A Contention Adapting Approach to Concurrent Ordered Sets[☆]

Konstantinos Sagonas[a,b], Kjell Winblad[a,*]

[a]*Department of Information Technology, Uppsala University, Sweden*
[b]*School of Electrical and Computer Engineering, National Technical University of Athens, Greece*

Abstract

With multicores being ubiquitous, concurrent data structures are increasingly important. This article proposes a novel approach to concurrent data structure design where the data structure dynamically adapts its synchronization granularity based on the detected contention and the amount of data that operations are accessing. This approach not only has the potential to reduce overheads associated with synchronization in uncontended scenarios, but can also be beneficial when the amount of data that operations are accessing atomically is unknown.

Using this adaptive approach we create a contention adapting search tree (CA tree) that can be used to implement concurrent ordered sets and maps with support for range queries and bulk operations. We provide detailed proof sketches for the linearizability as well as deadlock and livelock freedom of CA tree operations. We experimentally compare CA trees to state-of-the-art concurrent data structures and show that CA trees beat the best of the data structures that we compare against by over 50% in scenarios that contain basic set operations and range queries, outperform them by more than 1200% in scenarios that also contain range updates, and offer performance and scalability that is better than many of them on workloads that only contain basic set operations.

Keywords: concurrent data structures, ordered sets, linearizability, range queries

---

1. Introduction

With multicores being widespread, the need for efficient concurrent data structures has increased. This need has lead to an intensification of research in this area. For example, a large number of concurrent data structures for ordered sets have recently been proposed. To enable parallel operations in the data structure, some of them use fine-grained locking [1, 2, 3, 4] while others use lock-free techniques [5, 6, 7, 8, 9, 10, 11, 12].

[☆]Research supported in part by the European Union grant IST-2011-287510 “RELEASE: A High-Level Paradigm for Reliable Large-scale Server Software” and the Linnaeus centre of excellence UPMARC (Uppsala Programming for Multicore Architectures Research Center).

[*]Corresponding author
Email addresses: kostis@it.uu.se (Konstantinos Sagonas), kjell.winblad@it.uu.se (Kjell Winblad)

This article presents a family of concurrent ordered sets called *contention adapting search trees* (*CA trees*). In contrast to the data structures mentioned above, which use a fixed synchronization granularity, CA trees adapt their synchronization granularity at run time to fit the contention level. CA trees do this automatically by locally increasing the synchronization granularity where contention is estimated to be high and by decreasing the synchronization granularity in places where low contention is detected.

Even though many of the data structures for concurrent ordered sets that use a fixed synchronization granularity perform well when they are accessed by many threads in parallel, they all pay a price in memory overhead and performance for fine-grained synchronization when it is unnecessary. CA trees require very little extra memory for synchronization and have low performance overhead in low contention scenarios. Furthermore, by adapting their synchronization granularity, CA trees are able to provide good performance also in highly contended scenarios. Thus, with CA trees, programmers can get the benefits of both fine-grained synchronization and course-grained synchronization automatically, i.e., without applications needing to know or guess the level of contention in advance, something which is very difficult to do in code bases of significant size or when applications have several different execution phases.

Current research on concurrent ordered sets has mainly been focused on *single-key operations*, e.g., insert, remove and get (that retrieves a value associated with a key if it is present). Unfortunately, most of the recently proposed data structures lack efficient and scalable support for *multi-key operations* that atomically access multiple elements, such as range queries, range updates, bulk insert and remove. Multi-key operations are important for applications such as in-memory databases. Operations that operate on a single key and those that operate on multiple keys have inherently conflicting requirements. The former achieve good scalability by using fine-grained synchronization, while the latter are better off performance-wise if they employ more coarse-grained synchronization because of less synchronization overhead. The few data structures with scalable and efficient support for some multi-key operations [13, 14] have to be parameterized with the granularity of synchronization. Setting this parameter is inherently difficult since, as we just described, the usage patterns and contention level of applications are sometimes impossible to predict. This is especially true when the data structure is used to implement a general purpose key-value store. CA trees provide efficient support for multi-key operations and, in contrast to prior work on concurrent ordered sets, CA trees do not need to be parameterized with the synchronization granularity. Instead, heuristics are used to adapt the CA trees to a synchronization granularity that not only fits the contention level at hand, but also the type of operations that are used on them.

As we show in this article, CA trees provide good scalability and performance both in contended and uncontended situations. Moreover they are flexible: CA tree variants with versatile performance characteristics can be derived by selecting their underlying sequential data structure component. Experiments on scenarios with a variety of mixes of operations show that CA trees achieve performance that is significantly better than that obtained by state-of-the-art data structures for ordered sets with range query support (Section 8). All these make CA trees suitable for a multitude of applications, including in-memory databases, key-value stores and general purpose data structure libraries.

This article combines and extends two conference publications on CA trees that have described their support for single-key operations [15] and multi-key operations [16]. Besides presenting a uniform and more comprehensive description of the algorithms (Section 3) and performance (Section 8) of CA trees, compared to the conference publications this article also contains the following new material:

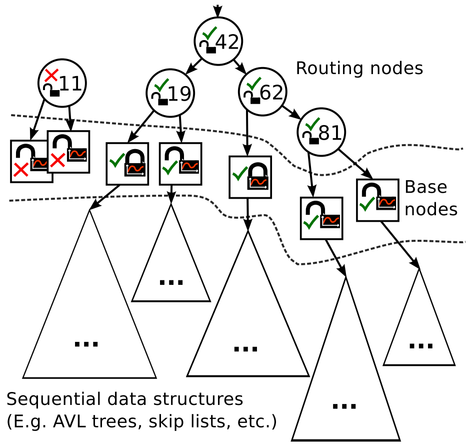
- Detailed arguments and proof sketches about the key properties (linearizability, deadlock and livelock freedom) of CA tree operations (Section 4.1).
- A description on how to make CA trees starvation free (Section 4.2).
- A more extensive discussion of the time complexity of operations (Section 4.3).
- A more up-to-date comparison with related work (Section 7), including experimental comparison with the LogAVL [4] data structure.
- Experimental results for the sequential performance of CA trees (Section 8).

Overview. We start by describing useful terminology and by giving a bird's eye view of CA trees (Section 2) before we describe the CA tree algorithms in detail (Section 3). We then state and prove the correctness properties that CA trees provide and discuss the complexity of CA tree operations (Section 4). Two important optional CA tree components are then described (Section 5) followed by the description of some optimizations (Section 6). Finally, we compare with related work (Section 7), experimentally compare CA trees to related data structures (Section 8), and end with some concluding remarks.

2. A Brief Overview of CA Trees

Let us first introduce some useful terminology. An *ordered set* is a data structure that represents a set of keys (and possible associated values) so that its keys are ordered according to some user defined order function. We use the term *single-key operation* to refer to operations that operate on a single key and/or associated value. Examples of common single-key operations for ordered sets are *insert* (that inserts a new key and associated value to its appropriate position in the set), *remove* (that removes an existing key) and *get* (that returns a value associated with an existing key). We call operations that operate on a range of elements *range operations* and use *multi-key operations* as a general term for operations that atomically access multiple elements. A *range query* operation atomically takes a snapshot of all keys that are in an ordered set and are within a certain range $[a, b]$ of keys. A *range update* atomically applies an update function to all values associated with keys in a specific key range. A *bulk insert* atomically inserts all elements in a list of keys or key-value pairs. (A *bulk remove* is defined similarly.)

As can be seen in Fig. 1, CA trees consist of three layers: one containing routing nodes, one containing base nodes and one containing sequential ordered set data structures. Essentially, the CA tree is an external binary search tree where the *routing nodes* are internal nodes whose sole purpose is to direct the search and the *base nodes* are the external nodes where the actual items are stored. All keys stored under the left pointer of a routing node are smaller than the routing node's key and all keys stored under the right pointer are greater than or equal to the routing node's key. A routing node also has a lock and a valid flag but these are only used rarely when a routing node is deleted to



42
11
19
62
81
Routing nodes
Base nodes
...
Sequential data structures
(E.g. AVL trees, skip lists, etc.)

Figure 1: The structure of a CA tree. Numbers denote keys, a node whose flag is valid is marked with a green hook; an invalid one with a red cross.

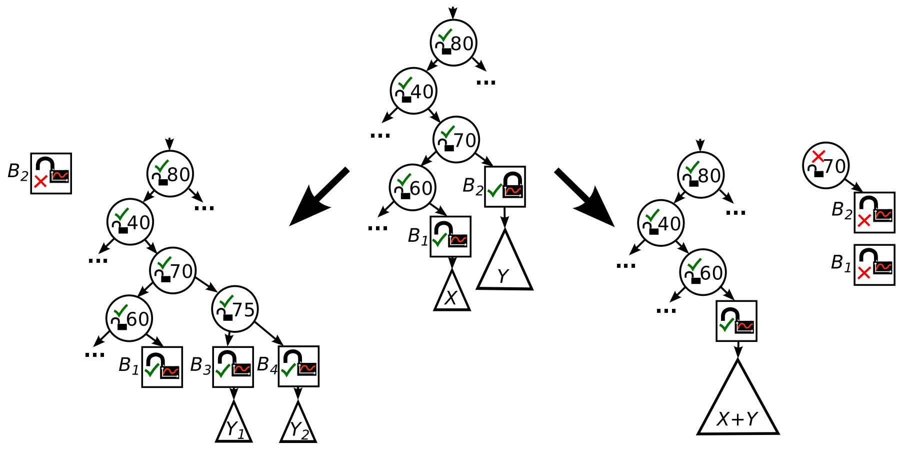
adapt to low contention. The nodes with the invalidated valid flags to the left of the tree in Fig. 1 are the result of the deletion of the routing node with key 11; nodes marked as invalid are no longer part of the tree.

A base node contains a statistics collecting (SC) lock, a valid flag and a sequential ordered set data structure. When a search in the CA tree ends up in a base node, the SC lock of that base node is acquired. This lock changes its statistics value during lock acquisition depending on whether the thread had to wait to get hold of the lock or not. The thread performing the search has to check the valid flag of the base node (retrying the operation if it is invalid) before it continues to search the sequential data structure inside the base node. The statistics counter in the SC lock is checked after an operation has been performed in the sequential data structure and before the lock is unlocked. When the statistics collected by the SC lock indicate that the contention is higher than a certain threshold in a base node B_2, then the sequential data structure in B_2 is split into two new base nodes that are linked together by a new routing node that replaces B_2 (see Figs. 2a and 2b). In the other direction, if the statistics counter in some base node B_2 indicates that the contention is lower than a threshold, then B_2 is joined with a neighbor base node B_1 by creating a new base node B_3 containing the keys from both B_1 and B_2 to replace B_1 and by splicing out the parent routing node of B_2 (see Figs. 2b and 2c).

3. Implementation

This section gives a detailed description of the CA tree operations with pseudocode[1]. We will first describe the implementation of the two components: SC locks and sequential ordered set data structures. We will then describe how to use these components

[1]The pseudocode that is referred to in the following sections is extracted from an executable Java implementation. The underlying Java code has been thoroughly checked with the Java Pathfinder [17] state space exploration framework. Both the Java code and the test code can be found online [18].



(a) After a split (b) Initial (c) After a join

Figure 2: Effect of the split and join operations on the initial CA tree, shown in the middle subfigure.

```
1 void statLock(StatLock slock) {
2   if (slock.mutex.tryLock()) {
3     slock.statistics -= SUCC_CONTRIB;
4     return;
5   }
6   slock.mutex.lock();
7   slock.statistics += FAIL_CONTRIB;
8 }
```

Figure 3: Pseudocde for statistics collecting lock.

to implement a CA tree supporting set operations that involve a single key as well as multi-key operations such as bulk insert/remove and range queries. Finally, we describe the implementation of an optimization for read-only operations.

3.1. Statistics Collecting Locks

We use a standard mutual exclusion (mutex) lock and an integer variable to create a statistics collecting lock. Pseudocode for such locks is shown in Fig. 3. The statistics variable is incremented or decremented after the lock has been taken. If the `tryLock` call on Line 2 succeeds, no contention was detected and the statistics variable is decremented with `SUCC_CONTRIB`. On the other hand, if the `tryLock` call fails, another thread is holding the lock so the statistics is incremented by `FAIL_CONTRIB` after the mutex lock has been acquired.

Two constants, `MIN_CONTENTION` and `MAX_CONTENTION`, are used to decide when to perform adaptations. If the statistics variable is greater than `MAX_CONTENTION`, the data structure adapts by splitting a base node because the contention is high. Symmetrically, the data structure adapts to low contention by joining base nodes when the statistics variable is less than `MIN_CONTENTION`. Intuitively one would like to adapt to high contention fast so that the available parallelism can be exploited. At least for CA trees, it is not as critical to adapt to low contention. The cost for using a CA tree adapted for slightly more contention than necessary is low. Therefore, the threshold for adapting to low contention can be higher than the threshold for adapting to high contention. This also has the benefit of avoiding too frequent splitting and joining of nodes. For CA trees we have found the values `MAX_CONTENTION` $= 1000$, `MIN_CONTENTION` $= -1000$, `SUCC_CONTRIB` $= 1$ and `FAIL_CONTRIB` $= 250$ to work well. These constants mean that it requires more than 250 uncontended lock calls for every contented lock call for the statistics to eventually indicate that low-contention adaptation needs to happen. Furthermore, it only requires a few contended lock calls in sequence for the statistics to indicate that high-contention adaptation should take place.

The overhead of maintaining statistics can be made very low. If one places the statistics counter on the same cache line as the lock data structure, it will be loaded into the core's private cache (in exclusive state) after the lock has been acquired and thus the counter can be updated very efficiently.

3.2. Ordered Sets with Split and Join Support

The sequential data structure component of a CA tree is used to store the keys that are in the set represented by the CA tree. As can be seen in Fig. 1 the sequential data structures are rooted in the base nodes. We will see that it is desirable that these data structures have efficient support for the operations supported by the CA tree. For efficient high and low contention adaptation we also need efficient support for the split and join operations.

The split operation splits an ordered set so that the maximum key in one of the resulting sets is smaller than the minimum key in the other. This operation can be implemented in many binary search trees by splicing out the root node of the tree and inserting the old root into one of its subtrees. Thus, split is as efficient as the tree's insert operation. The input of the join operation is two instances of the data structure where the minimum key in one of them is greater than the maximum key in the other. The resulting ordered set contains the union of the keys of the two input data structures.

AVL trees [19] and Red–Black trees [20] are balanced search trees that support both split and join operations in guaranteed $\mathcal{O}(log(N))$ time, where N is the total number of keys stored in the tree(s). A description of the join operation for AVL trees can be found in e.g., Knuth's book [21, page 474] and the corresponding description for Red–Black trees can be found in e.g., Tarjan's book [22, page 52]. It is also trivial to implement expected $\mathcal{O}(log(N))$ split and join operations in randomized ordered set data structures such as skip lists [23] and randomized search trees [23].

3.3. Single-key Operations

Figure 4 shows the CA tree algorithm for single-key operations. Since the algorithm is generic it can be used for all common set operations (e.g. insert, remove, get, etc.). The parameter named `operation` is the sequential data structure operation that shall be applied to the CA tree. The algorithm performs the following steps:(i) Lines 12 to 16 search the routing layer from the root of the tree until the search ends up in a

```
 9 Object doOperation(CATree tree, Op operation, K key, Object v) {
10   RouteNode prevNode = null;
11   Object currNode = tree.root;
12   while (currNode instanceof RouteNode) {
13     prevNode = currNode asInstanceOf RouteNode;
14     if (key < currNode.key) currNode = currNode.left;
15     else currNode = ((RouteNode)currNode).right;
16   }
17   BaseNode base = currNode asInstanceOf BaseNode;
18   statLock(base.lock);
19   if (! base.valid) {
20     statUnlock(base.lock);
21     return doOperation(tree, operation, key, v); // retry
22   } else {
23     Object result = operation.execute(base, key, v);
24     if (base.lock.statistics > MAX_CONTENTION) {
25       if (sizeLessThanTwo(base.root)) base.lock.statistics = 0;
26       else highContentionSplit(tree, base, prevNode);
27     } else if (base.lock.statistics < MIN_CONTENTION) {
28       if (prevNode == null) base.lock.statistics = 0;
29       else lowContentionJoin(tree, base, prevNode);
30     }
31     statUnlock(base.lock);
32     return result;
33   }
34 }
```

Figure 4: Generic pseudocode for single-key operations (insert, remove, get, etc.).

```
35 void highContentionSplit(CATree tree, BaseNode base, RouteNode parent) {
36   K splitKey = pickSplitKey(base.root);
37   part1, part2 = splitTree(splitKey, base.root);
38   RouteNode newRoute =
39     new RouteNode(new BaseNode(part1), splitKey, new BaseNode(part2));
40   base.valid = false;
41   if (parent == null) tree.root = newRoute;
42   else if (parent.left == base) parent.left = newRoute;
43        else parent.right = newRoute;
44 }
```

Figure 5: High-contention adaptation.

base node. (ii) Lines 18 and 19 lock the statistics lock in the base node and check the valid flag. If the valid flag is false the base node lock has to be unlocked (Line 20) and the operation has to be restarted (Line 21). (iii) Line 23 executes the operation on the sequential ordered set data structure inside the base node. (iv) Lines 24 to 30 evaluate the statistics variable and adapt the CA tree accordingly. Here one can add additional constraints for the adaptation. For example one might want to limit the total number of routing nodes or the number of routing nodes that can be traversed before a base node is reached. (v) Lines 31 and 32 finish the operation by unlocking the base node and returning the result from the operation. Below we describe the algorithms for high and low contention adaptation in detail.

3.4. High-contention Adaptation

High-contention adaptation is performed by splitting the contended base node. This creates two new base nodes each containing roughly half the items of the original base node. These two new nodes are linked with a new routing node containing a routing key K so that all keys in the left branch are smaller than K and the right branch contains the rest of the keys. Figure 5 contains the code. In Line 36, `pickSplitKey` picks a key that ideally divides the sequential ordered set data structure in half. The statement in Line 37 splits the data structure according to its split key.

The new routing node can be linked in at the place of the old base node without taking any additional locks or checking that the parent node is still the parent. The reason why it is correct to do so is because the parent of a base node cannot be changed when the base node is locked and has the valid flag set to true. It is easy to see that `highContentionSplit` preserves this invariant that we call the *fixed parent invariant*.

3.5. Low-contention Adaptation

Figure 6 shows the algorithm for `lowContentionJoin`. The goal of the function is to splice out the base node with low contention from the tree and transfer its data items to one neighboring base node. The code looks complicated at first glance but is actually very simple. Many of the **`if`** statements just handle symmetric cases for the left and right branch of a node. In fact, we just show the code for the case when the base node with low contention (called `base` in the code) is the left child of its parent routing node. (The rest of the code is completely symmetric.) Also, the following description will just explain the case when the base node with low contention is the left child of its parent.

In Line 47, we find the leftmost base node of the `parent`'s right branch. We try to lock this `neighborBase` in Line 48. If we fail to lock it or if `neighborBase` is invalid (Line 50 checks this) we reset the lock statistics and return without doing any adaptation. One can view these cases as that it is not a good idea to do adaptation now because the neighbor seems to be contended. Note that if instead of the `statTryLock` call we had used a forcing lock call, we could end up in a deadlock situation because our `base` could be another thread's `neighborBase` and vice versa. In Line 54, we know that there are no keys between the maximum key in `base` and the minimum key in `neighborBase`. (If there were, `neighborBase` would not have been valid.) We also know that there cannot be any keys between `base` and `neighborBase` as long as we are holding the locks of `base` and `neighborBase`, because one of these locks is held in all places where a base node could be added.

To complete the operation, we will first splice out the parent of `base` so that threads will be routed to the location of `neighborBase` instead of `base`. To do this, we can change the link to `parent` in the grandparent of `base` so that it points to the right child of `parent`. Splicing out the parent without acquiring any locks is not safe. The parent's right child pointer could be changed at any time by a concurrent low-contention adapting thread. Additionally, the grandparent could be deleted at any time by a concurrent low-contention adapting thread. To protect from concurrent threads changing the right pointer of the parent or the grandparent we require that the locks of both parent and grandparent (if the grandparent is not the root pointer) are acquired while we do the splicing. After acquiring the grandparent's lock, we also need to ensure that the

```
45 void lowContentionJoin(CATree tree, BaseNode base, RouteNode parent) {
46   if (parent.left == base) {
47     BaseNode neighborBase = leftmostBaseNode(parent.right);
48     if (! statTryLock(neighborBase.lock)) {
49       base.lock.statistics = 0;
50     } else if (! neighborBase.valid) {
51       statUnlock(neighborBase.lock);
52       base.lock.statistics = 0;
53     } else {
54       lock(parent.mutex);
55       parent.valid = false;
56       neighborBase.valid = false;
57       base.valid = false;
58       RouteNode gparent = null; // gparent = grandparent
59       do {
60         if (gparent != null) unlock(gparent.mutex);
61         gparent = parentOf(parent, tree);
62         if (gparent != null) lock(gparent.mutex);
63       } while (gparent != null && !gparent.valid);
64       if (gparent == null) {
65         tree.root = parent.right;
66       } else if (gparent.left == parent) {
67         gparent.left = parent.right;
68       } else {
69         gparent.right = parent.right;
70       }
71       unlock(parent.mutex);
72       if (gparent != null) unlock(gparent.mutex);
73       BaseNode newNeighborBase =
74         new BaseNode(join(base.root, neighborBase.root));
75       RouteNode neighborBaseParent = null;
76       if(parent.right == neighborBase) neighborBaseParent = gparent;
77       else neighborBaseParent = leftmostRouteNode(parent.right);
78       if(neighborBaseParent == null) {
79         tree.root = newNeighborBase;
80       } else if (neighborBaseParent.left == neighborBase) {
81         neighborBaseParent.left = newNeighborBase;
82       } else {
83         neighborBaseParent.right = newNeighborBase;
84       }
85       statUnlock(neighborBase.lock);
86     }
87   } else { ... } /* This case is symmetric to the previous one */
88 }
```

Figure 6: Low-contention adaptation.

grandparent has not been spliced out from the tree by checking its valid flag. Acquiring the lock of the parent (Line 54) is straightforward since we know that it is still our parent because of the fixed parent invariant. Acquiring the lock of the grandparent (Lines 58 to 63) is a little bit more involved. We repeatedly search the tree for the parent of `parent` until we find that the root pointer points to `parent` (`parentOf` returns `null`) or until we manage to take the lock of the grandparent and have verified that it is still in the tree by checking its valid flag. If the grandparent is the root pointer, we can

```
void manageCont(CATree tree, BaseNode base, RouteNode parent,
                boolean contended) {
  if (contended) base.lock.statistics += FAIL_CONTRIB;
  else base.lock.statistics -= SUCC_CONTRIB;
  if (base.lock.statistics > MAX_CONTENTION) {
    if (sizeLessThanTwo(base.root)) base.lock.statistics = 0;
    else highContentionSplit(tree, base, parent);
  } else if (base.lock.statistics < MIN_CONTENTION) {
    if (parent == null) base.lock.statistics = 0;
    else lowContentionJoin(tree, base, parent);
  }
}
```

Figure 7: Manage contention.

be certain that it will not be modified. This is because if a concurrent low-contention adaptation thread were to change the root pointer, it would first need to acquire the lock of `base`, which it cannot. Now we can splice out the parent (Lines 64 to 70) and unlock the routing node lock(s) that we have taken (Lines 71 and 72). The splicing out of the parent cannot falsify the fixed parent invariant. The only parents of base nodes the splicing out could change are `neighborBase` and `base`, which have got their valid flags set to false at Lines 56 and 57.

At this stage, it is safe to link in a new base node containing the union of the keys in `base` and `neighborBase` at the place of the old `neighborBase` (Lines 73 to 84). Notice that it is important that we mark `neighborBase` and `base` invalid (Lines 56 and 57) before we unlock them to make waiting threads retry their operations. Notice also that the parent of `neighborBase` might have been changed by Lines 64 to 70 so it would not have been safe to use the parent of `neighborBase` at the time of executing Line 47.

3.6. Multi-key Operations

CA trees also support operations that atomically operate on several keys, such as bulk insert, bulk remove, and swap operations that swap the values associated with two keys. Generic pseudocode for such operations appears in Fig. 8; its helper function `manageCont` appears in Fig. 7. Such operations start by sorting the elements given as their parameter (Line 108). Then all the base nodes needed for the operations are found (Line 113) and locked (Lines 116 and 117) in sorted order. Locking base nodes in a specific order prevents deadlocks. The function `lockIsContended` locks the base node without recording any statistics and returns `true` if contention was detected while locking it. The function `lockNoStats` just locks the base node lock without recording any statistics. When multi-key operations are given keys that all reside in one base node, naturally it suffices to lock only this base node. To detect this scenario, one simply has to query the sequential data structure in the current base node for the maximum key (Line 127). This can be compared to data structures that utilize non-adaptive fine-grained synchronization and thus either need to lock the whole data structure or all involved elements individually. Finally, multi-key operations end by adjusting the contention

```
101 Object[] doBulkOp(CATree tree, Op op, K[] keys, Object[] es) {
102   keys = keys.clone(); es = es.clone();
103   BaseNode baseNode;
104   RouteNode parent;
105   Object[] returnArray = new Object[keys.size];
106   boolean first = true;
107   boolean firstContended = true;
108   sort(keys, es);
109   List<(BaseNode, RouteNode)> lockedBaseNodes = new List<>();
110   int i = 0;
111   while (i < keys.size()) {
112    find_base_node_for_key:
113     baseNode, parent = getBaseNodeAndParent(tree, keys[i]);
114     if (lockedBaseNodes.isEmpty() || baseNode != lockedBaseNodes.last().elem1){
115       if (first) {
116         firstContended = lockIsContended(baseNode.lock);
117       } else lockNoStats(baseNode.lock);
118       if (! baseNode.valid) {
119         unlock(baseNode.lock);
120         goto find_base_node_for_key; // retry
121       }
122       lockedBaseNodes.addLast((baseNode, parent));
123     }
124     first = false;
125     returnArray[i] = op.execute(baseNode, keys[i], es[i]);
126     i++;
127     K maxKey = maxKey(baseNode.root);
128     while (i < keys.size() && maxKey != null && keys[i] <= maxKey) {
129       returnArray[i] = op.execute(baseNode, keys[i], es[i]);
130       i++;
131     }
132   }
133   if (lockedBaseNodes.size() == 1) {
134     baseNode, parent = lockedBaseNodes.get(0);
135     manageCont(tree, baseNode, parent, firstContended);
136     unlock(baseNode.lock);
137   } else {
138     for (i = 0; i < lockedBaseNodes.size(); i++) {
139       baseNode, parent = lockedBaseNodes.get(i);
140       if (i == (lockedBaseNodes.size()-1)) {
141         manageCont(tree, baseNode, parent, false);
142       } else baseNode.lock.statistics -= SUCC_CONTRIB;
143       unlock(baseNode.lock);
144     }
145   }
146   return returnArray;
147 }
```

Figure 8: Bulk operations.

statistics, unlock all acquired locks and, if required, split or join one of the base nodes (Lines 133 to 145).

```
BaseNode, List<RouteNode>
getNextBaseNodeAndPath(BaseNode b, List<RouteNode> p) {
  List<RouteNode> newPathPart;
  BaseNode bRet;
  if (p.isEmpty()) // The parent of b is the root
    return new Tuple(null, null);
  else {
    List<RouteNode> rp = p.reverse();
    if (rp.head().left == b) {
      bRet, newPathPart =
        leftmostBaseNodeAndPath(rp.head().right);
      return bRet, p.append(newPathPart);
    } else {
      K pKey = rp.head().key; // pKey = key of parent
      rp.removeFirst();
      while (rp.notEmpty()) {
        if (rp.head().valid && pKey < rp.head().key) {
          bRet, newPathPart =
            leftmostBaseNodeAndPath(rp.head().right);
          return bRet, rp.reverse().append(newPathPart);
        } else {
          rp.removeFirst();
        }
      }
    }
    return null, null;
  }
}
```

Figure 9: Find next base node.

3.7. Range Operations

We will now describe an algorithm for linearizable range operations that locks all base nodes that can contain keys in the range $[a, b]$. Generic pseudocode for such operations can be seen in Fig. 10. The helper function `getNextBaseNodeAndPath` that finds the next base node to lock appears in Fig. 9. To prevent deadlocks, the base nodes are always locked in increasing order of the keys that they can contain. Therefore, the first base node to lock is the one that can contain the smallest key a in the range. This first base node is found and locked at Lines 179 to 185 using the algorithm described for single-key operations but, in contrast to the algorithm for single key operations, here we also record the routing nodes on the path to the base node in the variable `path`. Finding the next base node (Lines 192 to 202) is not as simple as it might first seem because routing nodes can be spliced out and base nodes can be split. The two problematic cases that may occur are illustrated in Fig. 2. Suppose that the base node marked B_1 has been found through the search path with routing nodes with keys $80, 40, 70,$ and 60 as depicted in Fig. 2b. If the tree stays as depicted in Fig. 2b, the base node B_2 would be the next base node. However, B_2 may be split (Fig. 2a) or spliced out while the range operation is traversing the routing nodes (Fig. 2c). If one of these cases happens, the search may end up in the incorrect base node. However, this will be detected (Line 198) since the base node that the search ends up in will be invalid. Searches for the next base node that end up in an invalid base node will be retried (Line 201). When we find the

```
Object[] rangeOp(CATree tree, Op op, K lo, K hi) {
  List<RouteNode> path; BaseNode baseNode; RouteNode parent;
  List<(BaseNode, RouteNode)> lockedBaseNodes = new List<>();
 fetch_first_node:
  baseNode, path = getBaseNodeAndPath(tree, lo);
  boolean firstContended = lockIsContended(baseNode.lock);
  if (! baseNode.valid) {
     unlock(baseNode.lock);
     goto fetch_first_node;
  }
  while (true) {
    lockedBaseNodes.addLast((baseNode, path.last()));
    K baseNodeMaxKey = maxKey(baseNode.root);
    if (baseNodeMaxKey != null && hi <= baseNodeMaxKey)
      break; // All needed base nodes are locked
    BaseNode lastLockedBaseNode = baseNode;
   search_next_base_node:
    List<RouteNode> pathBackup = path.clone();
    baseNode, path = getNextBaseNodeAndPath(lastLockedBaseNode, path);
    if (baseNode == null)
      break;
    lockNoStats(baseNode.lock);
    if (! baseNode.valid) { // Try again
      unlock(baseNode.lock);
      path = pathBackup;
      goto search_next_base_node;
    }
  }
  ArrayList<Object> buff = new ArrayList<>();
  if (lockedBaseNodes.size() == 1) {
    baseNode, parent = lockedBaseNodes.get(0);
    buff.addAll(rangeOp(baseNode, op, lo, hi));
    manageCont(tree, baseNode, parent, firstContended);
    unlock(baseNode.lock);
  } else {
    for (int i = 0; i < lockedBaseNodes.size(); i++) {
      baseNode, parent = lockedBaseNodes.get(i);
      buff.addAll(rangeOp(baseNode, op, lo, hi));
      if (i == (lockedBaseNodes.size()-1)) {
        manageCont(tree, baseNode, parent, false);
      } else baseNode.lock.statistics -= SUCC_CONTRIB;
      unlock(baseNode.lock);
    }
  }
  return buff.toArray();
}
```

Figure 10: Range operations.

next base node we will *not* end up in the same invalid base node twice if the following algorithm (also depicted in Fig. 9) is applied:

1. If the last locked base node is the left child of its parent routing node P then find the leftmost base node in the right child of P (Fig. 9, Line 158).

2. Otherwise, follow the reverse search path from P until a *valid* routing node R

with a key greater than the key of P is found (Fig. 9, Line 164). If such an R is not found, the current base node is the rightmost base node in the tree so all required base nodes are already locked (Fig. 9, Lines 153 and 173). Otherwise, find the leftmost base node in the right branch of R (Fig. 9, Line 166).

The argument why this algorithm is correct is briefly as follows. For the former case (Item 1), note that the parent of a base node is guaranteed to stay the same while the base node is valid. For the latter case (Item 2), note that once we have locked a valid base node we know that no routing nodes can be added to the search path that was used to find the base node, since the base node in the top of the path must be locked for a new routing node to be linked in. Also, the above algorithm never ends up in the same invalid base node more than once since the effect of a split or a join is visible after the involved base nodes have been unlocked. Finally, if the algorithm ever finds a base node B_2 that is locked and valid and the previously locked base node is B_1, then there cannot be any other base node B' containing keys between the maximum key of B_1 and the minimum key of B_2. This is true because if a split or a join were to create such a B', then B_2 would not be valid.

4. Properties

In this section, we first formulate the correctness properties (*linearizability*, *deadlock freedom*, and *livelock freedom*) that CA trees guarantee, and provide detailed proof sketches for them (Section 4.1). We then discuss starvation freedom (Section 4.2) and the complexity of CA tree operations (Section 4.3).

4.1. Correctness Proofs

We make the following assumptions about initialization: (i) the valid flags in base nodes and routing nodes are initially set to true, and (ii) the root pointer is initially set to a base node with a sequential data structure containing zero keys. We will also use the following definitions in the proofs:

Definition 1. ***(Inside)*** *A routing node R is inside a CA tree C if R's valid flag is set to true and R is reachable from the root of C. A base node B is inside a routing node if B's valid flag is set to true and B is reachable from R. A key is inside a CA tree C if it is stored in a base node that is inside C.*

Definition 2. ***(Validated node)*** *A thread T has validated a base node or routing node N if T has read the valid flag in N and the flag's value was true.*

Definition 3. ***(Quiescent state version of a CA tree)*** *The quiescent state version of a CA tree C at time t is the CA tree that is created by blocking all threads that are not holding any base node lock(s) at time t and continuing executing all threads that are holding base node locks, with the exception that the threads skip calls to `highContentionSplit` and `lowContentionJoin`, until no base node lock is held by any thread.*

Definition 4. ***(Represented set)*** *The set of keys represented by a CA tree C at a time point t is the set of keys that are inside the quiescent state version of C at time t.*

Definition 5. ***(Binary Search Tree (BST) property)*** *A CA tree routing node R satisfies the binary search tree (BST) property if all keys that are inside the left branch of R are less than R's key and all keys that are inside the right branch of R are greater than or equal to R's key. A CA tree satisfies the BST property if all routing nodes that are inside the CA tree satisfy the BST property.*

Definition 6. ***(Linearization point)*** *The linearization point for an operation that is performed by acquiring locks is at any code point where the operation is holding all base node locks of the base nodes that the operation operates on. The linearization point for an operation that is performed by a successful optimistic read attempt is at any code point between the two scans of the sequence locks in the base nodes that the operation is accessing.*

In addition, we will rely on the following observations that can be validated from the structure of the pseudocode:

Observation 1. *An operation never changes the sequential data structure in a base node or a base/routing node's valid flag without holding the lock of the node.*

Observation 2. *A valid flag is never set to true after initialization.*

We claim that the following are invariants for the CA tree:

- I If a thread T finds a base node B by searching from the root of a CA tree C that T subsequently locks and validates at time point t, then B is inside C from the time t until the time that either T sets the valid flag of B to false or releases B's lock.
- II The parent of a base node B is the same as when B was inserted into the CA tree as long as the base node B is inside the CA tree.
- III The root pointer of a CA tree is never changed without holding the lock in the node that the root pointer points to.
- IV All pointers in the routing layer of a CA tree are pointing to different objects.
- V A left or right pointer of a routing node that points to a base node B is never changed without holding B's lock.
- VI A left or right pointer of a routing node R that points to a routing node is never changed without holding R's lock.

We will now state and give proofs for lemmas that are later used to prove the main theorems. Whenever the proofs for the two symmetric cases in `lowContentionJoin` are similar, for brevity we will only present the proof for the case for which we display pseudocode in Fig. 6.

Lemma 1. *Invariants I to VI always hold.*

Proof: We use a proof by induction to prove that the invariants always hold. The invariants are initially true since a CA tree initially consists of only one base node pointed to by the root pointer. The code lines that might make the invariants false are changes to the pointers in the routing layer and changes to valid flags. These changes are all atomic and appear on Lines 40 to 43, 55 to 57, 65, 67, 69, 79, 81 and 83, which are all located in the functions `highContentionSplit` (Fig. 5) and `lowContentionJoin` (Fig. 6). Our induction hypothesis is that the invariants hold just before the changes on the listed lines. The proof is completed by proving, for all listed changes, that given the induction hypothesis the invariants hold also after the change.

We will use the change on Line 41 (Fig. 5) as an example of how we can prove that the invariants hold even after a change given the induction hypothesis. The rest of the changes can be handled using similar arguments and are therefore omitted for brevity. From the assumption that Invariants I, II and III hold just before Line 41, it follows that the base node `base` was pointed to by the root of the tree just before the change. (`base` is locked and validated at all places that call `highContentionSplit`.) Therefore, given the induction hypothesis, Invariants II, III and IV clearly hold after the change. (The change atomically sets the root pointer of the CA tree so that it points to a new routing node with left and right pointers storing references to two new base nodes.) Furthermore, as the change does not make any base node other than `base` unreachable from the root, it follows from Observation 2 and from the fact that Line 40 sets `base`'s valid flag to false that Invariant I still holds after the change. Invariants V and VI are unaffected by the change of the root pointer since they concern changes to the left and right pointers of routing nodes. □

Lemma 2. *When a thread T finds a base node B following a path L of pointers (e.g., $[r_1.left, r_2.right, r_3.left, \ldots]$) starting from the root of a CA tree and subsequently locks and validates B, then the only change that another thread can do to the path L while B is locked by T is that a pointer gets spliced out atomically from the path. In this case, the routing node in which the pointer is located has had its valid flag set to false.*

Proof: Only changes to pointers in the routing layer could modify the path. Such changes only occur atomically in the functions for high- and low-contention adaptation. We use a proof by induction to prove the lemma. Our base case is that no changes to the routing layer have happened, in which case the lemma trivially holds. Our induction hypothesis is that the invariants hold directly before the atomic changes to the routing layer. We will now complete the proof by proving that the lemma must hold even after the changes to the routing layer given the induction hypothesis.

The function for high-contention adaptation cannot change the path. Firstly, the function `highContentionSplit` cannot change L after B has been locked by T since the function only changes one pointer to a base node for which it holds the lock (see Invariants I, II, III and V), and there is only one pointer on the path L that points to a base node and T is holding the lock of that base node. Secondly, if `highContentionSplit`'s change occurred before B was locked by T, then T must have seen the result of the change if T traversed the changed pointer since B's valid flag is true (cf. Observation 2 and Line 40).

Function `lowContentionJoin` performs at most two atomic changes to pointers in the routing layer. The first change happens between Lines 64 to 70. This change splices out exactly one base node and its parent routing node from the CA tree. To see this, note that, at the time of the change, the variable `base` holds a reference to a base node that is inside the CA tree (Invariant I), the variable `parent` holds a reference to the parent of `base` (Invariant II), `gparent` holds a reference to the grandparent of `base` or is `null`, in which case the root of the tree is the grandparent, (see the induction hypothesis), and the right pointer of parent as well as the pointer to parent in the grandparent cannot be changed while the acquired locks are held (see Invariants III, V and VI). Since we know that B is different from the spliced out base node, the only effect this change may have on L is to splice out one pointer. Thus, as the routing node holding the spliced out pointer has got invalidated (Line 55) before the splicing out happened, the lemma still holds after the change given the induction hypothesis. The second change that is done by `lowContentionJoin` is to replace one base node with another one (Lines 78 to 84). This change is very similar to the change done by `highContentionSplit` and can be handled in the same way. $\square$

Lemma 3. *If a thread T has searched for a key k in a CA tree C using the BST property and ended up in a base node B that T has subsequently locked and validated, then a search for k using the BST property in a quiescent state version of C would end up in B as well, as long as T is still holding the lock of B and has not called the functions for high-contention split or low-contention join.*

Proof: It follows from Definition 3 that the only event that might affect the truth of this lemma is that another thread is changing (or is holding a lock of at least one base node and is about to change) the search path to B concurrently with the operation whose search ended up in B. This is because the routing nodes would otherwise be identical in the actual search path and the path to B in the quiescent state version of the CA tree. It follows from Lemma 2 that even if such an event happens concurrently, a search for k will still end up in B in the quiescent state version of the CA tree. $\square$

Lemma 4. *The execution of the functions for high-contention split and low-contention join does not change the set represented by the CA tree and maintains the BST property in the quiescent state version of the CA tree.*

Proof: Firstly, note that, as we have already argued in the proof for Lemma 2, it follows from the invariants that the function for low-contention join (Fig. 6) splices out the base node `base` and its parent in one atomic step. While this change takes place the caller of `lowContentionJoin` is holding a lock of a base node referred to by the variable `neighborBase`. According to the BST property, `neighborBase` is at the new location for the keys in `base` after `base` and its parent has been spliced out. To see why this holds, note that Invariants I, II and IV together with Lemma 2 tell us that when we have locked and validated `neighborBase` (on Lines 48 and 50), then we know that `neighborBase` will be the *leftmost* base node in the right child of the parent of `base` until `base` is spliced out from the tree. The final change to the routing layer that is done by `lowContentionJoin` is to replace `neighborBase` (while `neighborBase` is still locked) with a new base node containing the keys of both `base` and `neighborBase` (Lines 78 to 84). From the above follows that the function for low-contention join does not change the set represented by the CA tree and that low-contention join maintains the BST property in the quiescent state version of the CA tree.

The argument for why the function for high-contention split preserves the set represented by the CA tree and maintains the BST property is similar. $\square$

Lemma 5. *Additions and removals of keys in the sequential data structures of base nodes maintain the BST property in the quiescent state version of the CA tree.*

Proof: Single-key operations (Fig. 4) and bulk operations (Fig. 8) are the only type of operations that insert or remove keys in base nodes' sequential data structures. From Lemma 3 it follows that single-key operations maintain the BST property in the quiescent state version of the CA tree. Bulk operations are constructed from multiple single-key operations with the exception of the optimization that avoids searches in the routing layer by checking the maximum key in the last locked base node (see Line 127). If this optimization would result in a violation of the BST property in the quiescent state version of the CA tree then the property would already have been violated so there would not be any BST property to maintain. □

Lemma 6. *The quiescent state version of a CA tree always satisfies the BST property.*

Proof: Initially, the lemma trivially holds. It is easy to see that the only changes that may cause a violation of the BST property in the quiescent state version of a CA tree are additions and removals of keys in the sequential data structures of base nodes and changes in the routing layer. Lemmas 4 and 5 tell us that such changes maintain the BST property in the quiescent state version of the CA tree. □

Lemma 7. *If a search that is using the BST property to search for a key k in a CA tree C ends up in a base node B which is subsequently locked and validated then:*

1. *k is in the set represented by C if and only if k is inside the sequential data structure S rooted at B and,*
2. *if the minimum key in S is k_1 and the maximum key in S is k_2 then the keys in $[k_1, k_2]$ are in the set represented by C if and only if they are in S.*

Proof: This follows from Definition 4 and Lemmas 3 and 6. □

Lemma 8. *A routing node R's valid flag is set to false by a thread T iff:(i) one of R's child nodes is a base node B that is locked by T, and (ii) T will splice out R and B from the tree and set the valid flag of B to false while still holding the lock of B.*

Proof: First, notice that Line 55 where the valid flag of a routing node referred to by the variable `parent` is set to false is the only place where a valid flag of a routing node is changed. Secondly, as already argued in the proof of Lemma 2, `parent` is spliced out together with its child `base` between Lines 64 to 70. Finally, the valid flag of `base` is set to false at Line 57. □

Lemma 9. *The algorithm for range operations (Fig. 10) locks all base nodes that may contain keys in the specified range.*

Proof: It follows from Lemma 7 that the first base node that is locked by the algorithm for range operations must contain the first key in the specified range if it is present in the set. Furthermore, it follows from Definition 3 together with Lemmas 2, 6 and 8 that if the range operation's search for the next base node (Fig. 9) ends up in a base node B that is locked and validated, then there are no keys in the set represented by the CA tree between the maximum key in the previously locked base node and the minimum key in B and no such keys can be added while the locks are held.

The algorithm for range operations attempts to lock base nodes until one of the following two conditions is met. The first condition is fulfilled if we reach a base node containing a key that is greater than or equal to the maximum key in the range (Line 189). Then, clearly we have locked all base nodes that may contain keys in the range since there are no keys between the maximum and minimum key of two consecutively locked base nodes, and we have locked the base node that must contain the first key in the range (if it is present) as well as a base node that contains a key that is greater than or equal to the largest key in the range. The second condition is when the algorithm cannot find any more base nodes to lock (Line 195). Definition 3 together with Lemmas 2 and 6 tell us that:(i) this can only happen when we have locked the base node that can contain the largest key, and (ii) no base node that can contain an even larger key can be added while the locks are held. □

Theorem 1. ***(Linearizability)*** *All operations on the set represented by a CA tree appear to happen instantly at the time of their linearization points (Definition 6).*

Proof: Lemma 7 tells us that when an operation adds, removes or looks up a key in the sequential data structure S of one of the base nodes, then this key cannot be in the sequential data structure of another base node and the key is inside S if and only if it is in the set. Additionally, Lemma 9 tells us that the range operation is performed on all sequential data structures of base nodes that may have keys in the range. An operation is holding the locks of all base nodes in which it operates, which prevents any other operations from making use of intermediate changes done by the operation and thus the operation will appear to happen at its linearization point. We can therefore conclude that all CA tree operations are linearizable. □

Theorem 2. ***(Deadlock freedom)*** *The CA tree operations are deadlock free.*

Proof: We will prove that the CA tree operations are deadlock free by showing that all threads either obtain locks in a specific order (thus, a deadlock cannot occur), or prevent a deadlock situation by using `tryLock` which, if unsuccessful, is followed by the release of the currently held locks.

We will first prove that a call to function `lowContentionJoin` (Fig. 6) cannot cause a deadlock. Notice that everywhere `lowContentionJoin` is called the lock of the base node given as parameter (called `base`) is held, the caller holds no other CA tree locks and `base` is always released after the call to `lowContentionJoin` has returned. Operations that call `lowContentionJoin` can use `tryLock` (Fig. 6, Line 48) to lock another base node. If the `tryLock` is unsuccessful, `lowContentionJoin` will return and the currently held lock will be released (Lines 31, 136, 143, 209 and 217). Also, `lowContentionJoin` is the only function that acquires locks in the routing nodes. Routing nodes are always locked after the base node locks. `lowContentionJoin` always acquires the parent routing node's lock before the grandparent routing node's lock (Lemma 2), so locking of routing nodes is ordered by the distance to the root of the tree. (Since no operation ever holds two routing node locks that are at the same level, it is not a problem that there is no order between routing nodes at the same level.)

The only other functions that can hold more than one base node locks are `doBulkOp` (Fig. 8) and `rangeOp` (Fig. 10). We will now prove that these functions always lock base nodes that are inside the quiescent state version of the tree in the left to right order (when depicted as in Fig. 1) and can thus not cause a deadlock situation. They both only hold the lock of at most one invalid base node since they immediately unlock a base node that is invalid after it has been locked, so we only need to consider base nodes that are inside the CA tree.

(i) The function `doBulkOp` sorts the keys (Fig. 8, Line 108) so smaller keys are processed before larger ones. Therefore, it follows from Lemmas 3 and 6 that a valid base node that is locked on Line 117 is ordered after all base nodes that the function already holds the lock for.

(ii) The function `rangeOp` (Fig. 10) finds the next base node to lock in the subtree rooted at the right branch of the first routing node on the reverse path to the previously locked base node that does not contain the previously locked base node (cf. the proof of Lemma 9). Therefore, it follows from Lemmas 2, 3 and 6 that base nodes that are locked by `rangeOp` are ordered after all base nodes that the operation has previously locked.

We can therefore conclude that the CA tree operations are deadlock free. All locks are acquired in a specific order or otherwise the lock is acquired by a `tryLock` call and, if the `tryLock` fails, all currently held locks are released. □

Theorem 3. ***(Livelock freedom)*** *The CA tree operations are livelock free.*

Proof: A livelock occurs when threads perform some actions that interfere with each other so that no thread makes any actual progress. There are only two situations when CA tree operations need to redo some steps because of interference from other threads:

(i) A thread needs to retry an operation or part of an operation if an invalid base node is encountered. The interfering thread must have completed an operation in this case. Otherwise no split or join could have happened. Furthermore, since a base node or routing node is invalidated and linked out from the tree while it is locked (cf. the proof of Lemma 2), a search will never end up in the same invalid base node when it is retried. For example, consider the case when the search for the next base node in `rangeOp` (Fig. 10) ends up in an invalid base node B because B and its parent R have been spliced out from the tree and R was previously on the path to the previously locked base node. Then, when the search for the next base node to lock is retried, the search will *not* end up in B again because of the validity check on Line 164 (Fig. 9).

(ii) Similarly, if the code in Fig. 6 (Lines 58 and 59) needs to be retried to find the grandparent of a base node, another interfering thread must have spliced out a routing node and has thus made progress.

The CA tree operations are therefore livelock free. □

4.2. Starvation Freedom

Even though CA trees are livelock free, individual operations can still be starved as in many high performance concurrent data structures [2, 5, 6]. Intuitively, it seems unlikely that this is a problem in practice because splits and joins happen relatively infrequently. Furthermore, since splits and joins of base nodes are not needed for correctness, one can introduce a simple extension, with low performance penalty in the common case, that would make all CA tree operations starvation free. This extension is implemented by adding a counter, whose value is initially zero, to the CA tree data structure. The functions for low- and high-contention adaptation then have to start by reading this counter, and aborting without performing any adaptation if the counter has a non-zero value. An operation that has performed more than some constant number of retries increments the counter atomically, thus stopping new adaptations from happening, to ensure that the operation will eventually complete. The counter is atomically decremented again when the operation has executed successfully so that adaptations can be enabled again. Of course, for this extension to make CA trees starvation free, all locks need to be starvation free so that a thread cannot get stuck forever in a lock acquisition.

4.3. Time Complexity

We will now derive the sequential access time complexity of the CA tree operations. Later we will argue that under some reasonable assumptions the expected execution time of an operation when a CA tree is accessed concurrently will be close to the operation's sequential execution time.

Let us assume that when an operation starts in a CA tree the number of routing nodes is D and the total number of keys in the CA tree is N. Furthermore, let us also assume that the sequential data structure operation *op* that is applied by a single-key operation and the join and split operations on the sequential data structure component all have worst case time complexity $\mathcal{O}(log(N'))$, where N' is the total number of keys in the data structure(s). The worst case sequential execution time for a single-key operation is then $\mathcal{O}(log(N) + D)$. This is because: (i) the maximum time spend on searching for the base node B is D; (ii) the application of *op* in the sequential data structure stored in B takes at most $\mathcal{O}(log(N))$ time; (iii) high-contention split only performs a constant amount of work plus a split operation in the sequential data structure that contains at most $N + 1$ keys; and (iv) low-contention join traverses at most D routing nodes and performs a join of two sequential data structures that in total contain at most $N + 1$ keys.

Let us now also assume that the size of the range given to a range operation is R, the time complexity of finding the position of a key in the sequential data structure of size N' is $\mathcal{O}(log(N'))$, finding the position of the smallest key in the sequential data structure can be done in constant time, and traversing the I following keys in increasing order given a key position can be done in $\mathcal{O}(I)$ time. Using these assumptions we can derive that the sequential worst case time complexity for a range operation is $\mathcal{O}(D + log(N) + R)$. To see why this is so, note that:(i) a range operation only needs to find the position of the first key in the range in one sequential data structure; (ii) at most $\mathcal{O}(D)$ routing nodes need to be traversed; and (iii) the range operation will perform at most one high- or low-contention adaptation.

As a bulk operation sorts the k key-value pairs that the operation is given as input, we assume that this sorting takes $\mathcal{O}(k * log(k))$ time. We also assume that the operation op that is applied to a sequential data structure for each key-value pair that is given to the bulk operation has time complexity $\mathcal{O}(log(N'))$, where N' is the number of keys in the sequential data structure. Using these, we can derive that the sequential access worst case time complexity for bulk operations is $\mathcal{O}(k * log(k) + k * (D + log(N + k))) = \mathcal{O}(k * (D + log(N + k)))$. This is because:(i) a bulk operation might need to traverse the routing nodes once for each key that it operates on; (ii) the sequential data structure that is operated on will contain at most $N + k$ keys when op is applied for the last key-value pair; and (iii) as with the other operations at most one high- or low-contention adaptation will happen.

Adversary workloads could make D grow arbitrary large. Even though our experiments do not indicate that this is a problem in practice it could be desirable to limit D to a constant and thereby also improve the sequential worst case time complexity of the CA tree operations. Limiting D by a constant can be done by not performing high-contention splits in a base node B if the number of routing nodes on the path from the root of the CA tree to B is larger than some constant. It follows from Lemma 2 that we can get a conservative estimate (i.e., not an under-approximation) of the number of routing nodes from the root of the CA tree to the last base node that an operation locks (which is also the one where high-contention split might happen). This can be done by incrementing a thread local counter every time a routing node is traversed downwards and decrementing this counter every time a routing node is traversed upwards.

Obviously, the worst case execution time for an operation when the CA tree is accessed concurrently depends on the number of threads that are accessing the CA tree as well as the maximum time that threads can spend holding base node locks. However, if (i) the keys that operations access are random, (ii) the number of keys that operations access is small compared to the total number of keys in the tree, and (iii) contention is kept constant, then we can expect the average execution time for an operation to be close to its sequential execution time. The reason for this is that the eager high-contention adaptations and the low-contention adaptations that only happen after many uncontended accesses should make conflicts rare.

5. Important Components

In this section, we will present two extensions to the basic CA tree algorithms presented in Section 3 that are important for achieving good performance. The first extension changes the contention statistics in the base nodes accessed by multi-key operations according to heuristics that aim towards reducing synchronization-related overheads in future multi-key operations. The second extension can substantially improve the performance of CA trees in read-heavy workloads by avoiding writes to shared memory in read-only operations. It is easy to see that these two extensions preserve the properties of CA trees that we have just presented and proved. Still, we end this section by an argument why the second extension does not affect linearizability.

5.1. Adaptation and Contention Statistics in Multi-key Operations

Before unlocking the last base node that is accessed in a multi-key operation, low-contention join or high-contention split is performed on that base node if the contention thresholds are reached. The pseudocode that handles this can be found in Fig. 7; it is called from Lines 135 and 141 in Fig. 8 and Lines 208 and 215 in Fig. 10.

A multi-key operation that only requires one base node changes the contention statistics counter in the same way as single-key operations. (I.e., it increases the statistics counter with a big amount when contention is detected in the lock, and decreases the counter with a small amount if no contention is detected.)

On the other hand, if a multi-key operation requires access to more than one base node, the contention statistics counter is decreased (Lines 141 and 142 in Fig. 8 and Lines 215 and 216 in Fig. 10) in all involved base nodes. This is done in order to reduce the number of base node locks that future multi-key operations need to acquire. Note that multi-key operations can benefit from coarse grained locking as the overheads associated with acquiring and releasing locks can be reduced, but on the other hand coarse grained locking can also induce more contention. Therefore, this heuristics will on one hand reduce the overhead of acquiring unnecessary many locks but may increase the contention. However, the eager adaptations to high contention will soon redo the joining of base nodes if the contention level gets too high. Furthermore, frequent splits and joins back and forth are avoided as the adaptations that are performed to reduce the performance penalty of acquiring unnecessarily many locks are done much less eagerly than the adaptations to high contention.

5.2. Sequence Lock Optimization for Read-only Operations

Writing to shared memory when doing read-only operations can easily become a scalability bottleneck because of the induced cache coherence traffic. We now address this issue by describing an optimization using sequence locks. This optimization lets read-only operations execute without writing to shared memory when they do not conflict with write operations. A basic sequence lock consists of one integer counter that is initialized to an even number [24]. A thread acquires a sequence lock non-optimistically by first waiting until the counter has an even number and then attempting to increment it by executing an atomic compare-and-swap (CAS) instruction. The sequence lock has been acquired non-optimistically if the CAS instruction succeeds. To unlock the sequence lock, the integer counter is incremented by one so that the counter stores an even number again. By using sequence locks in the statistics lock implementation one can easily make read-only CA tree operations optimistically attempt to perform the operation without writing to shared memory. Additionally, for the sequence lock optimization to work correctly, one also has to make sure that a write operation that interferes with an optimistic read attempt never causes an infinite loop or crash in the reader [25]. Luckily, for the sequential data structures that we have experimented with, this is a trivial task. Essentially, one only has to ensure that critical reads are from memory to prevent a reader from caching inconsistent values in registers which could potentially make the reader stuck in an infinite loop.

This sequence lock optimization can also be used in read operations that need to read from several base nodes atomically. This can be done by first scanning all the sequence locks in the base nodes to be accessed before doing the read. This first scan checks that the sequence locks are unlocked and saves the read sequence numbers. The scan is aborted if a sequence lock is locked or if a valid flag in one of the involved base nodes is set to false. A second validation scan of the sequence locks needs to be performed after the read operation has executed to validate that no writer has interfered, by checking that all sequence numbers are the same as in the first scan. An operation whose optimistic read attempt fails will acquire the sequence lock(s) non-optimistically.

When an optimistic read attempt succeeds, the statistics counter in the base node locks that are accessed is not updated since that would be a write to shared memory and would therefore defeat the purpose of the optimistic read. If the optimistic attempt fails for an operation on a single base node, then the contention statistics is increased by the constant `SUCC_CONTRIB` to make optimistic read failures less likely in the future.

Preservation of Linearizability. If a read-only operation is successfully performed in an optimistic attempt, then the second scan of sequence numbers in the accessed base nodes ensures that an equivalent result could have been obtained by executing the operations non-optimistically. (I.e., acquiring the locks in the accessed base nodes during the first scan of the sequence numbers and unlocking them during the second scan.) Thus it follows from Theorem 1 that operations that are performed by a successful optimistic read attempts are linearizable.

6. More Optimizations

In Section 5.2, we addressed a key performance problem of the CA tree algorithm by describing how read-only operations can execute optimistically without writing to shared memory as long as there is no conflict with a write operation. In this section we discuss a few more optimizations that can be applied to CA trees.

Sequence Locks with Support for Non-optimistic Read-only Critical Sections. One obvious way of increasing the level of parallelism is to use a sequence lock that in addition to optimistic read-only critical sections and write-only critical sections also supports multiple parallel non-optimistic read-only critical sections (e.g., the StampedLock from the standard library of Java 8). In our implementation, we use such a lock and acquire the base node lock in non-optimistic read-only mode when the optimistic read attempt fails. If the optimistic read fails and the lock is acquired in read-mode and only one base node is required for the operation, then our implementation adds to the contention statistics to decrease the likelihood of optimistic read failures in the future.

An Optimization for Highly Contended Base Nodes. A base node that contains only one element cannot be split to reduce contention. Therefore, it can be advantageous to apply an optimization that puts contended base nodes that just contain a single element into a different state where operations can manipulate the base node with atomic CAS instructions or writes without acquiring the base node lock. The benefits of this optimization are twofold: blocking is avoided and the number of writes to shared memory for modifying operations can be reduced from at least three to just one (a CAS or a write instead of a lock call, a write and an unlock call).

On workloads with single key operations, when contention is high (i.e., on small set sizes and many threads), this optimization can increase the performance of the CA tree by as much as 100%, making it outperform many state-of-the-art data structures on these kinds of workloads. We refer to an earlier paper [15] for a detailed description of how to implement this optimization and for its experimental evaluation.

Parallel Critical Sections with Hardware Lock Elision. Support for hardware transactional memory has recently started to become commonplace with Intel's Haswell architecture [26]. A promising way to exploit the hardware transactional memory is through *hardware lock elision* (HLE) [27]. HLE allows ordinary lock-based critical sections to be transformed to transactional regions. A transaction can fail if there are store instructions that interfere with other store or load instructions or if the hardware transactional memory runs out of its capacity. If the transaction fails in the first attempt, an ordinary lock will be acquired making it impossible for other threads to enter the critical region. Since the size of the transactional region is limited by the hardware's capacity to store the read and write set of the transaction, an adaptive approach like the CA tree seems like a perfect fit for exploiting HLE. We refer to an earlier paper [15] for a more detailed discussion on using HLE for CA trees and for its evaluation. That evaluation, conducted on an Intel(R) Xeon(R) CPU E3-1230 v3 (3.30GHz) Haswell processor released in 2013, showed only a small benefit of using HLE over traditional locking in CA trees. Still, we expect that combining HLE with CA trees will become more attractive as the capacity and performance of transactional memories improves.

7. Related Work

We begin the comparison with related work with a brief overview of recently published data structures for concurrent ordered sets and a discussion of how CA trees compare with them. We subsequently present a detailed comparison with concurrent ordered sets that offer efficient support for multi-key operations. We also briefly mention work that is not directly related to concurrent ordered sets for modern multicores but that is worth mentioning in the context of approaches that adapt to contention.

Ordered Sets with Single-key Operations. Fraser [11] created the first lock-free ordered set data structure based on the skiplist, which is similar to ConcurrentSkipListMap (SkipList) in the Java standard library. Since Fraser's algorithm, several lock-free binary search trees have been proposed [5, 6, 7, 8, 9, 10]. The relaxed balancing external lock-free tree by Brown *et al.* (called Chromatic) is one of the best performing lock-free search trees [8]. Chromatic is based on the Red–Black tree algorithm but has a parameter for the degree of imbalance that can be tolerated. This parameter can be set to give a good trade-off between contention created by balancing rotations and the balance of the tree[2]. A number of well performing lock-based trees have also been put forward recently [1, 2, 3, 4]. The tree of Bronson *et al.* (called SnapTree) is a partially external

[2]In our experimental evaluation, we use the value 6 for Chromatic's degree of imbalance parameter, since this value gives a good trade-off between balance and contended performance [8].

tree inspired by the relaxed AVL tree by Bougé *et al.* [28]. The SnapTree simplifies the delete operation by delaying removal of nodes until the node is close to a leaf and uses an invisible read technique from software transactional memory to get fast read operations. The contention-friendly tree (CFTree) by Crain *et al.* provides very good performance under high contention by letting a separate thread traverse the tree to do balancing and node removal, thus delaying these operations to a point where other operations might have canceled out the imbalance [3]. The recently proposed LogAVL tree by Drachsler *et al.* [4] is fully internal in contrast to SnapTrees and CFTrees. Its tree nodes do not only have a left and right pointer but also pointers to next and previous nodes in the key order. This makes it possible for searches in the LogAVL tree to find the correct node even if the search is lead astray by concurrent rotations.

Our CA trees can be said to be *partially external trees* since the routing layer contains nodes that do not contain any values. In contrast to SnapTrees and CFTrees however, which are also partially external, the routing nodes in CA trees are not a remainder of delete operations but are created deliberately to reduce contention where needed. It is also a big advantage in languages like C and C++ without automatic memory management that CA trees can lock the whole subtree that will be modified. This makes it possible to directly deallocate nodes instead of using some form of delayed deallocation. Some kind of special memory management is still needed for the routing nodes but, since it is likely that routing nodes are deleted much less frequently than ordinary nodes, CA trees are less dependent on memory management.

The CBTree [1] is another recently proposed concurrent binary search tree data structure that like splay trees automatically reorganizes so that more frequently accessed keys are expected to have shorter search paths. As CA trees are agnostic to the sequential data structure component, they can be used together with splay trees and can thus also get their properties. In libraries that provide a CA tree implementation, the sequential data structure can even be a parameter which allows to optimize the CA tree for the workload at hand. For example, if the workload is update-heavy, it might be better to use Red–Black trees instead of AVL trees as the sequential data structure, since Red–Black trees provide slightly cheaper update operations at the cost of longer search paths than AVL trees.

A key difference between CA trees and recent work on concurrent ordered sets is that CA trees optimize their granularity of locking according to the workload at hand, which is often very difficult to predict during the design of an application. Thus, CA trees are able to spend less memory and time on synchronization when contention is low but are still able to adapt well on highly contended scenarios.

Ordered Sets with Range Operation Support. In principle, concurrent ordered sets with linearizable range operations can be implemented by utilizing software transactional memory (STM): the programmer simply wraps the operations in transactions and lets the STM take care of the concurrency control to ensure that the transactions execute atomically. Even though some scalable data structures have been derived by carefully limiting the size of transactions (e.g. [14, 29]), currently transactional memory does not seem to offer a general solution with good scalability; cf. [14].

Brown and Helga have extended the *non-blocking k-ary search tree* [12] to provide lock-free range queries [13]. A k-ary search tree is a search tree where all nodes, both internal and leaves, contain up to k keys. The internal nodes are utilized for searching, and leaf nodes contain all the elements. Range queries are performed in k-ary search trees with immutable leaf nodes by using a scan and a validation step. The scan step scans all leaves containing keys in the range and the validation step checks a dirty bit that is set before a leaf node is replaced by a modifying operation. Range queries are retried if the validation step fails. Unfortunately, non-blocking k-ary search trees provide no efficient way to perform atomic range updates or multi-key modification operations. Additionally, k-ary search trees are not balanced, so pathological inputs can easily make them perform poorly.

Robertson investigated the implementation of lock-free range queries in a skip list: range queries increment a version number and a fixed size history of changes is kept in every node [30]. However, this solution does not scale well because of the centralized version number counter. Also, it does not support range updates.

Functional data structures or copy-on-write is another approach to provide linearizable range queries. Unfortunately, this requires copying all nodes in a path to the root in a tree data structure which induces overhead and makes the root a contended hot spot.

The SnapTree data structure [2] provides a fast $\mathcal{O}(1)$ linearizable clone operation by letting subsequent write operations create a new version of the tree. Linearizable range queries can be performed in a SnapTree by first creating a clone and then performing the query in the clone. SnapTree's clone operation is performed by marking the root node as shared and letting subsequent update operations replace shared nodes and their children while traversing the tree. To ensure that no existing update operation can modify the clone, an epoch object is used. The clone operation forces new updates to wait for a new epoch object by closing the current epoch and then waits for existing modification operations (that have registered their ongoing operation in the epoch object) before a new epoch object is installed. The *Ctrie* data structure [31] also has a fast clone operation whose implementation and performance characteristics resembles SnapTree's [13].

Range operations can be implemented in data structures that utilize fine-grained locking by acquiring all necessary locks. For example, in a tree data structure where all elements reside in leaf nodes, the atomicity of the range operation can be ensured by locking all leaves in the range. This requires locking at least n/k nodes, if the number of elements in the range is n and at most k elements can be stored in every node. When n is large or when k is small the performance of this approach is limited by the locking overhead. On the other hand, when n is small or when k is large the scalability is limited by coarse-grained locking. In contrast, in CA trees k is dynamic and is automatically adapted at runtime to provide a good trade-off between scalability and locking overhead.

The Leaplist [14] is a concurrent ordered set implementation with native support for range operations. Leaplist is based on a skip list data structure with fat nodes that can contain up to k elements. The efficient implementation of the Leaplist uses transactional memory to acquire locks and to check if read data is valid. The authors of the Leaplist paper mention that they tried to derive a Leaplist version based purely on fine-grained locking but failed [14], so developing a Leaplist without dependence on STM seems to be difficult. As in trees with fine-grained locking, the size of the locked regions in Leaplists is fixed and does not adapt according to the contention as in CA trees. Furthermore, the performance of CA trees does not depend on the availability and performance of STM.

Operations that atomically operate on multiple keys can be implemented in any data structure by utilizing coarse-grained locking. By using a readers-writer lock, one can avoid acquiring an exclusive lock of the data structure for some operations. Unfortunately, locking the whole data structure is detrimental to scalability if the data structure is contended. The advantage of coarse-grained locking is that it provides the performance of the protected sequential data structure in the uncontended case. CA trees provide the high performance of coarse-grained locking in the uncontended cases and the scalability of fine-grained synchronization in contended ones by adapting their granularity of synchronization according to the contention level.

Other Related Work. In the context of distributed DBMS, Joshi [32] presented the idea of adapting locking in the ALG search tree data structure. ALG trees are however very different from CA trees. In ALG trees the tree structure itself does not adapt to contention, only its locking strategy does. Furthermore, ALG trees do not collect statistics about contention, but use a specialized distributed lock management system to detect contention and adapt the locking strategies.

Various forms of adaptation to the level of contention have previously been proposed for e.g. locks [33], diffracting trees [34, 35] and combining [36].

8. Evaluation

Let us now investigate the scalability of two CA tree variants: one with an AVL tree as sequential data structure (CA-AVL) and one with a skip list with fat nodes (CA-SL) as data structure. On workloads with range operations we compare CA trees against the lock-free k-ary search tree [13] (k-ary), the Snap tree [2] (Snap) and a lock-free skip list (SkipList). On workloads with only single-key operations we also compare CA trees against the balanced lock-free chromatic tree [8] (Chrom), the contention-friendly tree [3] (CFTree), and the logically ordered AVL tree [4] (LogAVL). We mark the data structures that do not support linearizable range queries with dashed lines to make it easier to spot them. All implementations are those provided by the authors. SkipList is implemented by Doug Lea in the Java Foundation Classes as the class `ConcurrentSkipListMap`.[3]

The SkipList, marked with dashed gray lines in the graphs, does not cater for linearizable range queries nor range updates. We include SkipList in the measurements for workloads with range operations only to show the kind of scalability one can expect from a lock-free skip list data structure if one is not concerned about consistency of results from range operations. Range operations are implemented in SkipList by calling the `subSet` method which returns an iterable view of the elements in the range. Since changes in SkipList are reflected in the view returned by `subSet` and vice versa, range operations are not atomic.

[3]We do not compare experimentally against the Leaplist [14] whose main implementation is in C. Prototype implementations of the Leaplist in Java were sent to us by its authors, but they end up in deadlocks when running our benchmarks which prevents us from obtaining reliable measurements. Instead, we refer to Section 7 for an analytic comparison to the Leaplist.

In contrast, the k-ary search tree supports linearizable range queries and the Snap tree supports linearizable range queries through the `clone` method. However, neither the k-ary nor the Snap tree provide support for linearizable range updates. In the scenarios where we measure range updates, we implement them in these data structures by using a frequent read optimized readers-writer (RW) lock[4] with a read indicator that has one dedicated cache line per thread. Thus, all operations except range updates acquire the RW lock in read mode. We have confirmed that this method has negligible overhead for all cases where range updates are not used, but use the implementations of the data structures without range update support in scenarios that do not have range updates.

We use $k = 32$ (maximum number of elements in nodes) both for the CA-SL and k-ary trees. This value provides a good trade-off between performance of range operations and performance of single-key modification operations. For the CA trees, we initialize the contention statistics counters of the locks to 0 and add 250 to the counter to indicate contention; we decrease the counter by 1 to indicate low contention. The thresholds -1000 and 1000 are used for low contention and high contention adaptations.

The benchmark we use measures throughput of a mix of operations performed by N threads on the same data structure during t seconds. The keys and values for the operations get, insert and remove as well as the starting key for range operations are randomly generated from a range of size R. The data structure is pre-filled before the start of each benchmark run by performing $R/2$ insert operations. In all experiments presented in this article $R = 1\,000\,000$, thus we create a data structure containing roughly $500\,000$ elements. In all captions, benchmark scenarios are described by a strings of the form `w`:$A\%$ `r`:$B\%$ `q`:$C\%$-R_1 `u`:$D\%$-R_2, meaning that on the created data structure the benchmark performs $(A/2)\%$ insert, $(A/2)\%$ remove, $B\%$ get operations, $C\%$ range queries of maximum range size R_1, and $D\%$ range updates with maximum range size R_2. The size of each range operation is randomly generated between one and the maximum range size. The benchmarks presented in this article were run on a machine with four AMD Opteron 6276 (2.3 GHz, 16 cores, 16M L2/16M L3 Cache), giving a total of 64 physical cores and 128 GB of RAM, running Debian 3.16.0-4-amd64 and Oracle Hotspot JVM 1.8.0_31 (started with parameters `-Xmx4g`, `-Xms4g`, `-server` and `-d64`).[5] For each data point, we ran ten measurements for 10 seconds each in separate JVM instances. To give the JIT compiler time to compile the code, a 10 seconds long warm up run was performed before each measurement run. We report the average throughput from the ten measurements as well as error bars showing the minimum and maximum throughput, though often the error bars are very small and therefore not visible in the graphs.

Workloads with Single-key Operations. Figure 11 shows selected workloads with single-key operations. In the top we have update-only (Fig. 11a) and read-only (Fig. 11b) scenarios and in the bottom we have the mixed workloads with 50% updates + 50%

[4]We use the write-preference algorithm [37] for coordination between readers and writers and the `StampedLock` from the Java library for mutual exclusion.

[5]We also ran experiments on a machine with four Intel(R) Xeon(R) E5-4650 CPUs (2.70GHz each with eight cores and hyperthreading) both on a NUMA setting and on a single chip, showing similar performance patterns as on the AMD machine. Results are available online [18].

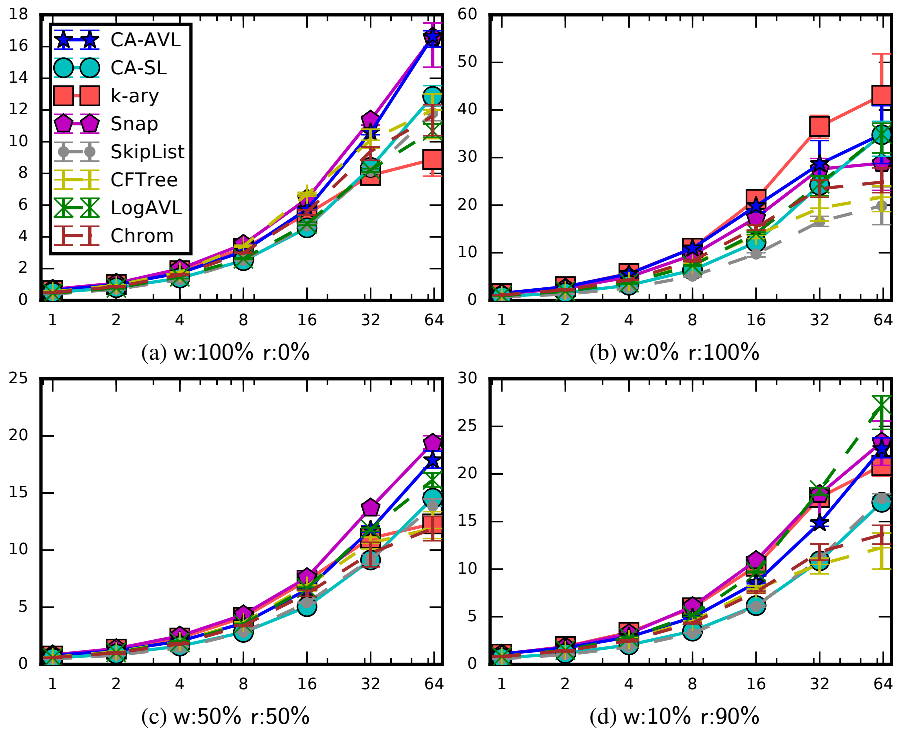
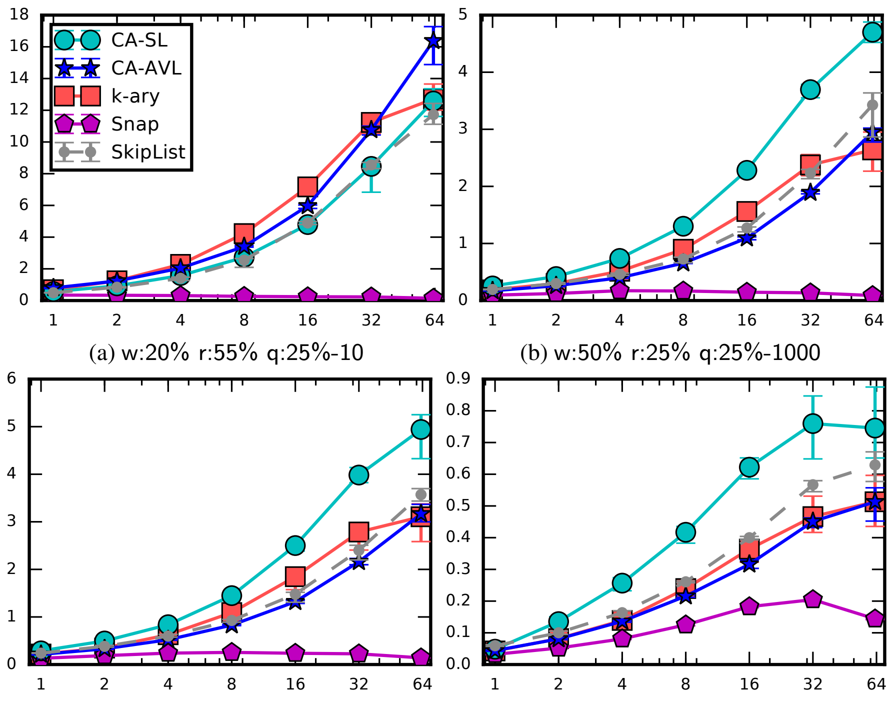


Figure 11: Throughput (ops/μs) on the y-axis and thread count on the x-axis. Scenarios are described by a strings of the form w:A% r:B%, meaning that the threads perform $(A/2)$% insert, $(A/2)$% remove and B% get operations.

reads (Fig. 11c) as well as with 10% updates + 90% reads (Fig. 11d). Even though the graphs are cluttered due to the many data structures that are included in the comparison, it is clear that the CA trees perform similar to and often better than many of the other data structures for concurrent ordered sets in a variety of scenarios with single-key operations. In the update heavy scenarios (Figs. 11a and 11c) the two data structures with best peak performance are CA-AVL and Snap. Only k-ary has a somewhat better peak performance than the CA trees in the read-only scenario (Fig. 11b). Finally, in the scenario with 90% reads + 10% writes (Fig. 11d) CA-AVL's peak performance is close to LogAVL which has the best peak performance.

Interestingly, k-ary has the worst peak performance in the update-only case and the best peak performance in the read-only case. k-ary has good cache locality due to the tree nodes that can store up to 32 keys (with our choice for k). However, the synchronization granularity in k-ary becomes less fine-grained for update operations with larger k values because updates in k-ary replace an old tree node with a new. The power of CA trees is that they avoid paying the overhead of fine-grained synchronization (i.e. memory overhead and performance overhead of synchronization instructions) when it is not needed, as in the read-only case, and they provide efficacy in scenarios when fine-grained synchronization is really beneficial, e.g., in the write-only scenario, by



(a) w:20% r:55% q:25%-10

(b) w:50% r:25% q:25%-1000

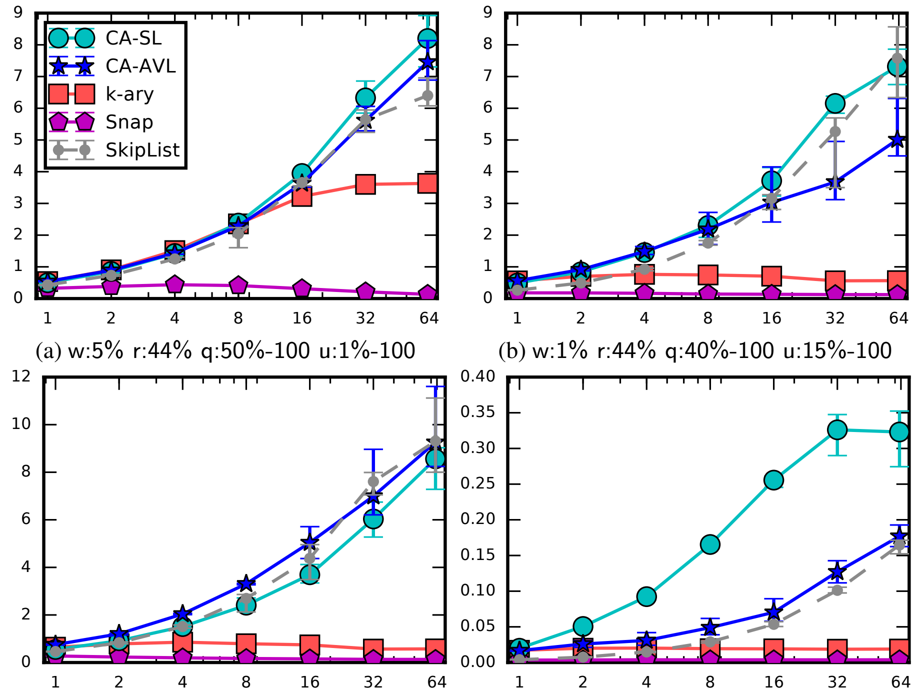
(c) w:20% r:55% q:25%-1000

(d) w:20% r:55% q:25%-10000

Figure 12: Throughput (ops/μs) on the y-axis and thread count on the x-axis. Scenarios are described by a strings of the form w:A% r:B% q:C%-R, meaning that the threads perform $(A/2)\%$ insert, $(A/2)\%$ remove, $B\%$ get operations, and $C\%$ range queries of maximum range size R.

adapting the synchronization granularity depending on the current contention level.

Workloads with Range Queries. Let us now discuss the performance results in Fig. 12, showing scenarios with range queries. Figure 12a, which shows throughput in a scenario with a moderate amount of modifications (20%) and small range queries, shows that the k-ary and CA-AVL trees perform best in this scenario, tightly followed by the CA-SL and SkipList with the non-atomic range queries. We also note that the Snap tree does not scale well in this scenario, which is not surprising since a range query with a small range size may eventually cause the creation of a copy of every node in the tree. Let us now look at Fig. 12b showing throughputs in a scenario with many modifications (50%) and larger range queries, and Fig. 12c corresponding to a scenario with the same maximum range query size and a more moderate modification rate (20%). First, note that the better cache locality for range queries in CA-SL and k-ary trees is visible in these scenarios where the range sizes are larger. k-ary only beats CA-AVL with a small amount up to 32 threads and then k-ary tree's performance drops. Note that a range query in the k-ary may need to be retried many times (possibly infinite) if an update operation interferes and sets one of the dirty bits between the first and second scans; cf. Section 7. This can be compared to the CA trees that acquire locks for reads if the first optimistic attempt



(a) w:5% r:44% q:50%-100 u:1%-100

(b) w:1% r:44% q:40%-100 u:15%-100

(c) w:3% r:27% q:50%-10 u:20%-10

(d) w:3% r:27% q:50%-10000 u:20%-10000

Figure 13: Throughput (ops/μs) on the y-axis and thread count on the x-axis. Scenarios are described by a strings of the form w:A% r:B% q:C%-R_1 u:D%-R_2, meaning that the threads perform $(A/2)$% insert, $(A/2)$% remove, B% get operations, C% range queries of maximum range size R_1, and D% range updates with maximum range size R_2.

fails, thus reducing the risk of retries. The scalability of the CA trees shown in Fig. 12b, i.e., in a scenario with 50% modification operations, shows that the range queries in the CA trees tolerate high contention. Finally, the scenario of Fig. 12d, with very wide range queries and moderate modification rate (20%), shows both the promise and the limit in the scalability of CA-SL. However, we note that SkipList, which does not even provide atomic range queries, does not beat CA-SL that outperforms the other data structures by at least 57% at 16 threads.

Workloads with Range Updates. We will now look at the scenarios that also contain range updates shown in Fig. 13. The first of them (Fig. 13a) shows that k-ary tree's scalability flattens out between 16 and 32 threads even with as little as 1% range updates. Instead, the CA trees provide good scalability all the way. Remember that we wrap the k-ary operations in critical sections protected by an RW-lock to provide linearizable range updates in the k-ary tree. In the scenario of Fig. 13b, where the percentage of range updates is 15%, we see that the k-ary tree does not scale at all while the CA trees and SkipList with the non-atomic range operations scale very well, outperforming the k-ary tree with more than 1200% in this case. The two scenarios in Figs. 13c and 13d

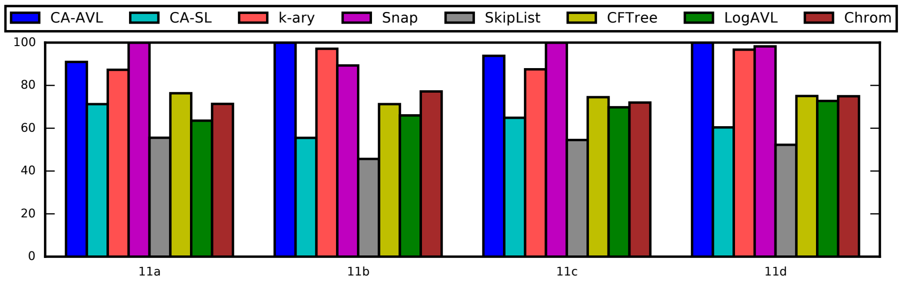


Figure 14: Bars showing the relative performance of the data structures in sequential scenarios. The labels on the x-axis denote subfigures (scenarios) in Fig. 11. The y-axis shows percentage of the best average throughput for the scenario, so higher is better.

have the same rate of operations but different maximum size for range queries and range updates. Their results clearly show the difference in performance characteristics that can be obtained by changing the sequential data structure component of a CA tree. CA-SL is faster for wider range operations due to its fat nodes providing good cache locality, but CA-SL is generally slower than the CA-AVL in scenarios with small range sizes. In Fig. 13d, where the conflict rate between operations is high, CA-SL reaches its peak performance at 32 threads where it outperforms all the other data structures by more than two times.

Sequential Performance. Although these are concurrent data structures, it is also interesting to compare them in terms of their sequential performance. Since this performance (i.e., when using only one thread) is not visible in Figs. 11 to 13, we show the relative sequential performance from these figures in Figs. 14 and 15, where the x-axis values refer to the corresponding subfigure.

In the scenarios with single-key operations (Fig. 14), k-ary, Snap and CA-AVL all perform better than 80% of the performance of the best performing data structure while the other data structures have performance worse than 80% of the best performing data structure. CFTree, which is optimized for heavily contended scenarios and uses a separate balancing thread, pays a big overhead in the sequential case due to the delayed balancing and the synchronization with the balancing thread. A LogAVL tree node does not only have a left and right pointer but also pointers to the nodes containing the closest keys that are greater than and smaller than the key of the node. These pointers are needed when searches in the tree are lead astray by concurrent rotations but cause overhead in the sequential case. Chrom, which has its imbalance parameter set to six (for improved performance in the concurrent cases), is less balanced than CA-AVL in the sequential case.

CA-AVL has the best sequential performance in the scenario with small range queries (Fig. 15, x-value 12a) and CA-SL has the best sequential performance with medium sized range queries (Fig. 15, x-values 12b and 12c). This is also the case for the corresponding contended scenarios (see Figs. 12a to 12c). More surprising is the scenario with large range queries (Fig. 15, x-value 12d). In this scenario, SkipList's

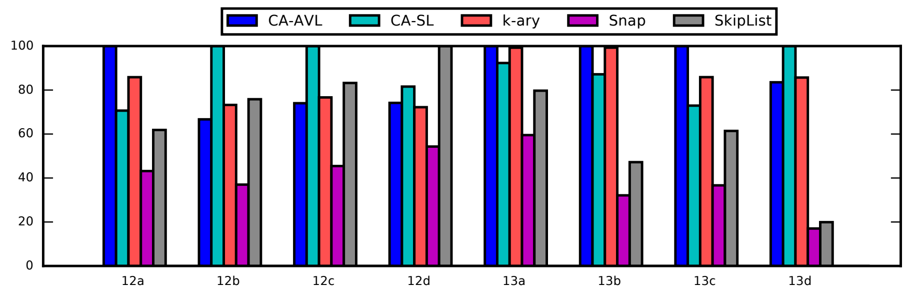


Figure 15: Bars showing the relative performance of the data structures in sequential scenarios. The labels on the x-axis denote subfigures (scenarios) in Figs. 12 and 13. The y-axis shows percentage of the best average throughput for the scenario, so higher is better.

sequential performance is better than CA-SL's despite CA-SL's better cache locality due to the fat skip list nodes and the better performance of CA-SL under contention (see Fig. 12d). The surprisingly good results for SkipList can be explained by an overhead in CA-SL's range queries that is imposed by the optimistic attempt. The range query operation takes a function object that will be applied to all keys in the range given as parameter. As SkipList's range query is not linearizable, this function can be applied on-the-fly while traversing the range in the skip list. However, CA-SL's optimistic range query attempts (that always succeed in the sequential scenario) have to first store all involved keys in an intermediate storage so that the sequence number in the base node lock can be validated before the function is applied on the keys. CA-SL's better cache locality also becomes less of an advantage in the sequential case when a large part of the data structure will be available in a fast processor cache close to the core that is performing the operations.

In the scenarios that also contain range updates (Fig. 15, x-values 13a–13d), k-ary and CA-AVL perform best with range operations of size 100 (x-values 13a and 13b), CA-AVL performs best in the scenario with range operations of size 10 (x-value 13c), and CA-SL performs best in the scenario with large range operations (x-value 13d).

To conclude, the CA trees have good sequential performance across all measured scenarios. This can be explained by CA trees' adaptiveness which allow them to perform essentially as their sequential component in the sequential case. Furthermore, the memory footprint of CA trees in the sequential case is also essentially that of its sequential data structure component in contrast to the other data structures that all need flags, pointers or locks in all their nodes to ensure correctness. Remember that a CA tree accessed sequentially will eventually just consist of one base node.

Adaption to Contention and Access Pattern. Finally, we report average base node counts for the CA trees in the end of running two sets of scenarios. The numbers in Fig. 16a show node counts (in k) for running with 64 threads but varying the maximum range size R. Figure 16b shows node counts (also in k) for scenarios with R fixed to 1000 but varying the number of threads. These numbers confirm that the CA trees' synchronization is adapting both to the contention level (increasing the number of

R	10	100	1000	10000
CA-SL	14.4	8.8	4.0	2.5
CA-AVL	15.6	8.7	3.6	2.2

(a) w:3% r:27% q:50%-R u:20%-R

threads	2	4	8	16	32	64
CA-SL	0.36	0.73	1.2	1.9	2.7	4.0
CA-AVL	0.34	0.68	1.1	1.6	2.4	3.6

(b) w:3% r:27% q:50%-1000 u:20%-1000

Figure 16: Average base node counts (in k) at the end of running two sets of benchmarks: one using 64 threads but varying the range size R, and one varying the number of threads.

threads results in more base nodes) and to the access patterns (increased range size results in fewer base nodes). We also confirmed by increasing the running time of the experiments from ten to twenty seconds that the number of base nodes in the data structure seems to have stabilized around a specific value after ten seconds, which means that base nodes do not get split indefinitely.

9. Concluding Remarks

Given the diversity in sizes and heterogeneity of multicores, it seems rather obvious that current and future applications will benefit from, if not require, data structures that can adapt dynamically to the amount of concurrency and the usage patterns of applications. This article has advocated the use of CA trees, a new family of lock-based concurrent data structures for ordered sets of keys and key-value pair dictionaries. CA trees' salient feature is their ability to automatically adapt their synchronization granularity to the current contention level and applications' access patterns. Furthermore, CA trees are flexible and efficiently support a wide variety of operations: single-key operations, multi-key operations, range queries and range updates. Their flexibility makes it easy to modify them to suit different use cases. As an example, we have used a CA tree as a core component in the implementation of an efficient concurrent priority queue [38]. Our experimental evaluation in this article as well as in the cited publication has demonstrated the good scalability and superior performance of CA trees compared to state-of-the-art lock-free concurrent data structures in a variety of scenarios.

As future work, we are planning to design CA tree inspired data structures with lock-free or wait-free progress guarantees and investigate their scalability and performance.

References

[1] Y. Afek, H. Kaplan, B. Korenfeld, A. Morrison, R. E. Tarjan, The CBTree: A practical concurrent self-adjusting search tree, Distributed Computing 27 (6) (2014) 393–417. doi:10.1007/s00446-014-0229-0.
URL http://dx.doi.org/10.1007/s00446-014-0229-0

[2] N. G. Bronson, J. Casper, H. Chafi, K. Olukotun, A practical concurrent binary search tree, in: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '10, ACM, New York, NY, USA, 2010, pp. 257–268. doi:10.1145/1693453.1693488.
URL http://dx.doi.org/10.1145/1693453.1693488

[3] T. Crain, V. Gramoli, M. Raynal, A contention-friendly binary search tree, in: Euro-Par 2013 Parallel Processing - 9th International Conference, Vol. 8097 of LNCS, Springer, 2013, pp. 229–240.
URL http://dx.doi.org/10.1007/978-3-642-40047-6_25

[4] D. Drachsler, M. Vechev, E. Yahav, Practical concurrent binary search trees via logical ordering, in: Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, ACM, New York, NY, USA, 2014, pp. 343–356. doi:10.1145/2555243.2555269.
URL http://doi.acm.org/10.1145/2555243.2555269

[5] F. Ellen, P. Fatourou, E. Ruppert, F. van Breugel, Non-blocking binary search trees, in: Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, PODC '10, ACM, New York, NY, USA, 2010, pp. 131–140. doi:10.1145/1835698.1835736.
URL http://doi.acm.org/10.1145/1835698.1835736

[6] A. Natarajan, N. Mittal, Fast concurrent lock-free binary search trees, in: Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, ACM, New York, NY, USA, 2014, pp. 317–328. doi:10.1145/2555243.2555256.
URL http://doi.acm.org/10.1145/2555243.2555256

[7] B. Chatterjee, N. Nguyen, P. Tsigas, Efficient lock-free binary search trees, in: Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing, PODC '14, ACM, New York, NY, USA, 2014, pp. 322–331. doi:10.1145/2611462.2611500.
URL http://doi.acm.org/10.1145/2611462.2611500

[8] T. Brown, F. Ellen, E. Ruppert, A general technique for non-blocking trees, in: Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, ACM, New York, NY, USA, 2014, pp. 329–342. doi:10.1145/2555243.2555267.
URL http://doi.acm.org/10.1145/2555243.2555267

[9] A. Natarajan, L. H. Savoie, N. Mittal, Concurrent wait-free red black trees, in: T. Higashino, Y. Katayama, T. Masuzawa, M. Potop-Butucaru, M. Yamashita (Eds.), Stabilization, Safety, and Security of Distributed Systems: 15th International Symposium, SSS 2013, Vol. 8255 of LNCS, Springer, 2013, pp. 45–60. doi:10.1007/978-3-319-03089-0_4.
URL http://dx.doi.org/10.1007/978-3-319-03089-0_4

[10] S. V. Howley, J. Jones, A non-blocking internal binary search tree, in: Proceedings of the 24th Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '12, ACM, New York, NY, USA, 2012, pp. 161–171. doi:10.1145/2312005.2312036.
URL http://doi.acm.org/10.1145/2312005.2312036

[11] K. Fraser, Practical lock-freedom, Ph.D. thesis, University of Cambridge Computer Laboratory (2004).
URL https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-579.pdf

[12] T. Brown, J. Helga, Non-blocking k-ary search trees, in: A. Fernàndez Anta, G. Lipari, M. Roy (Eds.), Principles of Distributed Systems: 15th International Conference, OPODIS 2011. Proceedings, Springer, 2011, pp. 207–221. doi:10.1007/978-3-642-25873-2_15.
URL http://dx.doi.org/10.1007/978-3-642-25873-2_15

[13] T. Brown, H. Avni, Range queries in non-blocking k-ary search trees, in: R. Baldoni, P. Flocchini, R. Binoy (Eds.), Principles of Distributed Systems: 16th International Conference, OPODIS 2012. Proceedings, Springer, 2012, pp. 31–45. doi:10.1007/978-3-642-35476-2_3.
URL https://doi.org/10.1007/978-3-642-35476-2_3

[14] H. Avni, N. Shavit, A. Suissa, Leaplist: Lessons learned in designing TM-supported range queries, in: Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing, PODC '13, ACM, New York, NY, USA, 2013, pp. 299–308. doi:10.1145/2484239.2484254.
URL http://doi.acm.org/10.1145/2484239.2484254

[15] K. Sagonas, K. Winblad, Contention adapting search trees, in: 14th International Symposium on Parallel and Distributed Computing, ISPDC, IEEE, 2015, pp. 215–224. doi:10.1109/ISPDC.2015.32.
URL http://dx.doi.org/10.1109/ISPDC.2015.32

[16] K. Sagonas, K. Winblad, Efficient support for range queries and range updates using contention adapting search trees, in: X. Shen, F. Mueller, J. Tuck (Eds.), Languages and Compilers for Parallel Computing - 28th International Workshop, LCPC, Vol. 9519 of LNCS, Springer, 2016, pp. 37–53. doi:10.1007/978-3-319-29778-1_3.
URL http://dx.doi.org/10.1007/978-3-319-29778-1_3

[17] W. Visser, K. Havelund, G. Brat, S. Park, F. Lerda, Model checking programs, Automated Software Engineering 10 (2) 203–232. doi:10.1023/A:1022920129859.
URL http://dx.doi.org/10.1023/A:1022920129859

[18] CA Trees, http://www.it.uu.se/research/group/languages/software/ca_tree.

[19] G. Adelson-Velskii, E. M. Landis, An algorithm for the organization of information, in: Proceedings of the USSR Academy of Sciences, Vol. 146, 1962, pp. 263–266.

[20] R. Bayer, Symmetric binary B-trees: Data structure and maintenance algorithms, Acta Informatica 1 (4) (1972) 290–306. doi:10.1007/BF00289509.
URL http://dx.doi.org/10.1007/BF00289509

[21] D. E. Knuth, The Art of Computer Programming: Sorting and Searching, vol. 3, 2nd Edition, Addison-Wesley, Reading, 1998.

[22] R. E. Tarjan, Data Structures and Network Algorithms, Vol. 14, SIAM, 1983.

[23] R. Seidel, C. R. Aragon, Randomized search trees, Algorithmica 16 (4–5) (1996) 464–497. doi:10.1007/BF01940876.
URL http://dx.doi.org/10.1007/BF01940876

[24] C. Lameter, Effective synchronization on Linux/NUMA systems, in: Proc. of the Gelato Federation Meeting, 2005.
URL http://www.kde.ps.pl/mirrors/ftp.kernel.org/linux/kernel/people/christoph/gelato/gelato2005-paper.pdf

[25] H.-J. Boehm, Can seqlocks get along with programming language memory models?, in: Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness, MSPC '12, ACM, New York, NY, USA, 2012, pp. 12–20. doi:10.1145/2247684.2247688.
URL http://doi.acm.org/10.1145/2247684.2247688

[26] P. Hammarlund, A. J. Martinez, A. A. Bajwa, D. L. Hill, E. Hallnor, H. Jiang, M. Dixon, M. Derr, M. Hunsaker, R. Kumar, R. B. Osborne, R. Rajwar, R. Singhal, R. D'Sa, R. Chappell, S. Kaushik, S. Chennupaty, S. Jourdan, S. Gunther, T. Piazza, T. Burton, Haswell: The fourth-generation Intel core processor, IEEE Micro 34 (2) (2014) 6–20. doi:10.1109/MM.2014.10.
URL http://doi.ieeecomputersociety.org/10.1109/MM.2014.10

[27] R. Rajwar, J. R. Goodman, Speculative lock elision: Enabling highly concurrent multithreaded execution, in: Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture, MICRO 34, IEEE Computer Society, Washington, DC, USA, 2001, pp. 294–305.
URL http://dl.acm.org/citation.cfm?id=563998.564036

[28] L. Bougé, J. Gabarró, X. Messeguer, N. Schabanel, Height-relaxed AVL rebalancing: A unified, fine-grained approach to concurrent dictionaries, Tech. Rep. RR1998-18, LIP, ENS Lyon (Mar. 1998).

[29] T. Crain, V. Gramoli, M. Raynal, A speculation-friendly binary search tree, in: Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '12, ACM, New York, NY, USA, 2012, pp. 161–170. doi:10.1145/2145816.2145837.
URL http://doi.acm.org/10.1145/2145816.2145837

[30] C. Robertson, Implementing contention-friendly range queries in non-blocking key-value stores, Bachelor thesis, The University of Sydney (Nov. 2014).

[31] A. Prokopec, N. G. Bronson, P. Bagwell, M. Odersky, Concurrent tries with efficient non-blocking snapshots, in: Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '12, ACM, New York, NY, USA, 2012, pp. 151–160. doi:10.1145/2145816.2145836.
URL http://doi.acm.org/10.1145/2145816.2145836

[32] A. M. Joshi, Adaptive locking strategies in a multi-node data sharing environment, in: Proceedings of the 17th International Conference on Very Large Databases, Morgan Kaufmann, 1991, pp. 181–191.

[33] B.-H. Lim, A. Agarwal, Reactive synchronization algorithms for multiprocessors, in: Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS VI, ACM, New York, NY, USA, 1994, pp. 25–35. doi:10.1145/195473.195490.
URL http://doi.acm.org/10.1145/195473.195490

[34] G. Della-Libera, N. Shavit, Reactive diffracting trees, in: Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '97, ACM, New York, NY, USA, 1997, pp. 24–32. doi:10.1145/258492.258495.
URL http://doi.acm.org/10.1145/258492.258495

[35] P. H. Ha, M. Papatriantafilou, P. Tsigas, Self-tuning reactive diffracting trees, Journal of Parallel and Distributed Computing 67 (6) (2007) 674–694. doi:10.1016/j.jpdc.2007.01.011.
URL http://www.sciencedirect.com/science/article/pii/S0743731507000184

[36] N. Shavit, A. Zemach, Combining funnels: a dynamic approach to software combining, Journal of Parallel and Distributed Computing 60 (11) (2000) 1355–1387. doi:10.1006/jpdc.2000.1621.
URL http://www.sciencedirect.com/science/article/pii/S0743731500916216

[37] I. Calciu, D. Dice, Y. Lev, V. Luchangco, V. J. Marathe, N. Shavit, NUMA-aware reader-writer locks, in: Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ACM, New York, NY, USA, 2013, pp. 157–166. doi:10.1145/2442516.2442532.
URL http://doi.acm.org/10.1145/2442516.2442532

[38] K. Sagonas, K. Winblad, The contention avoiding concurrent priority queue, in: Languages and Compilers for Parallel Computing: 29th International Workshop, LCPC 2016, Rochester, NY, USA, September 28-30, 2016, Revised Papers, Springer International Publishing, 2017, pp. 314–330. doi:10.1007/978-3-319-52709-3_23.
URL https://doi.org/10.1007/978-3-319-52709-3_23