABtree’s definition of a key being in the tree).

At the i th crash. At the time of the crash, the definition of a key being in the tree changes from Definition 5.3.2 to Definition 5.3.1. It must be shown that the keys in the tree after the crash are exactly those that were in the tree before the crash, plus any that were inserted by a simple insert linearized at the crash and minus any that were deleted by a successful delete linearized at the crash.

First consider the case when a key k is in the tree after a crash. That is, there exists some p-reachable leaf l such that $k = l.keys[i]$ (for some index i).

If k was also in the tree before the crash (according to Definition 5.3.2), it is only correct for k to be in the tree after the crash if no delete of k linearized at the crash. This is indeed the case, since a delete of k that linearized at the crash would have set $l.keys[i]$ to \perp and flushed \perp . But, by assumption, k is in the keys array of l after the crash.

Otherwise, if k was not in the tree before the crash, it is only correct for k to be in the tree after the crash if an insert of k *did* linearize at the crash. This is true. Since k was not in the tree before the crash but l was p-reachable and contained k , the l ’s version must have been odd at the crash (according to Definition 5.3.2). Thus, there must have been an ongoing insert that inserted k at the time of the crash. Since the crash occurred after the flush of k but before the version was incremented to an even number, this insert linearized at the crash.

A similar argument shows that k is *not* in the tree after a crash if and only if k was either deleted at the crash or was not in the tree before the crash (and was not inserted at the crash).

p-OCC-ABtree invariants 1-4 and 7 are maintained during a crash since they only describe persisted data. Invariants 5 and 6 might be incorrect since they refer to volatile fields. However, they are restored by the recovery procedure.

After the i th crash (recovery). The recovery procedure does not affect the set of p-reachable nodes or their keys or values, so the set of keys in the tree is fixed while the system is recovering. By the time the recovery procedure returns, all p-reachable nodes’ versions are 0, and thus the key in the tree is the same according to Definitions 5.3.1 and 5.3.2.

Additionally, all p-OCC-ABtree invariants are satisfied by the time the recovery procedure returns. p-OCC-ABtree invariants 1-4 and 7 were correct before recovery, and the recovery procedure fixes the volatile fields, which ensures that invariants 5 and 6 hold by the time it returns.

Thus, the execution up to the beginning of the operation in the $i + 1$ th era is strictly-linearizable, and the p-OCC-ABtree satisfies all invariants. \square

The proofs for the p-Elim-ABtree is similar. Note that elimination does not conflict with the change of linearizing some

operations at a crash. In both the p-OCC-ABtree and the p-Elim-ABtree, an operation O_e is only eliminated *after* the successful operation O_p has executed its second increment of the leaf’s version. Any simple insert or successful delete has linearized by this time (and a future crash does not change this fact).

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 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, DGT15, OLC-ART, 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).

System. Our volatile memory experiments (Figure 14, Figure 16) 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 and FPTree (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

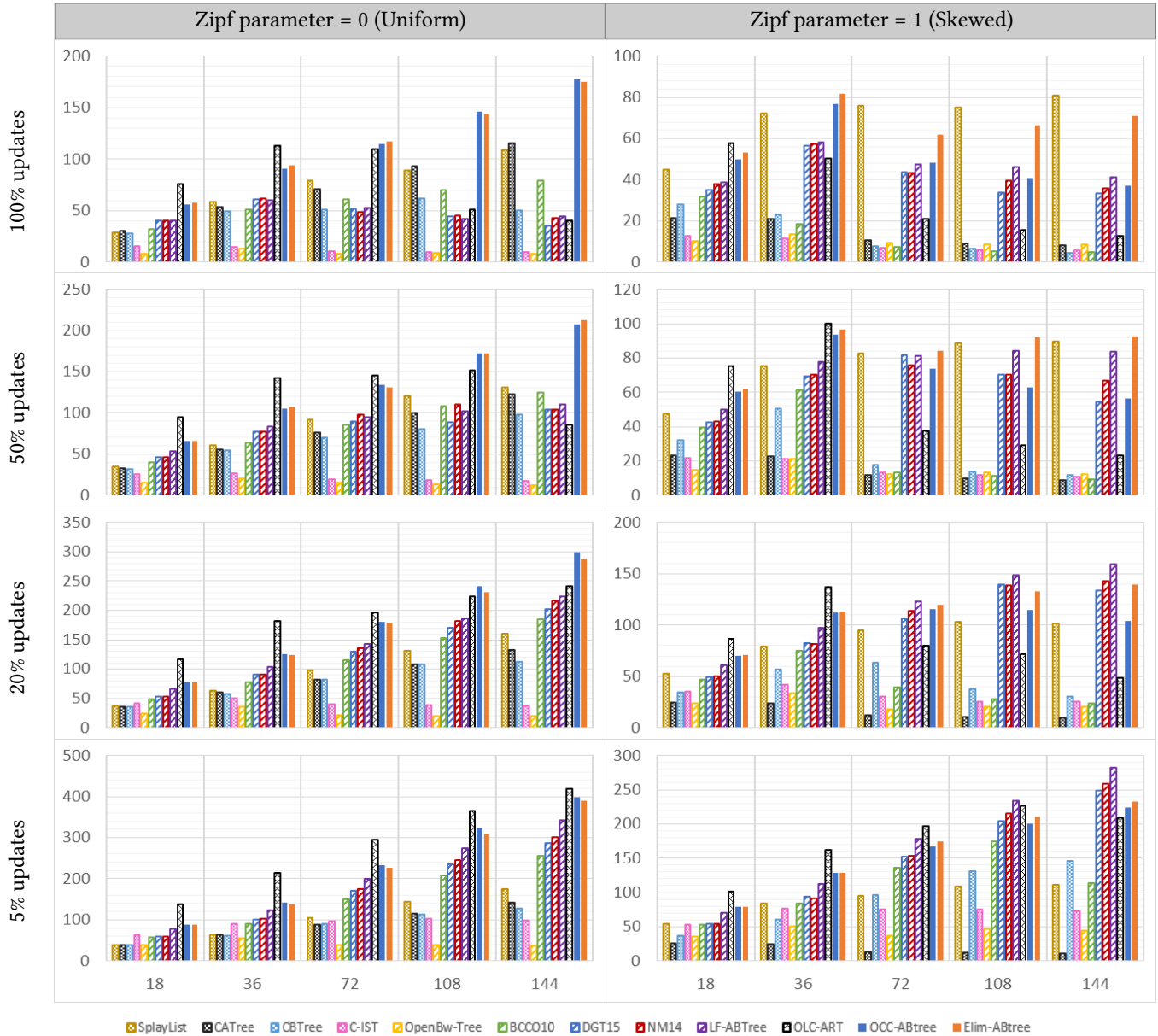


Figure 12. SetBench microbenchmark with 10K keys. x-axis: number of threads. y-axis: operations per μs .

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, CBTre, 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 CBTre and SplayList fell short of our expectations on the Zipfian workload. We expected that splaying would

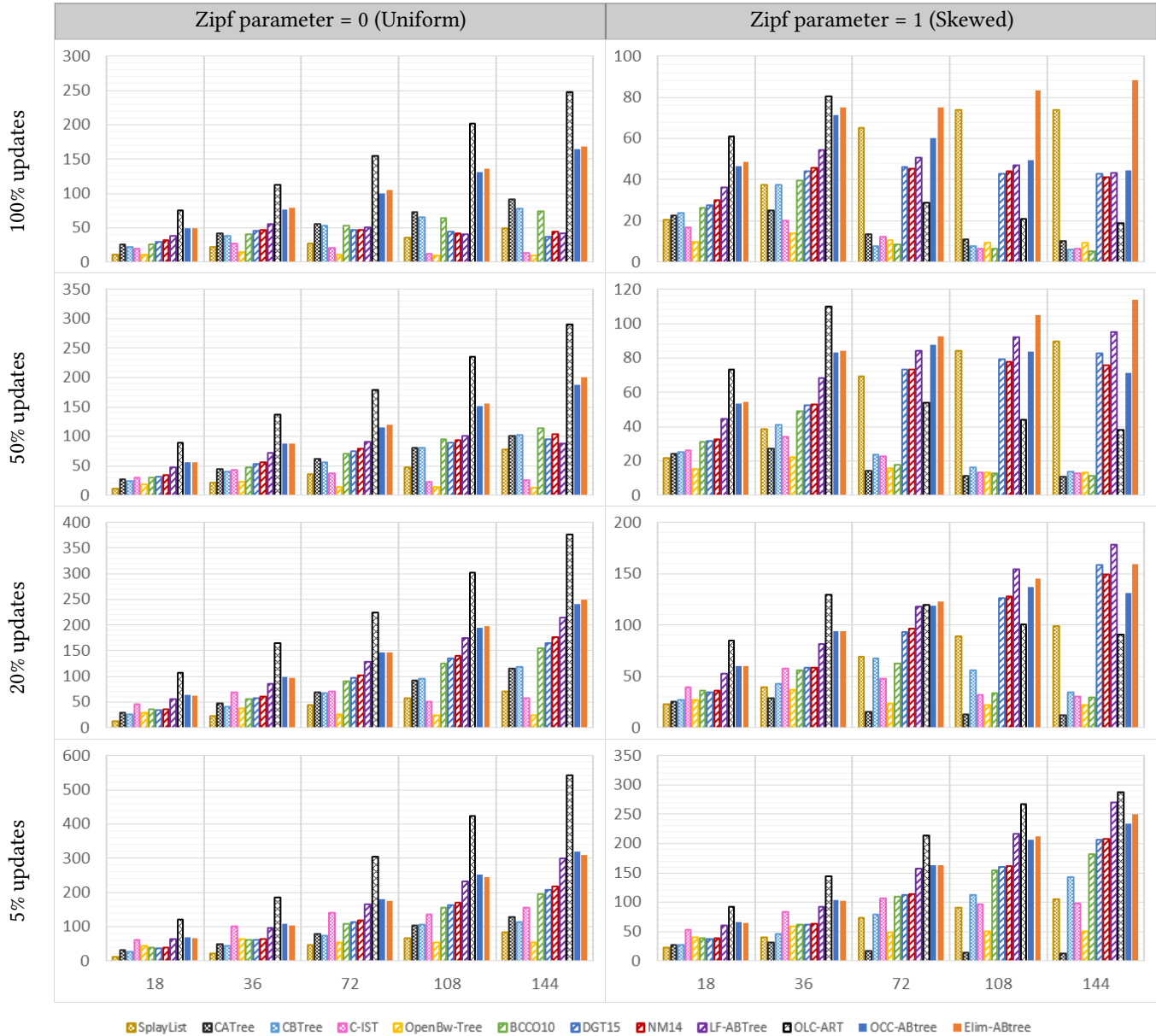


Figure 13. SetBench microbenchmark with 100K keys. x-axis: number of threads. y-axis: operations per μ s.

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

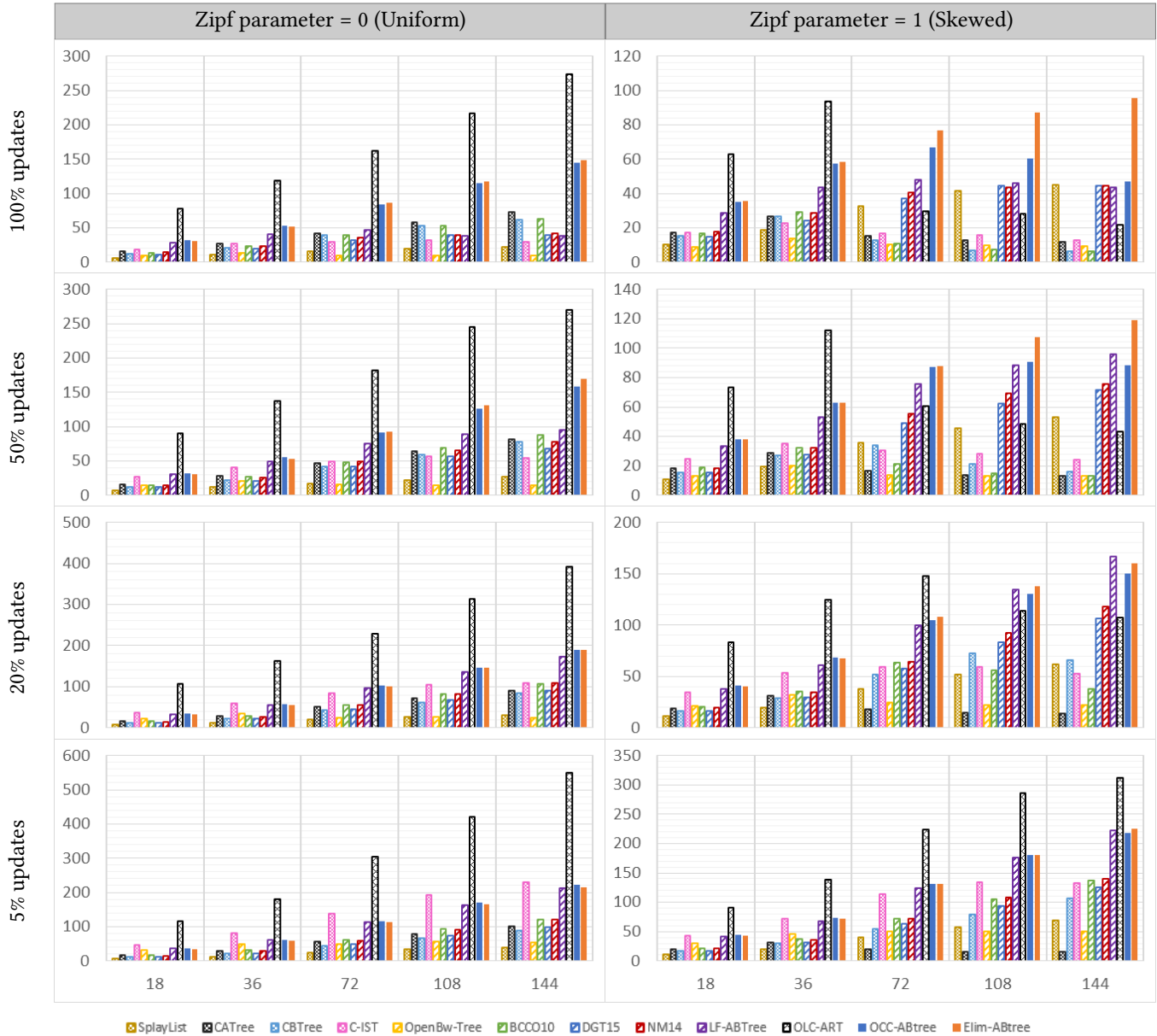


Figure 14. SetBench microbenchmark with 1M keys. x-axis: number of threads. y-axis: operations per μ s.

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.

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 16). Figure 16 does not contain the SplayList since it does not reclaim memory and consequently caused

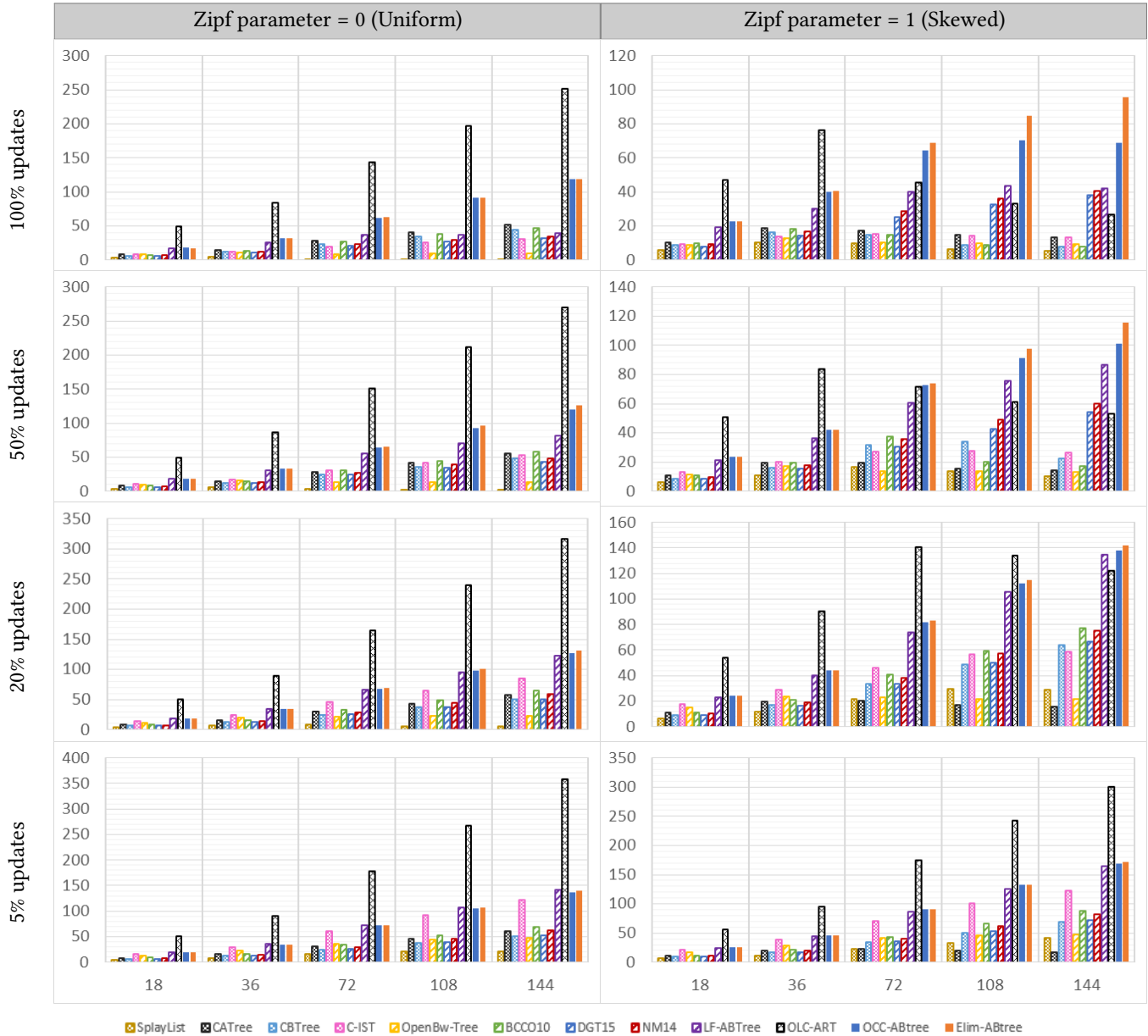


Figure 15. SetBench microbenchmark with 10M keys. x-axis: number of threads. y-axis: operations per μs .

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 implementations were from [41]). However, these implementations do not reclaim memory.

Figure 17 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 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

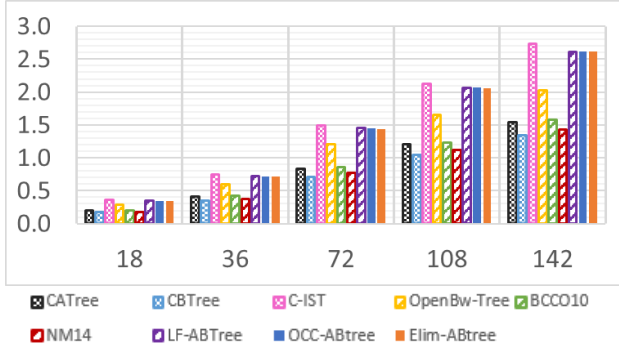


Figure 16. YCSB throughput on Workload A. x-axis: number of threads. y-axis: transactions per μ s.

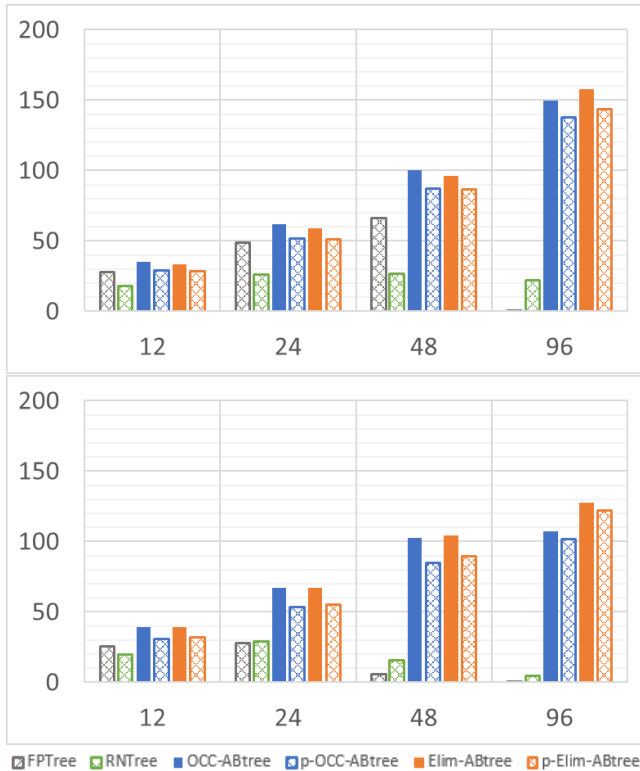


Figure 17. Comparing with other persistent trees: SetBench microbenchmark with 1M keys, 50% updates (25% insert and 25% delete). Left: Uniform access distribution. Right: Zipfian access distribution (with Zipf factor 1). x-axis: number of threads. y-axis: operations per μ s.

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.

Update rate:	Uniform			Zipfian		
	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.

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 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: DGEGR-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. Asynchronous 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 Drachler-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 (IPSP '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] Arunmozhi 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/ISPD.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/10.1145/3341301.3359635>

[//doi.org/10.1145/215399.215419](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] 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 Storage Technologies* (San Jose, California) (*FAST'11*). USENIX Association, USA, 5.
- [54] Tianzheng Wang, Justin Levandoski, and Per-Ake Larson. 2018. Easy lock-free indexing in non-volatile memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, Paris, France, 461–472.
- [55] 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>
- [56] 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>
- [57] 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.5733351>.

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.5733351>.
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.