A Practical Concurrent Binary Search Tree

Nathan G. Bronson Jared Casper Hassan Chafi Kunle Olukotun

Computer Systems Laboratory
Stanford University
{nbronson, jaredc, hchafi, kunle}@stanford.edu

Abstract

We propose a concurrent relaxed balance AVL tree algorithm that is fast, scales well, and tolerates contention. It is based on optimistic techniques adapted from software transactional memory, but takes advantage of specific knowledge of the the algorithm to reduce overheads and avoid unnecessary retries. We extend our algorithm with a fast linearizable `clone` operation, which can be used for consistent iteration of the tree. Experimental evidence shows that our algorithm outperforms a highly tuned concurrent skip list for many access patterns, with an average of 39% higher single-threaded throughput and 32% higher multi-threaded throughput over a range of contention levels and operation mixes.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features – Concurrent programming structures; E.1 [*Data Structures*]: Trees; D.1.3 [*Programming Techniques*]: Concurrent Programming – Parallel programming

General Terms Algorithms, Performance

Keywords Optimistic Concurrency, Snapshot Isolation

1. Introduction

The widespread adoption of multi-core processors places an increased focus on data structures that provide efficient and scalable multi-threaded access. These data structures are a fundamental building block of many parallel programs; even small improvements in these underlying algorithms can provide a large performance impact. One widely used data structure is an ordered map, which adds ordered iteration and range queries to the key-value association of a map. In-memory ordered maps are usually implemented as either skip lists [19] or self-balancing binary search trees.

Research on concurrent ordered maps for multi-threaded programming has focused on skip lists, or on leveraging software transactional memory (STM) to manage concurrent access to trees [3, 12]. Concurrent trees using STM are easy to implement and scale well, but STM introduces substantial baseline overheads and performance under high contention is still an active research topic [2]. Concurrent skip lists are more complex, but have dependable performance under many conditions [9].

©ACM, (2009). This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in *PPoPP '10: Proceedings of the 15th ACM SIGPLAN Symposium on Principals and Practice of Parallel Programming*, January, 2010. http://doi.acm.org/10.1145/nnnnnnn.nnnnnnn

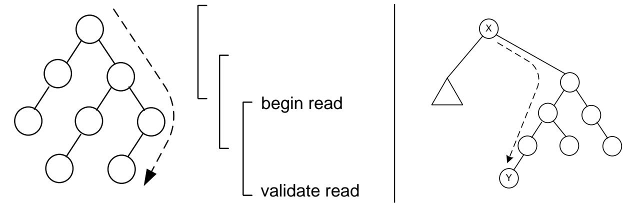

Figure 1. a. Hand-over-hand optimistic validation. **b.** Finding the successor for deletion.

In this paper we present a concurrent relaxed balance AVL tree. We use optimistic concurrency control, but carefully manage the tree in such a way that all atomic regions have fixed read and write sets that are known ahead of time. This allows us to reduce practical overheads by embedding the concurrency control directly. It also allows us to take advantage of algorithm-specific knowledge to avoid deadlock and minimize optimistic retries. To perform tree operations using only fixed sized atomic regions we use the following mechanisms: search operations overlap atomic blocks as in the hand-over-hand locking technique [5]; mutations perform rebalancing separately; and deletions occasionally leave a routing node in the tree. We also present a variation of our concurrent tree that uses lazy copy-on-write to provide a linearizable `clone` operation, which can be used for strongly consistent iteration.

Our specific contributions:

- We describe hand-over-hand optimistic validation, a concurrency control mechanism for searching and navigating a binary search tree. This mechanism minimizes spurious retries when concurrent structural changes cannot affect the correctness of the search or navigation result (Section 3.3).
- We describe partially external trees, a simple scheme that simplifies deletions by leaving a routing node in the tree when deleting a node that has two children, then opportunistically unlinking routing nodes during rebalancing. As in external trees, which store values only in leaf nodes, deletions can be performed locally while holding a fixed number of locks. Partially external trees, however, require far fewer routing nodes than an external tree for most sequences of insertions and deletions (Section 3.5).
- We describe a concurrent partially external relaxed balance AVL tree algorithm that uses hand-over-hand optimistic validation, and that performs all updates in fixed size critical regions (Section 3).
- We add copy-on-write to our optimistic tree algorithm to provide support for an atomic `clone` operation and snapshot isolation during iteration (Section 3.10).

- We show that our optimistic tree outperforms a highly-tuned concurrent skip list across many thread counts, contention levels, and operation mixes, and that our algorithm is much faster than a concurrent tree implemented using an STM. Our algorithm's throughput ranges from 13% worse to 98% better than the skip list's on a variety of simulated read and write workloads, with an average multi-threaded performance improvement of 32%. We also find that support for fast cloning and consistent iteration adds an average overhead of only 9% to our algorithm (Section 5).

2. Background

An AVL tree [1] is a self-balancing binary search tree in which the heights of the left and right child branches of a node differ by no more than one. If an insertion to or deletion from the tree causes this balance condition to be violated then one or more rotations are performed to restore the AVL invariant. In the classic presentation, nodes store only the difference between the left and right heights, which reduces storage and update costs. Balancing can also be performed if each node stores its own height.

The process of restoring the tree invariant becomes a bottleneck for concurrent tree implementations, because mutating operations must acquire not only locks to guarantee the atomicity of their change, but locks to guarantee that no other mutation affects the balance condition of any nodes that will be rotated before proper balance is restored. This difficulty led to the idea of relaxed balance trees, in which the balancing condition is violated by mutating operations and then eventually restored by separate rebalancing operations [10, 14, 16]. These rebalancing operations involve only local changes. Bougé et al. proved that any sequence of localized application of the AVL balancing rules will eventually produce a strict AVL tree, even if the local decisions are made with stale height information [6].

Binary search trees can be broadly classified as either internal or external. Internal trees store a key-value association at every node, while external trees only store values in leaf nodes. The non-leaf nodes in an external tree are referred to as *routing nodes*, each of which has two children. Internal trees have no routing nodes, while an external tree containing n values requires n leaves and $n - 1$ routing nodes.

Deleting a node from an internal tree is more complicated than inserting a node, because if a node has two children a replacement must be found to take its place in the tree. This replacement is the successor node, which may be many links away (y is x's successor in Figure 1.b). This complication is particularly troublesome for concurrent trees, because this means that the critical section of a deletion may encompass an arbitrary number of nodes. The original delayed rebalancing tree side-stepped this problem entirely, supporting only `insert` and `search` [1]. Subsequent research on delayed rebalancing algorithms considered only external trees. In an external tree, a leaf node may always be deleted by changing the link from its grandparent to point to its sibling, thus splicing out its parent routing node (see Figure 7.b).

While concurrent relaxed balance tree implementations based on fine-grained read-write locks achieve good scalability for disk based trees, they are not a good choice for a purely in-memory concurrent tree. Acquiring read access to a lock requires a store to a globally visible memory location, which requires exclusive access to the underlying cache line. Scalable locks must therefore be striped across multiple cache lines to avoid contention in the coherence fabric [15], making it prohibitively expensive to store a separate lock per node.

Optimistic concurrency control (OCC) schemes using version numbers are attractive because they naturally allow invisible readers, which avoid the coherence contention inherent in read-write locks. Invisible readers do not record their existence in any globally visible data structure, rather they read version numbers updated by writers to detect concurrent mutation. Readers 'optimistically' assume that no mutation will occur during a critical region, and then retry if that assumption fails. Despite the potential for wasted work, OCC can provide for better performance and scaling than pessimistic concurrency control [20].

Software transactional memory (STM) provides a generic implementation of optimistic concurrency control, which may be used to implement concurrent trees [12] or concurrent relaxed balance trees [3]. STM aims to deliver the valuable combination of simple parallel programming and acceptable performance, but internal simplicity is not the most important goal of a data structure library.[1] For a widely used component it is justified to expend a larger engineering effort to achieve the best possible performance, because the benefits will be multiplied by a large number of users.

STMs perform conflict detection by tracking all of a transaction's accesses to shared data. This structural validation can reject transaction executions that are semantically correct. Herlihy et al. used early release to reduce the impact of this problem [12]. Early release allows the STM user to manually remove entries from a transaction's read set. When searching in a binary tree, early release can mimic the effect of hand-over-hand locking for successful transactions. Failed transactions, however, require that the entire tree search be repeated. Elastic transactions require rollback in fewer situations than early release and do not require that the programmer explicitly enumerate entries in the read set, but rollback still requires that the entire transaction be reexecuted [8].

Skip lists are probabilistic data structures that on average provide the same time bounds as a balanced tree, and have good practical performance [19]. They are composed of multiple levels of linked lists, where the nodes of a level are composed of a random subset of the nodes from the next lower level. Higher lists are used as hints to speed up searching, but membership in the skip list is determined only by the bottom-most linked list. This means that a concurrent linked list algorithm may be augmented by lazy updates of the higher lists to produce a concurrent ordered data structure [9, 18]. Skip lists do not support structural sharing, so copy-on-write cannot be used to implement fast cloning or consistent iteration. They can form the foundation of an efficient concurrent priority queue [21].

3. Our Algorithm

We present our concurrent tree algorithm as a map object that supports five methods: `get`, `put`, `remove`, `firstNode`, and `succNode`. For space reasons we omit practical details such as user-specified `Comparators` and handling of `null` values. The `get`(k) operation returns either v, where v is the value currently associated with k, or `null`; `put`(k, v) associates k and a non-null v and returns either v_0, where v_0 is the previous value associated with k, or `null`; `remove`(k) removes any association between k and a value v_0, and returns either v_0 or `null`; `firstNode()` returns a reference to the node with the minimal key; and `succNode`(n) returns a reference to the node with the smallest key larger than n`.key`. The `firstNode` and `succNode` operations can be used trivially to build an ordered iterator. In Section 4 we will show that `get`, `put`, and `remove` are linearizable [13]. We will discuss optimistic hand-over-hand locking in the context of `get` (Section 3.3) and partially external trees in the context of `remove` (Section 3.5).

Our algorithm is based on an AVL tree, rather than the more popular red-black tree [4], because relaxed balance AVL trees are

[1] Here we are considering STM as an internal technique for implementing a concurrent tree with a non-transactional interface, not as a programming model that provides atomicity across multiple operations on the tree.

```
 1  class Node<K,V> {
 2      volatile int height;
 3      volatile long version;
 4      final K key;
 5      volatile V value;
 6      volatile Node<K,V> parent;
 7      volatile Node<K,V> left;
 8      volatile Node<K,V> right;
 9      ...
10  }
```

Figure 2. The fields for a node with key type `K` and value type `V`.

```
11  static long Unlinked = 0x1L;
12  static long Growing = 0x2L;
13  static long GrowCountIncr = 1L << 3;
14  static long GrowCountMask = 0xffL << 3;
15  static long Shrinking = 0x4L;
16  static long ShrinkCountIncr = 1L << 11;
17  static long IgnoreGrow = ~(Growing | GrowCountMask);
```

Figure 3. Version manipulation constants.

less complex than relaxed balance red-black trees. The AVL balance condition is more strict, resulting in more rebalancing work but smaller average path lengths. Pfaff [17] characterizes the workloads for which one tree performs better than the other, finding no clear winner. Our contributions of hand-over-hand optimistic validation and local deletions using partially external trees are also applicable to relaxed balance red-black trees. Lookup, insertion, update, and removal are the same for both varieties of tree. Only the post-mutation rebalancing (Section 3.6) is affected by the choice.

3.1 The data structure: `Node`

The nodes that compose our tree have a couple of variations from those of a normal AVL tree: nodes store their own height rather than the difference of the heights of the children; nodes for a removed association may remain in the tree with a `value` of `null`; and nodes contain a version number used for optimistic concurrency control. Figure 2 shows the fields of a node. All fields except for `key` are mutable. `height`, `version`, `value`, `left`, and `right` may only be changed while their enclosing node is locked. `parent` may only be changed while the parent node is locked (both the old and the new parent must be locked). The delayed balancing of our tree is a property of the algorithm, and is not visible in the type signature.

For convenience, the map stores a pointer to a *root holder* instead of the root node itself. The root holder is a `Node` with no key or value, whose right child is the root. The root holder allows the implementation to be substantially simplified because it never undergoes rebalancing, never triggers optimistic retries (its `version` is always zero), and allows all mutable nodes to have a non-null parent. The map consists entirely of the `rootHolder` reference.

3.2 Version numbers

The version numbers used in our algorithm are similar to those in McRT-STM, in which a reserved 'changing' bit indicates that a write is in progress and the remainder of the bits form a counter [20]. (Our algorithm separates the locks that guard node update from the version numbers, so the changing bit is not overloaded to be a mutex as in many STMs.) To perform a read at time t_1 and verify that the read is still valid at t_2: at t_1 read the associated version number v_1, blocking until the change bit is not set; read the protected value x; then at t_2 reread the version number v_2. If $v_1 = v_2$ then x was still valid at t_2.

Our algorithm benefits from being able to differentiate between the structural change to a node that occurs when it is moved down the tree (shrunk) and up the tree (grown). Some operations are invalidated by either shrinks or grows, while others are only invalidated by shrinks. We use a single 64-bit value to encode all of the version information, as well as to record if a node has been unlinked from the tree. There is little harm in occasionally misclassifying a grow as a shrink, because no operation will incorrectly fail to invalidate as a result. We therefore overlap the shrink counter and the grow counter. We use the most significant 53 bits to count shrinks, and the most significant 61 bits to count grows. This layout causes a grow to be misclassified as a shrink once every 256 changes, but it never causes a shrink to be misclassified as a grow. The bottom three bits are used to implement an unlink bit and two change bits, one for growing and one for shrinking (Figure 3).

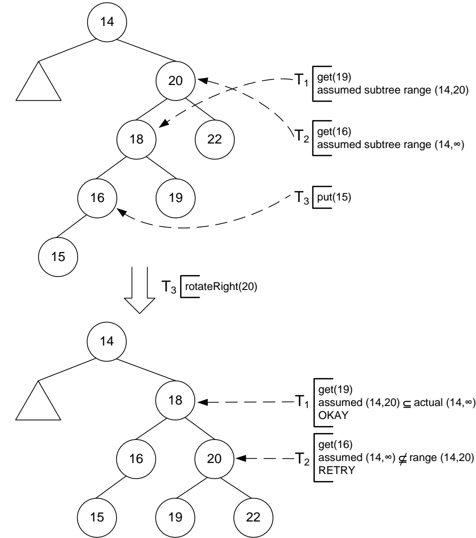

Figure 4. Two searching threads whose current pointer is involved in a concurrent rotation. The node 18 grew, so T_1's search is not invalidated. The node 20 shrank, so T_2 must backtrack.

3.3 Hand-over-hand optimistic validation: `get`(k)

If k is present in the map then `get`(k) must navigate from the root holder to the node that holds k. If k is not present in the tree then `get` must navigate to the node that would be k's parent if it were inserted. If no concurrency control is performed, a search may be led astray by a concurrent rotation. The well-known lock-based technique for handling this is hand-over-hand locking (also known as spider locks, lock chaining, chain locking, ...), which decreases the duration over which locks are held by releasing locks on nodes whose rotation can no longer affect the correctness of the search [5]. With both exclusive locks or read-write locks the root lock must be acquired by each accessor, making it a point of contention. We use optimistic validation to guard critical regions, chaining in a manner similar to hand-over-hand locking. This avoids the scalability problems of the root lock while using only fixed size critical regions (see Figure 1.a).

The key to hand-over-hand optimistic validation is to reason explicitly about the implicit state of a search, which consists of an open interval of keys that must either be absent from the entire tree or present in the current subtree. Each time that the search process performs a comparison and navigates downward, the interval is

```
18  static Object Retry = new Object();
19
20  V get(K k) {
21    return (V)attemptGet(k, rootHolder, 1, 0);
22  }
23
24  Object attemptGet(
25        K k, Node node, int dir, long nodeV) {
26    while (true) {
27      Node child = node.child(dir);
28      if (((node.version^nodeV) & IgnoreGrow) != 0)
29        return Retry;
30      if (child == null)
31        return null;
32      int nextD = k.compareTo(child.key);
33      if (nextD == 0)
34        return child.value;
35      long chV = child.version;
36      if ((chV & Shrinking) != 0) {
37        waitUntilNotChanging(child);
38      } else if (chV != Unlinked &&
39                 child == node.child(dir)) {
40        if (((node.version^nodeV) & IgnoreGrow) != 0)
41          return Retry;
42        Object p = attemptGet(k, child, nextD, chV);
43        if (p != Retry)
44          return p;
45  } } }
```

Figure 5. Finding the value associated with a k.

reduced. At all times the interval includes the target key, so if the subtree ever becomes empty we can conclude that no node with that key is present. The optimistic validation scheme only needs to invalidate searches whose state is no longer valid.

If a search that was valid when it was at node n has since traversed to a child node c of n, and the pointer from n to c has not been modified, then the search is still valid, because the subtree has not been changed. To prevent false invalidations we use separate version numbers to track changes to n.left and n.right. We reduce storage overheads by actually storing the version number that protects the link from n to c in c.version. (This requires us to traverse the link twice, once to locate the version number and once as a read that is actually protected by OCC.) To successfully navigate through a node, there must be a point where both the inbound and outbound link are valid. This is accomplished by validating the version number that protects the inbound link after both the outbound link and its version number have been read.

A search may still be valid despite a change to a child link, if every node in the tree within the computed bounds must still be contained in the subtree. Consider the scenario in Figure 4, in which a mutating thread T_3 performs a rotation while two searches are in progress. (We defer a discussion of the visible intermediate states during rotation to Section 3.7.) T_1's search points to the node that is raised by the rotation, which means that its implicit state is not invalidated. Intuitively if a rotation 'grows' the branch rooted at a node, then a search currently examining the node will not fail to find its target. Conversely, if a rotation 'shrinks' the branch under a node, then a search pointing to that node may falsely conclude that a node is not present in the tree. In the latter case the implicitly computed subtree bounds are incorrect. T_2's search points to a shrinking node, which means that it must backtrack to node 14, the previous level of the search. Changes to a child pointer may also be 'neutral' if they preserve the range of the keys contained in the subtree, as can occur during deletion.

Figure 5 shows the code for the get operation. The bulk of the work is accomplished by attemptGet. attemptGet assumes that

```
46  V put(K k, V v) {
47    return (V)attemptPut(k, v, rootHolder, 1, 0);
48  }
49
50  Object attemptPut(
51        K k, V v, Node node, int dir, long nodeV) {
52    Object p = Retry;
53    do {
54      Node child = node.child(dir)
55      if (((node.version^nodeV) & IgnoreGrow) != 0)
56        return Retry;
57      if (child == null) {
58        p = attemptInsert(k, v, node, dir, nodeV);
59      } else {
60        int nextD = k.compareTo(child.key);
61        if (nextD == 0) {
62          p = attemptUpdate(child, v);
63        } else {
64          long chV = child.version;
65          if ((chV & Shrinking) != 0) {
66            waitUntilNotChanging(child);
67          } else if (chV != Unlinked &&
68                     child == node.child(dir)) {
69            if (((node.version^nodeV) & IgnoreGrow)!=0)
70              return Retry;
71            p = attemptPut(k, v, child, nextDir, chV);
72      } } }
73    } while (p == Retry);
74    return p;
75  }
76  Object attemptInsert(
77        K k, V v, Node node, int dir, long nodeV) {
78    synchronized (node) {
79      if (((node.version^nodeV) & IgnoreGrow) != 0 ||
80          node.child(dir) != null)
81        return Retry;
82      node.setChild(dir, new Node(
83          1, k, v, node, 0, null, null));
84    }
85    fixHeightAndRebalance(node);
86    return null;
87  }
88  Object attemptUpdate(Node node, V v) {
89    synchronized (node) {
90      if (node.version == Unlinked) return Retry;
91      Object prev = node.value;
92      node.value = v;
93      return prev;
94    }
95  }
```

Figure 6. Inserting or updating the value associated with k.

the pointer to node was read under version number nodeV, and is responsible for revalidating the read at Line 40 after performing a validated read of the child pointer at Line 39. If there is no child then the final validation of the traversal to node occurs on Line 28. Note that Line 27's access to the child is not protected, it is merely used to gain access to the version number that protects the inbound pointer. The hand-off of atomically executed regions occurs between the child read and the final validation (Line 28).

attemptGet returns the special value Retry on optimistic failure, which triggers a retry in the outer call. get emulates the first half of a hand-off by pretending to have followed a pointer to rootHolder under version 0. get does not need a retry loop because the outer-most invocation cannot fail optimistic validation, as rootHolder.version is always 0. IgnoreGrow is used to ignore changes that grow the subtree, since they cannot affect the

correctness of the search. We discuss the `waitUntilNotChanging` method in Section 3.9.

Conceptually, the execution interval between reading the version number at Line 35 and the validation that occurs in the recursive invocation at Line 40 constitute a read-only transaction. Each level of the `attemptGet` recursion begins a new transaction and commits the transaction begun by its caller. Interestingly, we can only determine retroactively whether any particular validation at Line 40 was the final validation (and hence the commit) of the atomic region. If a recursive call returns `Retry` the enclosing loop will attempt another validation, which has the effect of extending the duration of the transaction begun by the caller. This 'resurrection' of a transaction that was previously considered committed is only possible because these atomic regions perform no writes.

3.4 Insertion and update: put(k, v)

The `put` operation may result in either an insertion or an update, depending on whether or not an association for the key already is present in the tree. It starts in exactly the same manner as `get`, because its first task is to identify the location in the tree where the change should occur. We do not extract this common functionality into a separate function, because during insertion we must perform the last check of the hand-over-hand optimistic validation after a lock on the parent has been acquired. Figure 6 shows the implementation of `put`.

If we discover that `node.key` matches the target key then we can be trivially certain that any concurrent tree rotations will not affect the ability of the search to find a matching node. This means that on Line 90 of the `attemptUpdate` helper function we do not need to examine `node.version` for evidence of shrinks, rather we only verify that the node has not been unlinked from the tree. The lock on a node must be acquired before it is unlinked, so for the duration of `attemptUpdate`'s critical region we can be certain that the node is still linked.

To safely insert a node into the tree we must acquire a lock on the future parent of the new leaf, and we must also guarantee that no other inserting thread may decide to perform an insertion of the same key into a different parent. It is not sufficient to merely check that the expected child link is still null after acquiring the parent lock. Consider the tree $b(a, d(\cdot, e))$, in which $p(l, r)$ indicates that l is the left child of p and r is the right child of p, $\cdot$ represents a null child link, and the keys have the same ordering as the name of their node. A put of c by a thread T_1 will conclude that the appropriate parent is d. If some other thread performs a left-rotation at b then the tree will be $d(b(a, \cdot), e)$, which may cause a second thread T_2 to conclude that the insertion should be performed under b. If T_2 proceeds, and then later performs a double rotation (left at b, right at d) the resulting tree will be $c(b(a, \cdot), d(\cdot, e))$. To guard against sequences such as this, Lines 79 and 80 of `attemptInsert` perform the same closing validation as in `get`. Any rotation that could change the parent into which k should be inserted will invalidate the implicit range of the traversal that arrived at the parent, and hence will be detected by the final validation.

3.5 Partially external trees: remove(k)

Up until now we have only considered operations that require locking a single node; removals are more difficult. In an internal tree with no routing nodes, deletion of a node n with two children requires that n's successor s be unlinked from n.`right` and linked into n's place in the tree. Locating s may require the traversal of as many as n.`height` $- 1$ links. In a concurrent tree the unlink and relink of s must be performed atomically, and any concurrent searches for s must be invalidated. Every node along the path from n to s's original location must be considered to have shrunk, and

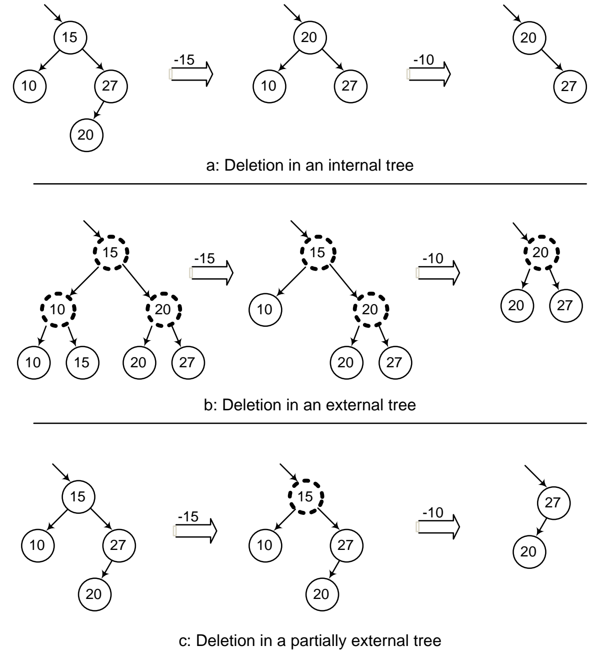

Figure 7. A sequence of two deletions in different types of trees.

```
 96 V remove(K k) {
 97   return (V) attemptRemove(k, rootHolder, 1, 0);
 98 }
 99 ... // attemptRemove is similar to attemptPut
100 boolean canUnlink(Node n) {
101   return n.left == null || n.right == null;
102 }
103 Object attemptRmNode(Node par, Node n) {
104   if (n.value == null) return null;
105   Object prev;
106   if (!canUnlink(n)) {
107     synchronized (n) {
108       if (n.version == Unlinked || canUnlink(n))
109         return Retry;
110       prev = n.value;
111       n.value = null;
112     }
113   } else {
114     synchronized (par) {
115       if (par.version == Unlinked || n.parent != par
116           || n.version == Unlinked)
117         return Retry;
118       synchronized (n) {
119         prev = n.value;
120         n.value = null;
121         if (canUnlink(n)) {
122           Node c = n.left == null ? n.right : n.left;
123           if (par.left == n)
124             par.left = c;
125           else
126             par.right = c;
127           if (c != null) c.parent = par;
128           n.version = Unlinked;
129     } } }
130     fixHeightAndRebalance(par);
131   }
132   return prev;
133 }
```

Figure 8. Removing `k`'s association, either by unlinking the node or by converting it to a routing node.

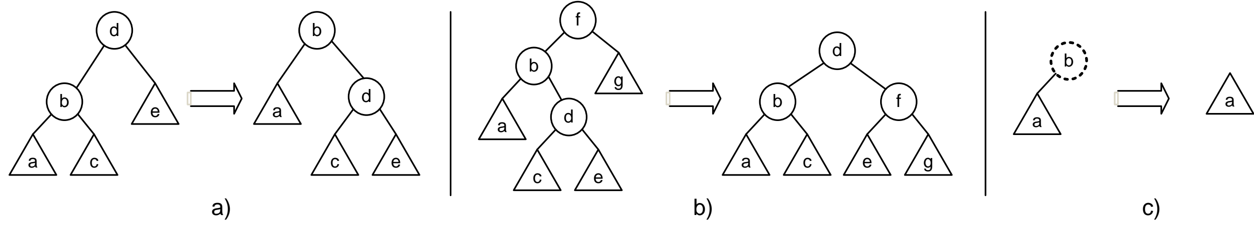

Figure 9. Local tree improvements: **a.** A right rotation of d. **b.** A right rotation of f over a left rotation of b. **c.** Unlink of the routing node b.

hence must be locked. This excessive locking negatively impacts both performance and scalability.

Previous research on concurrent relaxed balance trees handles this problem by using external trees [16] (or by prohibiting deletion entirely). In an external tree all key-value associations are held in leaf nodes, so there is never a deletion request that cannot be satisfied by a local operation. An external tree of size N requires $N - 1$ routing nodes, increasing the storage overhead and the average search path length. We would like to avoid these penalties while still taking advantage of an external tree's simple deletion semantics.

Our solution is to use what we refer to as *partially external trees*, in which routing nodes are only created during the removal of a node with two children. Routing nodes are never created during insertion, and routing nodes with fewer than two children are unlinked during rebalancing. Removal of a node with fewer than two children is handled by immediately unlinking it. In the worst case, partially external trees may have the same number of routing nodes as an external tree, but we observe that in practice, the number of routing nodes is much smaller (see Figure 14). To illustrate, Figure 7 shows a sequence of deletions in an internal tree, an external tree, and a partially external tree.

Our tree algorithm uses the same `Node` data type to represent both key-value associations and routing nodes. This allows a value node to be converted to a routing node by modifying a field in the node, no changes to inter-node links are required. Specifically, value nodes are those with non-null `values` and routing nodes are those that have a null `value` [2]. A routing node for k is converted back to a value node by a call to `put`(k, v).

Figure 8 gives some of the code for implementing `remove`. The process of removal follows the same pattern as `put`. Where Line 58 of `attemptPut` performs an insertion, `attemptRemove` merely returns `null`, and where Line 62 of `attemptPut` calls `attemptUpdate`, `attemptRemove` calls `attemptRmNode`.

`attemptRmNode` repeats our algorithm's motif of check, lock, recheck, then act. Combined with a retry loop in the caller, the motif implements optimistic concurrency control. `attemptRmNode`, however, is complicated enough to illustrate a way in which OCC tailored to an application can reduce optimistic failure rates.

Line 106 performs a check to see if the node may be unlinked or if it should be converted to a routing node. If unlinking is possible, locks are acquired on both the parent `p` and the node `n`, and then Line 121 verifies that unlinking is still possible. If unlinking is no longer possible, a generic OCC algorithm would have to roll back and retry, but this is not actually necessary. All of the locks required to convert `n` to a routing node are already held, so regardless of the outcome of this check the critical section may complete its work. In contrast, optimistic retry is required if the recheck of `canUnlink` on Line 108 shows that unlinking has become possible, because the locks held are only a subset of those required for unlinking.

[2] In the benchmarked implementation we support user-supplied null values by encoding and decoding them as they cross the tree's public interface.

3.6 Local tree improvements: `fixHeightAndRebalance`

The implementations of `get`, `put`, and `remove` require only a binary search tree, but to achieve good performance the tree must be approximately balanced. Our algorithm performs local improvements to the tree using the `fixHeightAndRebalance` method, which is called when a node is inserted or unlinked from the tree. This method recomputes the `height` field from the heights of the children, unlinks routing nodes with fewer than two children, and performs a single or double rotation to reduce imbalance. Our algorithm applies the same set of rotations as in a sequential AVL tree. Figure 9 shows a right rotation of d, a double rotation of f (we refer to this as a right-over-left rotation), and an unlink of b. The remaining possible local improvements are mirror images of these.

In a strict AVL tree the balance factor is never smaller than -2 or larger than $+2$, but in a relaxed balance AVL tree this is not the case. Multiple insertions or deletions may have accumulated before rebalancing, leading to a balance factor outside the normal range. We use the same rotation selection criteria as Bougé et al. [6]. If the apparent balance of a node n is n`.left.height` $-$ n`.right.height`, then we apply a right rotation if a node's apparent balance is $\geq +2$ and a left rotation if a node's apparent balance is ≤ -2. Prior to a right rotation of a node n, if the apparent balance of n`.left` is ≤ 0 a left rotation is first performed on n`.left`. A similar rule is applied prior to a left rotation of n.

In a concurrent relaxed balance tree there is an important distinction between the actual height and the value stored in `Node.height`. A node's `height` field records only the height as was apparent at a previous point in time, not the height that would be computed by a traversal of the tree in its current state. Bougé et al. establish the important theoretical result that, even if rebalancing is performed using only the apparent height, a sequence of localized improvements to the tree eventually results in a strict AVL tree [6]. The difficulty lies in efficiently identifying the set of nodes which can be improved.

Our algorithm guarantees that the tree will be a strict AVL tree whenever it is quiescent. Each mutating operation (insertion, removal, or rotation) is careful to guarantee that repairs will only be required for a node n or one of its ancestors, and that if no repair to n is required, no repair to any of n's ancestors is required. The node n is the one that is passed to `fixHeightAndRebalance`. The only mutation that may require repairs that don't fit this model is a double rotation in which the lower (first) rotation is not sufficient to restore balance. In this situation we do not attempt to merge the two rotations. We instead perform the lower rotation and then reapply the balance rules to the same node n.

Repair of a node requires that the children's heights be read, but performance and scalability would be heavily impacted if locks were required to perform these reads. Version numbers could be used to build a read-only atomic region, but this is not necessary. When considering if a node n should have its `height` field updated or should be rotated (both of which must be done under a lock), it is correct to perform the read of n`.left.height` and n`.right.height` without locks or optimistic retry loops. If no

```
134 // n.parent, n, and n.left are locked on entry
135 void rotateRight(Node n) {
136   Node nP = n.parent;
137   Node nL = n.left;
138   Node nLR = nL.right;
139
140   n.version |= Shrinking;
141   nL.version |= Growing;
142
143   n.left = nLR;
144   nL.right = n;
145   if (nP.left == n) nP.left = nL; else nP.right = nL;
146
147   nL.parent = nP;
148   n.parent = nL;
149   if (nLR != null) nLR.parent = n;
150
151   val h = 1 + Math.max(height(nLR), height(n.right));
152   n.height = h;
153   nL.height = 1 + Math.max(height(nL.left), h);
154
155   nL.version += GrowCountIncr;
156   n.version += ShrinkCountIncr;
157 }
```

Figure 10. Performing a right rotation. Link update order is important for interacting with concurrent searches.

other thread is concurrently modifying those fields, then the check is atomic without locks or version numbers. If one of the reads is incorrect, then the thread that is performing a concurrent change is responsible for repairing n. If another thread is responsible for the repair, then it is okay if the current `fixHeightAndRebalance` incorrectly decides that no repair of n is necessary.

We omit the code for `fixHeightAndRebalance` due to space constraints (a download link for complete code is given in Appendix A), but it uses the same concurrency control structure as `get` and `remove`. An outer loop performs unlocked reads to determine whether the height should be adjusted, a node should be unlinked, a rotation should be performed, or if the current node needs no repair. If a change to the node is indicated, the required locks are acquired, and then the appropriate action is recomputed. If the current locks are sufficient to perform the newly computed action, or if the missing locks can be acquired without violating the lock order, then the newly computed action is performed. Otherwise, the locks are released and the outer loop restarted. If no local changes to the tree are required then control is returned to the caller, otherwise the process is repeated on the parent.

The critical section of a right rotation is shown in Figure 10. This method requires that the parent, node, and left child be locked on entry. Java monitors are used for mutual exclusion between concurrent writers, while optimistic version numbers are used for concurrency control between readers and writers. This separation allows the critical region to acquire permission to perform the rotation separately from reporting to readers that a change is in progress. This means that readers are only obstructed from Line 140 to Line 156. This code performs no allocation, has no backward branches, and all function calls are easily inlined.

3.7 Link update order during rotation

The order in which links are updated is important. A concurrent search may observe the tree in any of the intermediate states, and must not fail to be invalidated if it performs a traversal that leads to a branch smaller than expected. If the update on Line 145 was performed before the updates on Lines 143 and 144, then a concurrent search for `n.key` that observed only the first link change could follow a path from `n.parent` to `n.left` to `n.left.right` (none of these nodes are marked as shrinking), incorrectly failing to find `n`. In general, downward links originally pointing to a shrinking node must be changed last and downward links from a shrinking node must be changed first. A similar logic can be applied to the ordering of parent updates.

```
158 static int SpinCount = 100;
159
160 void waitUntilNotChanging(Node n) {
161   long v = n.version;
162   if ((v & (Growing | Shrinking)) != 0) {
163     int i = 0;
164     while (n.version == v && i < SpinCount) ++i;
165     if (i == SpinCount) synchronized (n) { };
166   }
167 }
```

Figure 11. Code to wait for an obstruction to clear.

3.8 Iteration: `firstNode()` and `succNode(`n`)`

`firstNode`() and `succNode`(n) are the internal building blocks of an iterator interface. Because they return a reference to a `Node`, rather than a value, the caller is responsible for checking later that the node is still present in the tree. In an iterator this can be done by internally advancing until a non-null `value` is found.

`firstNode` returns the left-most node in the tree. It walks down the left spine using hand-over-hand optimistic validation, always choosing the left branch. Optimistic retry is only required if a node has shrunk.

`succNode` uses hand-over-hand optimistic validation to traverse the tree, but unlike searches that only move down the tree it must retry if either a shrink or grow is encountered. A complex implementation is possible that would tolerate grows while following parent links and shrinks while following child links, but it would have to perform key comparisons to determine the correct link to follow. We instead apply optimistic validation to the normal tree traversal algorithm, which is able to find the successor based entirely on the structure of the tree. If n is deleted during iteration then `succNode`(n) searches from the root using n`.key`.

3.9 Blocking readers: `waitUntilNotChanging`

Prior to changing a link that may invalidate a concurrent search or iteration, the writer sets either the `Growing` or `Shrinking` bit in the version number protecting the link, as described in Section 3.2. After the change is completed, a new version number is installed that does not have either of these bits set. During this interval a reader that wishes to traverse the link will be obstructed.

Our algorithm is careful to minimize the duration of the code that executes while the version has a value that can obstruct a reader. No system calls are made, no memory is allocated, and no backward branches are taken. This means that it is very likely that a small spin loop is sufficient for a reader to wait out the obstruction. Figure 11 shows the implementation of the waiter.

If the spin loop is not sufficient to wait out the obstruction, Line 165 acquires and then releases the changing node's monitor. The obstructing thread must hold the monitor to change `node.version`. Thus after the empty synchronized block has completed, the version number is guaranteed to have changed. The effect of a properly tuned spin loop is that readers will only fall back to the synchronization option if the obstructing thread has been suspended, which is precisely the situation in which the reader should block itself. Tolerance of high multi-threading levels requires that threads that are unable to make progress quickly block themselves using the JVM's builtin mechanisms, rather than wasting resources with fruitless retries.

3.10 Supporting fast `clone`

We extend our concurrent tree data structure to support `clone`, an operation that creates a new mutable concurrent tree containing the same key-value associations as the original. After a `clone`, changes made to either map do not affect the other. `clone` can be used to checkpoint the state of the map, or to provide snapshot isolation during iteration or bulk read.

We support fast cloning by sharing nodes between the original tree and the clone, lazily copying shared nodes prior to modifying them. This copy-on-write scheme requires that we be able to mark all nodes of a tree as shared without visiting them individually. This is accomplished by delaying the marking of a node until its parent is copied. All nodes in the tree may be safely shared once the root node has been explicitly marked and no mutating operations that might have not observed the root's mark are still active.

The `clone` method marks the root as shared, and then returns a new enclosing tree object with a new root holder pointing to the shared root. Nodes are explicitly marked as shared by setting their `parent` pointer to `null`. Clearing this link also prevents a Java reference chain from forming between unshared nodes under different root holders, which would prevent garbage collection of the entire original tree. Lazy copying is performed during the downward traversal of a `put` or `remove`, and during rebalancing. The first access to a child link in a mutating operation is replaced by a call to `unsharedLeft` or `unsharedRight` (see Figure 12). Both children are copied at once to minimize the number of times that the parent node must be locked.

Mutating operations that are already under way must be completed before the root can be marked, because they may perform updates without copying, and because they need to traverse parent pointers to rebalance the tree. To track pending operations, we separate updates into *epochs*. `clone` marks the current epoch as closed, after which new mutating operations must await the installation of a new epoch. Once all updates in the current epoch have completed, the root is marked shared and updates may resume. We implement epochs as objects that contain a count of the number of pending mutating operations, a flag that indicates when an epoch has been closed, and a condition variable used to wake up threads blocked pending the completion of a close. The count is striped across multiple cache lines to avoid contention. Each snap-tree instance has its own epoch instance.

```
168  Node unsharedLeft(Node p) {
169    Node n = p.left;
170    if (n.parent != null)
171      return n;
172    lazyCopyChildren(n);
173    return p.left;
174  }
175  Node unsharedRight(Node p) { ... }
176
177  void lazyCopyChildren(Node n) {
178    synchronized (n) {
179      Node cl = n.left;
180      if (cl != null && cl.parent == null)
181        n.left = lazyCopy(cl, n);
182      Node cr = n.right;
183      if (cr != null && cr.parent == null)
184        n.right = lazyCopy(cr, n);
185    }
186  }
187  Node lazyCopy(Node c, Node newPar) {
188    return new Node(c.key, c.height, c.value, newPar,
189        0L, markShared(c.left), markShared(c.right));
190  }
191  Node markShared(Node node) {
192    if (node != null) node.parent = null;
193    return node;
194  }
```

Figure 12. Code for lazily marking nodes as shared and performing lazy copy-on-write. Nodes are marked as shared while copying the parent.

4. Correctness

Deadlock freedom: Our algorithm uses the tree to define allowed lock orders. A thread that holds no locks may request a lock on any node, and a thread that has already acquired one or more locks may only request a lock on one of the children of the node most recently locked. Each critical region preserves the binary search tree property, and each critical region only changes child and parent links after acquiring all of the required locks. A change of p`.left` or p`.right` to point to n requires a lock on both p, n, and the old parent of n, if any exists. This means that it is not possible for two threads T_1 and T_2 to hold locks on nodes p_1 and p_2, respectively, and for T_1 to observe that n is a child of p_1 while T_2 observes that n is a child of p_2.

This protocol is deadlock free despite concurrent changes to the tree structure. Consider threads T_i that hold at least one lock (the only threads that may participate in a deadlock cycle). Let a_i be the lock held by T_i least recently acquired, and let z_i be the lock held by T_i most recently acquired. The node z_i is equal to a_i or is a descendent of a_i, because locks are acquired only on children of the previous z_i and each child traversal is protected by a lock held by T_i. If T_i is blocked by a lock held by T_j, the unavailable lock must be a_j. If the unavailable lock were not the first acquired by T_j then both T_i and T_j would agree on the parent and hold the parent lock, which is not possible. This means that if a deadlock cycle were possible it must consist of two or more threads $T_1 \cdots T_n$ where z_i is the parent of $a_{(i \mod n)+1}$. Because no such loop exists in the tree structure, and all parent-child relationships in the loop are protected by the lock required to make them consistent, no deadlock cycle can exist.

Linearizability: To demonstrate linearizability [13] we will define the linearization point for each operation and then show that operations for a particular key produce results consistent with sequential operations on an abstract map structure.

We define the linearization point for `put`(k, v) to be the last execution of Line 82 or 92 prior to the operation's completion. This corresponds to a successful `attemptInsert` or `attemptUpdate`. We define the linearization point for `get`(k) to be the last execution of Line 27 if Line 39 is executed, Line 34 if that line is executed and `child.value` $\neq$ `null` or `child.version` $\neq$ `Unlinked`, or otherwise Line 128 during the successful `attemptRmNode(_,child)` that removed `child`. If `remove`(k) results in the execution of either Line 111 or 120 we define that to be the linearization point. We omit the details for the linearization point of a `remove` operation that does not modify the map, but it is defined analogously to `get`'s linearization point.

Atomicity and ordering is trivially provided between `put`s that linearize at Line 92 (updates) and removals that change the tree (both the introduction of routing nodes and unlinking of nodes) by their acquisition of a lock on the node and their check that `node.version` $\neq$ `Unlinked`. Nodes are marked unlinked while locked, so it is not possible for separate threads to simultaneously lock nodes n_1 and n_2 for k, and observe that neither is unlinked. This means that any operations that operate on a locked node for k must be operating on the same node instance, so they are serialized. The only mutating operation that does not hold a lock on the node for k while linearizing is insertion, which instead holds a lock on

the parent. The final hand-over-hand optimistic validation during insert (Line 80) occurs after a lock on the parent has been acquired. The validation guarantees that if a node for k is present in the map it must be in the branch rooted at `node.child(dir)`, which is observed to be empty. This means that no concurrent update or remove operation can observe a node for k to exist, and that no concurrent insert can disagree about the parent node into which the child should be inserted. Since concurrent inserts agree on the parent node, their lock on the parent serializes them (and causes the second insert to discover the node on Line 80, triggering retry).

Linearization for `get` is a bit trickier, because in some cases we perform the last validation (Line 28) prior to reading `child.value` (Line 34); during this interval `child` may be unlinked from the tree. If a node for k is present in the tree at Line 27 then `get` will not fail to find it, because the validation at Line 28 guarantees that the binary search invariant held while reading `node.child(dir)`. This means that when returning at Line 39 we may correctly linearize at Line 27. If a `child` is discovered and has not been unlinked prior to the read of its `value`, then the volatile read of this field is a correct linearization point with any concurrent mutating operations. If `child` $\neq$ `null` but it has been unlinked prior to Line 34 then we will definitely observe a `value` of `null`. In that case `attemptRmNode` had not cleared the child link of `node` (Line 124 or 126) when we read `child`, but it has since set `child.version` to `Unlinked` (Line 128). We therefore declare that `get` linearizes at this moment when k was definitely absent from the map, before a potential concurrent insert of k.

5. Performance

In this section we evaluate the performance of our algorithm. We compare its performance to Doug Lea's lock-free `ConcurrentSkipListMap`, the fastest concurrent ordered map implementation for Java VMs of which the authors are aware. We also evaluate our performance relative to two red-black tree implementations, one of which uses a single lock to guard all accesses and one of which is made concurrent by an STM.

The benchmarked implementation of our algorithm is written in Java. For clarity this paper describes `put`, `remove`, and `fixHeightOrRebalance` as separate methods, but the complete code does not have this clean separation. The `ConcurrentMap` operations of `put`, `putIfAbsent`, `replace`, and `remove` are all implemented using the same routine, with a case statement to determine behavior once the matching node has been found. In addition, an attempt is made to opportunistically fix the parent's `height` during insertion or removal while the parent lock is still held, which reduces the number of times that `fixHeightOrRebalance` must reacquire a lock that was just released. The benchmarked code also uses a distinguished object to stand in for a user-supplied null value, encoding and decoding at the boundary of the tree's interface.

Experiments were run on a Dell Precision T7500n with two quad-core 2.66Ghz Intel Xeon X5550 processors, and 24GB of RAM. Hyper-Threading was enabled, yielding a total of 16 hardware thread contexts. We ran our experiments in Sun's Java SE Runtime Environment, build 1.6.0_16-b01, using the HotSpot 64-Bit Server VM with default options. The operating system was Ubuntu 9.0.4 Server, with the x86_64 Linux kernel version `2.6.28-11-server`.

Our experiments emulate the methodology used by Herlihy et al. [11]. Each pass of the test program consists of each thread performing one million randomly chosen operations on a shared concurrent map; a new map is used for each pass. To simulate a variety of workload environments, two main parameters are varied: the proportion of `put`, `remove`, and `get` operations, and the range from which the keys are selected (the "key range"). Increasing the number of mutating operations increases contention; experiments

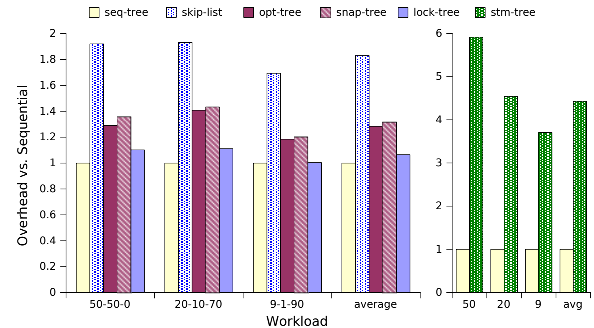

Figure 13. Single thread overheads imposed by support for concurrent access. Workload labels are ⟨put%⟩-⟨remove%⟩-⟨get%⟩. A key range of 2×10^5 was used for all experiments.

with 90% `get` operations have low contention, while those with 0% `get` operations have high contention. The key range affects both the size of the tree and the amount of contention. A larger key range results in a bigger tree, which reduces contention.

To ensure consistent and accurate results, each experiment consists of eight passes; the first four warm up the VM and the second four are timed. Throughput results are reported as operations per millisecond. Each experiment was run five times and the arithmetic average is reported as the final result.

We compare five implementations of thread-safe ordered maps:

- **skip-list** - Doug Lea's `ConcurrentSkipListMap` This skip list is based on the work of Fraser and Harris [9]. It was first included in version 1.6 of the Java™ standard library.
- **opt-tree** - our optimistic tree algorithm.
- **snap-tree** - the extension of our algorithm that provides support for fast cloning and snapshots.
- **lock-tree** - a standard `java.util.TreeMap` wrapped by `Collections.synchronizedSortedMap()`. Iteration is protected by an explicit lock on the map.
- **stm-tree** - a red-black tree implemented in Scala[3] using CCSTM [7]. STM read and write barriers were minimized manually via common subexpression elimination. To minimize contention no size or modification count are maintained.

We first examine the single-threaded impacts of supporting concurrent execution. This is important for a data structure suitable for a wide range of uses, and it places a lower bound on the amount of parallelism required before scaling can lead to an overall performance improvement. We compare the sequential throughput of the five maps to that of an unsynchronized `java.util.TreeMap`, labeled "seq-tree". Values are calculated by dividing the throughput of seq-tree by that of the concurrent map. Figure 13 shows that on average our algorithm adds an overhead of 28%, significantly lower than the 83% overhead of skip-list, but more than the 6% imposed by an uncontended lock. The STM's performance penalty averages 443%. As expected, snap-tree is slower than opt-tree, but the difference is less than 3% for this single-threaded configuration.

Our second experiment evaluates the number of nodes present in a partially external tree compared to a fully external tree. Internal trees are the baseline, as they contain no routing nodes. To simulate a range of workloads we perform a million `put` or `remove` operations, varying the fraction of `puts` from 0% to 100%. In this

[3] The Scala compiler emits Java bytecodes directly, which are then run on the Java VM. Scala code that does not use closures has performance almost identical to the more verbose Java equivalent.

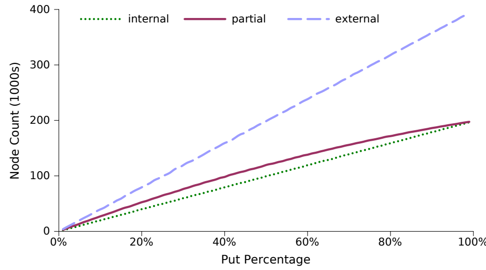

Figure 14. Node count as tree size increases. One million operations were performed with a varying ratio of `put` and `remove` operations, and a key range of 2×10^5; the number of nodes in the resulting tree is shown.

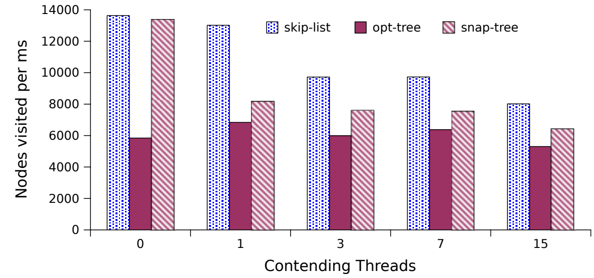

Figure 16. Iteration throughput of a single thread, with contending threads performing the 20-10-70 workload over 2×10^5 keys. Skip-list and opt-tree perform inconsistent iteration, snap-tree performs an iteration with snapshot isolation.

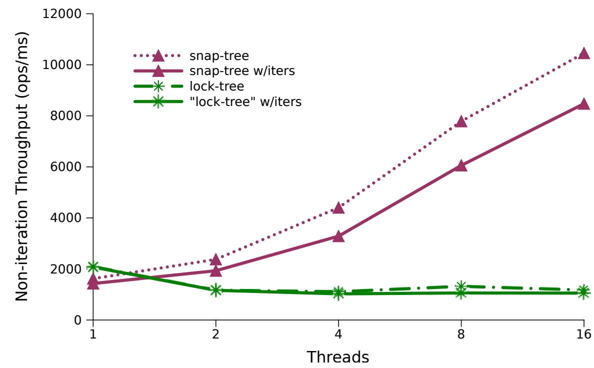

Figure 17. Throughput performing the 20-10-70 workload over 2×10^5 keys with concurrent consistent iterations.

experiment we use a key range of 2×10^5. The results, presented in Figure 14, show that partially external trees require far fewer routing nodes (on average 80% fewer) than external trees. A key range of 2×10^6 with 10 million operations yields a similar curve.

Figure 15 shows how throughput scales as the number of threads is swept from 1 to 64, for a range of operation mixes and various levels of contention. Moving left to right in the figure, there are fewer mutating operations and thus lower contention. Moving bottom to top, the range of keys get larger, resulting in bigger trees and lower contention. Thus the lower left graph is the workload with the highest contention and the upper right is the workload with the lowest contention.

As expected, the throughput of each map, with the exception of lock-tree, generally increases as more of the system's 16 hardware thread contexts are utilized. At multiprogramming levels of 2 and 4 (32 and 64 threads) throughput flattens out. Higher numbers of threads increase the chances that a single `fixHeightAndRebalance` call can clean up for multiple mutating operations, reducing the amortized cost of rebalancing and allowing scaling to continue past the number of available hardware threads in some cases. As the key range gets smaller the absolute throughput increases, despite the higher contention, showing that both Lea's and our algorithms are tolerant of high contention scenarios. The absolute throughput also increases as the number of mutating operations decreases (going from left to right), as would be expected if reads are faster than writes. The course-grained locking of lock-tree imposes a performance penalty under any amount of contention, preventing it from scaling in any scenario. Stm-tree exhibits good scaling, especially for read-dominated configurations, but its poor single-threaded performance prevents it from being competitive with skip-list or either of our tree algorithms.

With a large key range of 2×10^6, our algorithm outperforms skip-list by up to 98%, with an average increase in throughput of 62% without fast clone and snapshot support, and 55% with such support. Both of our tree implementations continue to exhibit higher throughput with a key range of 2×10^5, but as the key range decreases and contention rises, the advantage becomes less pronounced. Opt-tree performs on par with skip-list for a key range of 2×10^4, but fails to maintain its performance advantage in multiprogramming workloads with a key range of 2×10^3, the only workload in which skip-list has noticeably higher performance. In the worst case for opt-tree (64 threads, 20-10-70 workload, and a key range of 2×10^3) it was 13% slower than skip-list. In the worst case for snap-tree (64 threads, 50-50-0 workload, and a key range of 2×10^3) it was 32% slower than skip-list. Averaged over all workloads and thread counts, opt-tree was 32% faster than skip-list and snap-tree was 24% faster than skip-list.

The primary difference between opt-tree and snap-tree is snap-tree's epoch tracking. This imposes a constant amount of extra work on each put or remove. As expected, the overhead of supporting snapshots decreases when moving right in the table, to configurations with fewer mutating operations. The relative cost of epoch tracking is also reduced as the tree size increases, because more work is done per operation. Across the board, snap-tree imposes a 9% overhead when compared to opt-tree, with a worst-case penalty of 31%. Snap-tree's overhead for read operations is negligible.

We next examine the performance of iterating sequentially through the map while concurrent mutating operations are being performed. Our standard per-thread workload of 20% `puts`, 10% `removes`, and 70% `gets`, and a key range of 2×10^5 is interleaved at regular intervals with a complete iteration of the map. On average only one thread is iterating at a time. We calculate throughput as the total number of nodes visited, divided by the portion of total running time spent in iteration; the results are presented in Figure 16. Our experimental setup did not allow us to accurately measure the execution breakdown for multiprogramming levels greater than one, so we only show results up to 16 threads. At its core, `ConcurrentSkipListMap` contains a singly-linked list, so we expect it to support very fast iteration. Iteration in opt-tree is much more complex; nevertheless, its average performance is 48% that of skip-list. No optimistic hand-over-hand optimistic validation is required to iterate the snapshot in a snap-tree, so its performance is intermediate between skip-list and opt-tree, even though it is providing snapshot consistency to the iterators.

Snap-tree provides snapshot isolation during iteration by traversing a clone of the original tree. This means that once the epoch transition triggered by `clone` has completed, `puts` and `removes` may operate concurrently with the iteration. To evaluate the performance impact of the lazy copies requires by subsequent writes, Figure 17 plots the throughput of non-iteration operations during

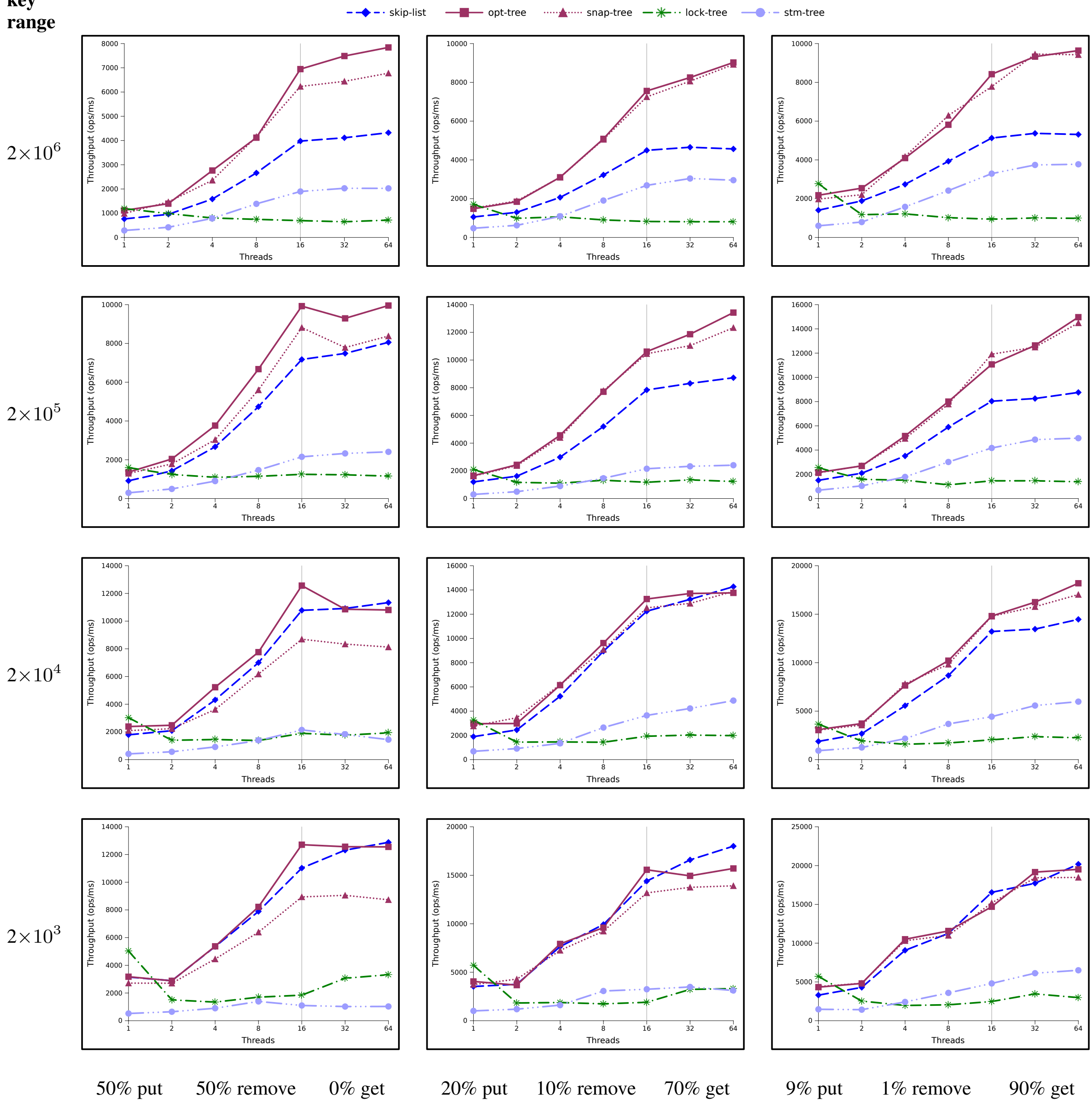

Figure 15. Each graph shows the throughput of the maps as thread count ranges from 1 to 64. “skip-list” is `ConcurrentSkipListMap`, “opt-tree” is our basic optimistic tree algorithm, “snap-tree” is the extension of our algorithm that supports fast snapshots and cloning, “lock-tree” is a synchronized `java.util.TreeMap`, and “stm-tree” is a red-black tree made concurrent with an STM. Moving left to right, the operation mix has fewer mutating operations and thus lower contention. Moving bottom to top, the range of keys get larger, resulting in bigger trees and lower contention. Thus the lower left graph is the workload with the highest contention and the upper right is the workload with the lowest contention. 16 hardware threads were available.

the same workload as Figure 16, along with the throughput with no concurrent iterations. Only snap-tree and lock-tree are shown, because they are the only implementations that allow consistent iteration. On average, concurrent iterations lower the throughput of other operations by 19% in snap-tree.

6. Conclusion

In this paper we use optimistic concurrency techniques adapted from software transactional memory to develop a concurrent tree data structure. By carefully controlling the size of critical regions and taking advantage of algorithm-specific validation logic, our tree delivers high performance and good scalability while being tolerant of contention. We also explore a variation of the design that adds support for a fast clone operation and that provides snapshot isolation during iteration.

We compare our optimistic tree against a highly tuned concurrent skip list, the best performing concurrent ordered map of which we are aware. Experiments shows that our algorithm outperforms the skip list for many access patterns, with an average of 39% higher single-threaded throughput and 32% higher multi-threaded throughput. We also demonstrate that a linearizable clone operation can be provided with low overhead.

A. Code

Java implementations of opt-tree and snap-tree are available from `http://github.com/nbronson/snaptree`.

Acknowledgments

This work was supported by the Stanford Pervasive Parallelism Lab, by Dept. of the Army, AHPCRC W911NF-07-2-0027-1, and by the National Science Foundation under grant CNS–0720905.

References

[1] G. Adel'son-Vel'skiĭ and E. M. Landis. An algorithm for the organization of information. In *Proceedings of the USSR Academy of Sciences*, volume 145, pages 263–266, 1962. In Russian, English translation by Myron J. Ricci in Soviet Doklady, 3:1259-1263, 1962.

[2] M. Ansari, C. Kotselidis, M. Lujan, C. Kirkham, and I. Watson. On the performance of contention managers for complex transactional memory benchmarks. In *ISPDC '09: Proceedings of the 8th International Symposium on Parallel and Distributed Computing*, pages 83–90, 2009.

[3] L. Ballard. Conflict avoidance: Data structures in transactional memory. Brown University Undergraduate Thesis, 2006.

[4] R. Bayer. Symmetric binary B–Trees: Data structure and maintenance algorithms. *Acta Inf.*, 1:290–306, 1972.

[5] R. Bayer and M. Schkolnick. Concurrency of operations on B–Trees. In *Readings in Database Systems (2nd ed.)*, pages 216–226, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[6] L. Bougé, J. Gabarró, X. Messeguer, and N. Schabanel. Height-relaxed AVL rebalancing: A unified, fine-grained approach to concurrent dictionaries. ResearchReport RR1998-18, LIP, ENS Lyon, March 1998.

[7] N. Bronson. CCSTM. http://github.com/nbronson/ccstm.

[8] P. Felber, V. Gramoli, and R. Guerraoui. Elastic transactions. In *DISC '09: Proceedings of the 23rd International Symposium on Distributed Computing*, pages 93–107, 2009.

[9] K. Fraser. *Practical Lock Freedom*. PhD thesis, University of Cambridge, 2003.

[10] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science*, pages 8–21, Oct. 1978.

[11] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit. A provably correct scalable concurrent skip list. In *OPODIS '06: Proceedings of the 10th International Conference On Principles Of Distributed Systems*, 2006.

[12] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101, New York, NY, USA, 2003. ACM.

[13] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[14] K. S. Larsen. AVL trees with relaxed balance. In *IPPT '94: Proceedings of the 8th International Symposium on Parallel Processing*, pages 888–893, Washington, DC, USA, 1994. IEEE Computer Society.

[15] J. M. Mellor-Crummey and M. L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. In *PPOPP '91: Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 106–113, New York, NY, USA, 1991. ACM.

[16] O. Nurmi, E. Soisalon-Soininen, and D. Wood. Concurrency control in database structures with relaxed balance. In *PODS '87: Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 170–176, New York, NY, USA, 1987. ACM.

[17] B. Pfaff. Performance analysis of BSTs in system software. *SIGMETRICS Performance Evaluation Review*, 32(1):410–411, 2004.

[18] W. Pugh. Concurrent maintenance of skip lists. Technical Report CS-TR-2222.1, 1989.

[19] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *ACM Transactions on Database Systems*, 33(6):668–676, 1990.

[20] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. Cao Minh, and B. Hertzberg. McRT–STM: A high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 187–197, New York, NY, USA, March 2006. ACM Press.

[21] H. Sundell and P. Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. *Journal of Parallel Distributed Computing*, 65(5):609–627, 2005.