Contention Adapting Search Trees

Konstantinos Sagonas*† and Kjell Winblad*
* Department of Information Technology, Uppsala University, Uppsala, Sweden
† School of ECE, National Technical University of Athens, Greece

***Abstract*—With multicores being ubiquitous, concurrent data structures are becoming increasingly important. This paper proposes a novel approach to concurrent data structure design where the data structure collects statistics about contention and adapts dynamically according to this statistics. We use this approach to create a contention adapting binary search tree (CA tree) that can be used to implement concurrent ordered sets and maps. Our experimental evaluation shows that CA trees scale similar to recently proposed algorithms on a big multicore machine on various scenarios with a larger set size, and outperform the same data structures in more contended scenarios and in sequential performance. We also show that CA trees are well suited for optimization with hardware lock elision. In short, we propose a practically useful and easy to implement and show correct concurrent search tree that naturally adapts to the level of contention.**

I. Introduction

With multicores being widespread, the need for efficient concurrent data structures has increased. In this paper we propose a novel adaptive technique for creating concurrent data structures. Our technique collects statistics about contention in locks and does local adaptations dynamically to reduce the contention or to optimize for low contention. This is the first contribution of this paper. Previous research on adapting to the level of contention has focused on objects where access cannot be easily distibuted, such as locks [24], shared counters [13], [17], stacks and queues [32]. In contrast to these works, our work targets data structures where the access patterns differ across parts of the data structure and/or often change during the lifetime of the program. A concrete example is an ordered map where the keys are timestamps and new timestamps are inserted by several threads concurrently while old timestamps are only accessed rarely. In this scenario, the part of the data structure containing the most recent timestamps will be highly contended while the rest of the data structure will be accessed under low contention. In other scenarios, the amount of concurrent accesses can vary a lot depending on input, the machine the program is running on and its load, the program's phase, etc.

We demonstrate the benefits of our contention adapting technique by describing and evaluating a data structure for concurrent ordered sets or maps. We call this data structure contention adapting search tree or CA tree for short. The design of CA trees is the second contribution of this paper.

Current scalable data structures for concurrent ordered sets and maps either use fine-grained locking [2], [6], [12], [14] or lock-free techniques [7], [10], [15], [16], [20], [26], [27] to enable parallel operations in the tree. For fine-grained synchronization, they all pay a price in sequential efficiency and/or memory usage. Futhermore, they typically do not support operations that atomically operate on a number of elements. The CA tree differs from them in that it dynamically optimizes its synchronization granularity according to the current access patterns. This way, the CA tree only needs to pay the cost of fine-grained synchronization in performance and memory footprint when such synchronization is actually beneficial.

Another nice property of CA trees is that they use a sequential ordered set or map data structure as an exchangeable component. The sequential data structure component can be, for example, an AVL tree [1], a Red-Black tree [4], a Splay tree [33], or a Skip list [28]. Hence, the resulting CA tree inherits its performance characteristics from the sequential component that it uses. We are not aware of any other efficient concurrent search tree that provides this kind of flexibility.

The next section presents a high level view of CA trees, followed by the algorithm (Sect. III) and its properties (Sect. IV). We then describe optimizations (Sect. V), review related work (Sect. VI), experimentally compare CA trees against related data structures and evaluate their performance (Sect. VII). The paper ends with some concluding remarks (Sect. VIII).

II. A Brief Overview of CA Trees

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 containing the actual keys of the items stored. All keys stored under the left pointer of a routing node are less than the routing node's key and all keys stored under the right pointer are greater or equal to the 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 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 the base 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 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

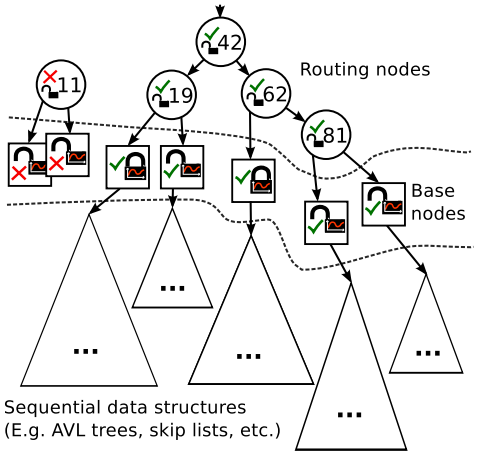

Fig. 1. The structure of a CA tree. Numbers denote keys, a node whose flag is valid is marked with a green hook; an invalid one with a red cross.

contention is higher than a certain threshold, the base node is split by dividing the sequential data structure into two new base nodes and linking them into the tree with the help of a new routing node. In the other direction, if the statistics in some base node A indicate that the contention is lower than a threshold, then A is joined with a neighbor base node B by creating a new base node containing the keys from both A and B to replace B and deleting the parent routing node of A.

III. Implementation

This section describes the implementation of CA trees and gives arguments for the algorithm's correctness. 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 to implement a CA tree.

A. Statistics Collecting Locks

We use a standard mutual exclusion (mutex) lock and an integer variable to create a statistics collecting lock. Java-like code for such locks is shown in Fig. 2. The statistics variable is incremented or decremented after the lock has been taken. If the `statTryLock` call succeeds, no contention is detected and the statistics variable is decremented with `SUCC_CONTRIB`. On the other hand, if the `statTryLock` failed, another thread was holding the lock so the statistics is incremented by `FAIL_CONTRIB`. The code uses the function `statTryLock` that just forwards its calls to the underlying mutex lock and is omitted for brevity.

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

Fig. 2. Statistics collecting lock.

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, it 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 urgent 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 always 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 variable 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.

B. Ordered Sets with Split and Join Support

The literature contains a variety of data structures suitable for implementing ordered sets. All of them are efficient for the common set operations: INSERT, DELETE and LOOKUP. They either provide amortized, expected or guaranteed $\mathcal{O}(log(N))$ complexity for these operations as well as ordered traversal of the set in time linear in the size of the set. 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 into two so that the maximum key in one of the sets is smaller than the minimum key in the other. This operation can be implemented in 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.

Formally, 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 data structure contains the union of the keys of the two input data structures. AVL trees and red-black trees support this operation in guaranteed $\mathcal{O}(log(N))$ time. In this paper we experiment with AVL trees [1] as the sequential ordered set data structure. A description of the JOIN operation for AVL trees can be found in Knuth's book [22, page 474].

C. Implementing Contention Adapting Trees

Figure 3 shows the main algorithm for all operations in a CA tree. Since the algorithm is generic it can be used for all common ordered set operations (e.g. INSERT, LOOKUP, 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 4 to 8 search the routing layer from the root of the tree until the search ends up in a base node. (ii) Lines 10 to 13 lock the statistics lock in the base node and check the valid flag. The operation has to be restarted if the valid flag is false. In that case the lock is unlocked and the operation is restarted. (iii) Line 15 executes the operation

```
1  Object doOperation(CATree tree, Op operation, Key key) {
2    RoutingNode prevNode = null;
3    Object currNode = tree.root;
4    while (currNode isInstanceOf RoutingNode) {
5      prevNode = currNode;
6      if (key < currNode.key) currNode = currNode.left;
7      else currNode = currNode.right;
8    }
9    BaseNode base = currNode asInstanceOf BaseNode;
10   statLock(base.lock);
11   if (base.valid == false) {
12     statUnlock(base.lock);
13     return doOperation(tree, operation, key); // retry
14   } else {
15     Object result = operation.execute(base.root, key);
16     if (base.lock.statistics > MAX_CONTENTION) {
17       if (size(base.root) < 2) base.lock.statistics = 0;
18       else highContentionSplit(tree, base, prevNode);
19     } else if (base.lock.statistics < MIN_CONTENTION) {
20       if (prevNode == null) base.lock.statistics = 0;
21       else lowContentionJoin(tree, base, prevNode);
22     }
23     statUnlock(base.lock);
24     return result;
25   }
26 }
```

Fig. 3. The CA tree algorithm.

on the sequential ordered set data structure inside the base node. (iv) Lines 16 to 22 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 23 and 24 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.

D. 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 nodes of the original tree. 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 4 contains the code. `pickSplitKey` (line 3) picks a key that ideally divides the sequential ordered set data structure in half. If this data structure is a tree, a convenient way of doing this is to select the key of the root node. The statement on line 4 splits the data structure according to the split key.

The new routing node can be linked in at the place of the old base nodes 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 stays the same during the entire lifetime of the node. It is easy to see that `highContentionSplit` preserves this invariant. The next section will make it clear that `lowContentionJoin` also preserves the same invariant since it does not delete the parent of a base node that is valid.

Notice that it is important to mark the old base as invalid before unlocking its lock as is done in line 10. It is safe to replace the old base node since threads that have ended up in the old base node will retry their operation when they see that the old base node is invalid.

```
1  void highContentionSplit(CATree tree, BaseNode base,
2                           RoutingNode parent) {
3    Key splitKey = pickSplitKey(base.root);
4    Tuple<Tree> split = splitTree(splitKey, base.root);
5    RoutingNode newRoute =
6      RouteNode(BaseNode(split.elem1), splitKey, BaseNode(split.elem2));
7    if (parent == null) tree.root = newRoute;
8    else if (parent.left == base) parent.left = newRoute;
9         else parent.right = newRoute;
10   base.valid = false;
11 }
```

Fig. 4. High-contention adaptation.

E. Low-contention Adaptation

Figure 5 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 the 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.

We first note that, as discussed in the previous section, if we find a base node `base` from a parent routing node `parent`, the base node is guaranteed to not get a new parent as a result of a concurrent change from another thread. This will be true as long as the base node is locked and marked valid (which it is when entering `lowContentionJoin`).

In line 4 we find the leftmost base node of the `parent`'s right branch. We try to lock this `neighborBase` in line 5. If we fail to lock it or if `neighborBase` is invalid (line 7 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 we instead of the `statTryLock` call had used a forcing lock call, one could arrive in a deadlock situation because our `base` could be another thread's `neighborBase` and vice versa. In line 11 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 can not be any keys between `base` and `neighborBase` as long as we are holding the locks of `base` and `neighborBase`.

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 parent or the grandparent we require that the lock of both parent and grandparent (if the grandparent is not the root pointer) are acquired before we do the splicing. After acquiring the grandparent's lock, we also need to ensure that the grandparent has not been spliced out

```
1  void lowContentionJoin(CATree tree, BaseNode base,
2                         RoutingNode parent) {
3    if (parent.left == base) {
4      BaseNode neighborBase = leftmostBaseNode(parent.right);
5      if (!statTryLock(neighborBase.lock)) {
6        base.lock.statistics = 0;
7      } else if (!neighborBase.valid) {
8        statUnlock(neighborBase.lock);
9        base.lock.statistics = 0;
10     } else {
11       lock(parent.lock);
12       parent.valid = false;
13       neighborBase.valid = false;
14       base.valid = false;
15       RoutingNode gparent = null; // gparent = grandparent
16       do {
17         if (gparent != null) unlock(gparent.lock);
18         gparent = parentOf(parent, tree);
19         if (gparent != null) lock(gparent.lock);
20       } while (gparent != null && !gparent.valid);
21       if (gparent == null) {
22         tree.root = parent.right;
23       } else if (gparent.left == parent) {
24         gparent.left = parent.right;
25       } else {
26         gparent.right = parent.right;
27       }
28       unlock(parent.lock);
29       if (gparent != null) unlock(gparent.lock);
30       BaseNode newNeighborBase =
31         BaseNode(joinTrees(base.root, neighborBase.root));
32       RoutingNode neighborBaseParent = null;
33       if(parent.right == neighborBase) neighborBaseParent = gparent;
34       else neighborBaseParent = leftmostRouteNode(parent.right);
35       if(neighborBaseParent == null) {
36         tree.root = newNeighborBase;
37       } else if (neighborBaseParent.left == neighborBase) {
38         neighborBaseParent.left = newNeighborBase;
39       } else {
40         neighborBaseParent.right = newNeighborBase;
41       }
42       statUnlock(neighborBase.lock);
43     }
44   } else { ... } /* This case is symmetric to the previous one */
45 }
```

Fig. 5. Low-contention adaptation.

from the tree by checking its valid flag. Acquiring the lock of the parent (line 11) is straightforward since we know that it is still our parent as we argued before. Acquiring the lock of the grandparent (lines 15–20) 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. If the grandparent is the root pointer, we can 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 can not. Now we can splice out the parent (lines 21–27) and unlock the routing node lock(s) that we have taken (lines 28–29).

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 30–41). Notice that it is important that we mark `neighborBase` and `base` invalid (lines 13–14) before we unlock them to make waiting threads retry their operations. Notice also that the parent of `neighborBase` might have been changed by lines 21 to 27 so it would not have been safe to use the parent of `neighborBase` at the time of executing line 4.

F. Atomically Accessing Multiple Elements

The algorithm presented in Fig. 3 can be used for operations such as INSERT, DELETE and LOOKUP that atomically operate on a single element. For many applications it is also important to provide support for operations that atomically operate on multiple of elements. We describe how to efficiently perform atomic BULK INSERT, BULK DELETE as well as range queries and updates in a companion document [31].

IV. Properties

This section presents the properties that CA trees provide: deadlock freedom, livelock freedom, and linearizability. We then discuss the time complexity of CA tree operations.

A. Deadlock and Livelock Freedom

We will show that the CA tree algorithm is deadlock free by showing that all operations either obtain locks in a specific order so 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. Operations that call `lowContentionJoin` can use `tryLock` (Fig. 5, line 5). If the `tryLock` is unsuccessful, `lowContentionJoin` will return and the currently held lock will be released (Fig. 3, line 23). 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, so routing node locks are 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.)

A livelock occurs when threads perform some actions that interfere with each other so that none of them 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 if an invalid base node is seen. The interfering thread must have completed an operation in this case. Otherwise no split or join could have happened. (ii) If the code in Fig. 5 (lines 15–20) needs to be retried to find the grandparent of a base node, another thread must have spliced out a routing node and has thus made progress. The CA tree operations are therefore livelock free.

B. Linearizability

To show that a CA tree operation is linearizable [19], we have to show that the operation appears to happen instantaneously at some point during its execution and gives the same result as the corresponding operation in a sequential ordered set. Let us consider the algorithm in Fig. 3. First, we will show that if we perform a search in the routing layer that ends up on line 15 in Fig. 3, `base` will point to the base node that must contain the key we searched for, if it exists in the ordered set. This property that we call P must hold if CA tree operations correspond to sequential ordered set operations, since we could otherwise miss keys. P holds trivially after a CA tree has been initialized since the CA tree then has only one base node

that the tree's root directly points to. A high-contention split changes a pointer in the routing layer from pointing to the old split base node B to a new routing node R (Fig. 4, lines 7–9). As argued in Section III-D, no thread can interfere with this change. A search will still end up in the base node responsible for the key searched for after the change since, by definition of `splitTree`, all keys less than R's key will be in the left branch of R and the rest of the keys will be in its right branch. Let us assume that a search S_1 happens concurrently with the split and misses the split's pointer update so it ends up in B. Since the split invalidates B (Fig. 4, line 10) before B is unlocked (Fig. 3, line 23), S_1 must see that B is invalid and therefore will retry the search (Fig. 3, line 13). When S_1 is retried it cannot end up in B again since the split removes B from the routing layer before it unlocks B. We will now argue that P also holds after a low-contention join when the variable `base` given as parameter to the join function (Fig. 5, line 1) is the left child of its parent routing node. The case when `base` is the right child of its parent is symmetric. P trivially holds if `neighborBase` is not successfully locked (line 5) or if `neighborBase` is invalid (line 7) since the join will not modify the CA tree at all in these cases. The fist change that the join can do to the routing layer of the CA tree is to splice out the parent of `base` (lines 21–27). No other operation can interfere with this change since we are holding the lock of both the parent and grandparent routing nodes; see Section III-E. Let S_2 be a search for a key in `base` that takes place directly after this step. The parent of `base` is replaced with the subtree T located at the right branch of the parent of `base`. The search S_2 must therefore go to the leftmost base node in T, which is `neighborBase` just after the join's first change to the routing layer. Note that as we argue in Section III-E `neighborBase` must still be the leftmost base node of T just after we have spliced out `base`. The next step of the join is to replace `neighborBase` with a new base node containing the keys of both `base` and `neighborBase` (lines 35–41). Searches that happen concurrently with the join and end up in `base` or `neighborBase` will retry the search when the join's updates are visible, which can be demonstrated using a similar argument as for split. Thus, property P holds since split and join are the only functions that modify the routing layer and P holds initially as well as before, during and after splits and joins.

The code from line 15 to 23 in Fig. 3 is protected by the base node lock and thus happens in isolation from other threads. The modification or reading of the data structure happens between these lines (line 15). The linearization point can be set to any point between line 15 to 23 since the operation can appear to happen at this point from the perspective of all other operations.

C. Balancing and Time Complexity

When a CA tree is accessed sequentially, the execution time of its operations will depend on the sequential data structure that is used, the current depth of the routing layer and the time for low-contention adaptation. Therefore, the sequential execution time of a CA tree operation *op* is $\mathcal{O}(log(N) + D)$, given that the maximum depth of the routing layer is D, the number of keys is N and that the JOIN operation, the SPLIT operation and *op* all take $\mathcal{O}(log(N))$ time in the sequential data structure. One can easily change the CA tree algorithm to provide better theoretical sequential execution time guarantees by avoiding base node splits that could lead to a search path greater than a constant. However, our experience does not indicate that limiting the search path in the routing layer in such a way is necessary in practice.

Splitting and joining of base nodes can be seen as a dynamic optimization of the underlying sequential tree for parallel execution that only occurs when the statistics indicate that such an optimization would improve the execution time. Thanks to the efficient $\mathcal{O}(log(N))$ time JOIN and SPLIT operations that are supported by several balanced search trees, this optimization can give very good performance as we will show in Section VII.

V. Optimizations

A. Optimizations for Read-only Operations

A possible performance concern for CA trees in read-heavy scenarios is that even read-only operations acquire a lock. On a multicore machine this can become a performance and scalability bottleneck since acquiring a lock causes some shared memory to be written. This memory will be invalidated in the private caches of other cores which may cause future cache misses and expensive cache coherence traffic. Many concurrent search trees (both lock-free and lock-based) gain performance by making read-only operations traverse the tree without writing to shared memory [6], [12], [14]. Thus, we present optimizations that can make CA trees perform better in read-heavy scenarios. Their effect is evaluated in Section VII.

Optimistic readers with sequence locks: A sequence lock is implemented with one integer counter that is initialized to an even number [23]. A thread acquires a sequence lock by first waiting until the counter has an even number and then tries to increment it by one with an atomic compare-and-swap (CAS) instruction. To unlock the sequence lock, the integer is just increased by one to an even number. Threads doing a read-only critical section can do an invisible optimistic attempt by checking that the counter has an even number before the critical section and then checking that it is still the same even number after the critical section. A thread will fail the optimistic read attempt if it detects that a writer has interfered. Sequence locks are badly suited for some complex critical sections where an intermediate state produced by a writer could lead to a crash or an infinite loop in a reader. However, LOOKUP operations of a search tree can safely be made optimistic since the intermediate state produced by update operations can in the worst case make the search end up in the wrong node. Since this will be detected when validating the sequence number after the LOOKUP, it is not a correctness problem. We have implemented a sequence lock optimization and evaluate it in Section VII-A. In our implementation the SC locks in the base nodes use a sequence lock as mutex. LOOKUPS optimistically try to do an invisible read and if that fails on the first attempt the operation proceeds by acquiring the statistics lock.

Multiple parallel readers with readers-writer locks: Another optimization is to use a readers-writer lock in the statistics lock implementation. Similar to the optimistic readers extension this allows multiple parallel readers for read-only operations

like LOOKUP. We evaluate this extension in Section VII-B. In our readers-writer lock implementation we change LOOKUPS to first check if the statistics lock is not write-locked and acquire the statistics lock for reading in that case. If the statistics lock is write-locked, the statistics lock is acquired for write as normal. The readers-writer lock extension has an advantage compared to the optimistic readers extensions in languages without automatic memory management. Since readers are invisible in the optimistic readers extension some form of delayed deallocation has to be used for the tree nodes (and for data stored in them) to avoid freeing memory that is currently read. This is not necessary when a readers-writer lock is used since writers get exclusive access to the base nodes. Even through there are several methods for safe delayed allocation in languages without automatic memory management such as C and C++, they often incur a non-negligible overhead [18]. We therefore experimented with the sequence lock in a Java implementation and the readers-writer lock extension in a C implementation.

B. Parallel Critical Sections with Hardware Lock Elision

Some support for hardware transactional memory has recently started to become commonplace with Intel's Haswell architecture. A promising way to exploit the hardware transactional memory is through *hardware lock elision* (HLE) [29]. 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. CA trees make it possible to dynamically adapt the sizes of the critical regions to fit the hardware. One could simply use HLE to implement the SC locks. However this would lead to unnecessarily many failed transactions because the statistics counter that is modified inside the critical reader will be a hot spot. In our implementation, we try to avoid this problem by letting read-only operations first attempt to acquire the HLE lock and perform the critical section without reading or writing the statistics counter. If the first attempt fails, the SC lock is acquired in write mode and the operation is performed as usual. We evaluate the effect of HLE in Section VII-B.

C. An Optimization for Highly Contended Base Nodes

A base node that contains only one element cannot be split to reduce contention. Therefore, we describe 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 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 one (just one CAS or write instead of a lock call, a write and an unlock call).

To do this optimization we start from the "optimistic readers with sequence locks" optimization described above and add a state flag and a non-zero indicator to the base nodes. When the state flag is set to *off*, the base node is in the normal locking state and when the state flag is set to *on* the base node is in the lock-free state. A base node is transferred to the lock-free state when a high-contention adaptation is triggered and the base node contains one or zero elements. The transfer is done by flipping the state flag and installing the non-zero indicator while holding the sequence lock. We use a simple non-zero indicator with one cache line per core in the system to reduce false sharing between active threads.

Operations such as INSERT or DELETE that modify only one element start by checking the sequence counter of the base node lock and then the state flag before they acquire the base node lock or attempt to perform the operation in a lock-free way. If the state flag is set to *on* and the operation can be performed in the lock-free state, the operation proceeds by registering in the non-zero indicator. Then the operation checks the sequence counter in the lock again. If the sequence counter still has the same value, the operation can proceed to perform the operation by doing the appropriate CAS or write in the base node before unregistering in the non-zero indicator. If the sequence counter has changed, the operation has to unregister in the non-zero indicator and retry the operation because another operation has requested exclusive access to the base node. Operations that acquire the lock of a base node also have to check the state flag after the lock has been acquired and either transfer the base node to the locking state or unlock the lock and retry the operation, if the state flag is set to on. The transfer from lock-free state to the locking state is performed after the base node lock has been acquired by waiting for the non-zero indicator to indicate that no modifying operations are active in the base node and changing the state flag. A base node has to be transferred from the lock-free state to the locking state when e.g. an INSERT would result in more than one element in the base node or when a low-contention join needs exclusive access to the base node. To avoid that a CA tree gets stuck in a state where all base nodes are converted to the lock-free state, one can use random probing to occasionally transfer base nodes from the lock-free state to the locking state.

VI. Related Work

In the context of distributed DBMS, Joshi [21] 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 [24], diffracting trees [13], [17] and combining [32]. The reactive diffracting tree of Della-Libera and Shavit [13] shares some design ideas

with CA trees. Diffracting trees distribute accessing threads evenly over the leaf nodes of the tree and are used to implement shared counters and load balancing. Like CA trees, reactive diffracting trees estimate the contention and shrink and grow the synchronization granularity according to this estimate. However, their algorithm estimates contention by recording the times to execute a function in contrast to our simple SC lock. The self-tuning diffracting tree algorithm by Ha *et al.* does a local estimation of the number of threads that are accessing the tree from the number of threads that are waiting in a leaf [17]. Such an estimate does not make sense in a binary search tree where the threads are not distributed evenly. Also, our growing and shrinking procedures are very different from the one proposed by Ha *et al.* as well as that proposed by Della-Libera and Shavit.

A large number of concurrent ordered set data structures for multicores have recently been proposed. We will not be able to cover them all here but we will briefly describe the data structures that we compare against in Section VII-A and discuss a couple of other data structures that are interesting in this context. Fraser [16] 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 (e.g. [7], [10], [15], [20], [26], [27]). The relaxed balancing external lock-free tree by Brown *et al.* (Chromatic) is one of the best performing lock-free search trees [7]. 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[1]. A number of well performing lock-based trees have also been put forward recently [2], [6], [12], [14]. The tree of Bronson *et al.* (SnapTree) is a partially external tree inspired by the relaxed AVL tree by Bougé *et al.* [5]. SnapTree uses a copy on write technique to get a fast atomic snapshot capability [6]. This copy on write technique makes it possible for the SnapTree to provide operations that atomically read multiple elements, but the technique does not work for operations that atomically modify multiple elements. 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 [12]. The recently proposed LogOrdAVL tree by Drachsler *et al.* [14] 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 LogOrdAVL 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 large advantage in languages like C and C++ without automatic memory management that that CA trees lock the whole part of the tree 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 they are deleted much less frequently than ordinary nodes, CA trees are less dependent on memory management.

The CB tree [2] 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 sequential data structure since Red-Black trees provide slightly cheaper update operations at the cost of longer search paths than AVL trees.

The speculation-friendly tree by Crain *et al.* [11] utilizes transactional regions as our hardware lock elision optimized CA tree variant. To reduce expensive retries of transactional regions, speculation-friendly trees divide tree operations into several phases that are executed in different transactional regions.

The 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 can often be 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 to scale well on highly contended scenarios as we will see in the next section.

VII. Evaluation

We now evaluate the scalability of CA tree variants and the effect of the optimizations we presented in Section V.

A. Comparison to Other Data Structures

First we will compare CA trees with some recently proposed highly scalable algorithms for concurrent search trees. The code for all algorithms was provided by their authors. For brevity we refer to these algorithms by the acronyms we introduced in Section VI. The CFTree, which is represented with dashed gray lines in the graphs, is incompatible with the standard interface for sets and maps since it has a separate maintenance thread that has to be started and stopped when using the data structure. However, we have chosen to include the CFTree in our comparison anyway to show what kind of scalability one can expect when one is willing to use a non-standard API and dedicate one core for maintaining the tree's balance. We refer to our plain CA tree as CATree and to that with the sequence

[1] 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 [7].

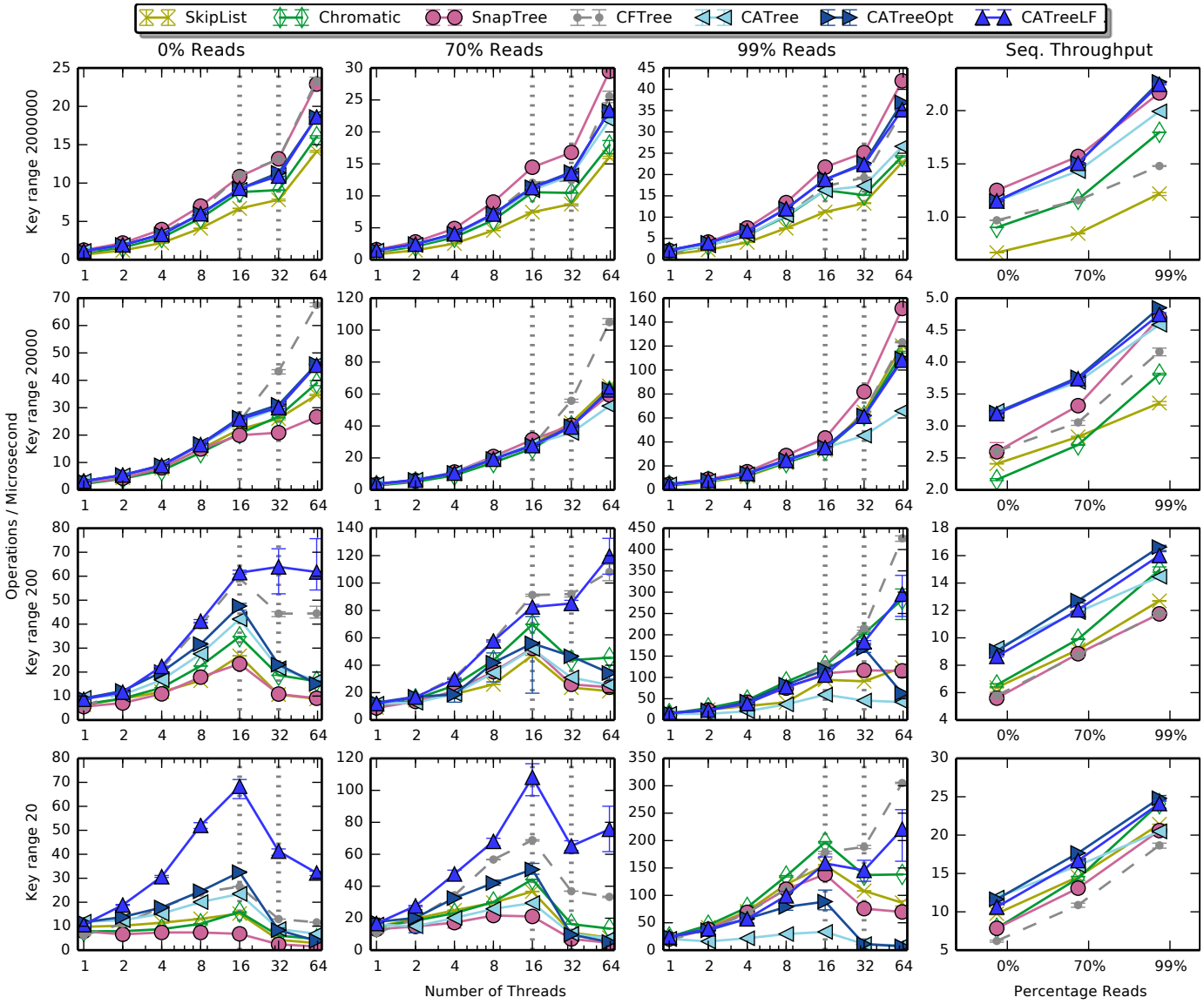

Fig. 6. Comparison to related concurrent data structures.

lock optimization as CATreeOpt. The version with the sequence lock optimization and the optimization that converts highly contented base nodes to a lock-free state is called CATreeLF. All CA tree variants use an AVL tree as sequential data structure.

Benchmark: The benchmark measures throughput of operations that a number of threads perform on the same data structure. The data structure is initialized by inserting $\frac{keyRange}{2}$ keys with keys randomly chosen from the interval $(0, keyRange]$. The threads select the operations randomly so that $L\%$ of them are LOOKUP, $\frac{100-L}{2}\%$ are INSERT, and the rest are DELETE operations. All threads select the next key randomly with an uniform distribution from the whole key range.

Setting: The machine we used has four Intel(R) Xeon(R) CPU E5-4650 CPUs (2.70GHz), eight cores each (i.e. the machine has a total of 32 physical cores, each with hyper-threading, which makes a total of 64 logical cores), 128GB of RAM and runs Debian Linux 3.10.17-amd64. We pin thread counts up to 16 to one NUMA node (i.e., on 16 logical but only 8 physical cores), 32 threads to two NUMA nodes, and so on. This way, we can see both single chip and NUMA performance in the same graph (the borders are indicated with dashed gray lines in the graphs). The Oracle Java 1.8.0_31 HotSpot(TM) (started with parameters `-Xmx8g`, `-Xms8g`, `-server` and `-d64`) was used to run the benchmarks. We ran one warm up run for 10 seconds and then three measurement runs for 10 seconds on the same JVM for every data point. In graphs we show the average throughput as well as error bars with the minimum and maximum.

Results: Selected graphs from the benchmark are shown in Fig. 6. From the graphs with the larger key range (2000000), it can be seen that CATreeOpt and CATreeLF performs similar to the best of the other data structures across all thread counts. The lock free data structures' (Chromatic and SkipList) performance is behind the rest of the data structures with the larger key range which can be explained with their longer search paths. (Remember that Chromatic is an external tree where all keys are stored in the leaves.) With key range 20000 and 99% reads, the CATree scales poorly compared to the other CA tree variants, which is not surprising since it needs to acquire a lock for every read operation, while the other CA tree variants can do

TABLE I
AVERAGE BASE NODE COUNTS (IN k) AT THE END OF RUNNING THE BENCHMARK WITH 70% READS AND KEY RANGE 2000000.

threads	2	4	8	16	32	64
CATree	0.24	0.68	1.5	2.9	6.7	13.2
CATreeOpt	0.72	2.1	4.5	8.9	19.6	36.8
CATreeLF	0.72	2.0	4.5	8.8	19.6	36.6

reads without writing to shared memory at all. CFTree scales best followed by the CA tree variants with key range 20000 and the more write heavy scenarios with 0% reads and 70% reads. Remember however that CFTree is incompatible with the normal interface for ordered sets and maps due to requiring a separate maintenance thread.

Let us now focus on the graphs with key range 200 and 20 that show the strength of CATreeLF. It is clear that CATreeLF copes best with the extremely high contention created with key range 200 or 20 combined with 0% or 70% read operations. Not surprisingly, inspection of CATreeLF after the benchmark runs shows that CATreeLF has converted all its base nodes to the lock-free state in these scenarios. Thus the operations are performed by modifications or reads in the leaves of the tree without acquiring any locks. The other data structures do not adapt to such high contention and keep doing restructuring operations, resulting in sub-optimal performance. In the less contended scenarios (key range 200 and 20 and 99% reads), the CFTree with its spinning balancing thread achieves best scalability, followed by CATreeLF and the lock-free tree Chromatic.

Finally, we discuss the sequential performance of these data structures, shown in the rightmost column of Fig. 6. Overall, the CA trees give the best sequential performance even though some of the other data structures match the CA trees' performance in some scenarios. The CA trees' performance in the sequential case is essentially the same as that of their sequential data structure (AVL tree) wrapped in a lock. As discussed previously, Chromatic and SkipList suffer from their longer search paths when the set size is large, and the CFtree probably suffers from its delayed balancing by the dedicated balancing thread in the sequential cases. The SnapTree has excellent sequential performance for larger set sizes where its synchronization overhead is a small part of the whole operation but suffers from its synchronization overhead for small set sizes.

The average base node counts (in k) collected after running the experiments with 70% reads and key range 2000000 are shown in TABLE I. CATreeOpt and CATreeLF end up with more base nodes than CATree since their substantially faster read operation makes conflicts with writes more likely. The base node counts also confirm that the CA trees adapt to the contention level (an increased thread count results in an increased base node count). Finally, we confirmed that the base node counts stabilize around a specific value given a static contention level by checking the base node counts after running the experiments for longer periods of time, which means that the base node count does not increase indefinitely.

B. Effect of HLE and RW Lock Optimizations

Finally, we evaluate the RW lock and HLE optimizations we described in Section V. The benchmark is the same as before,

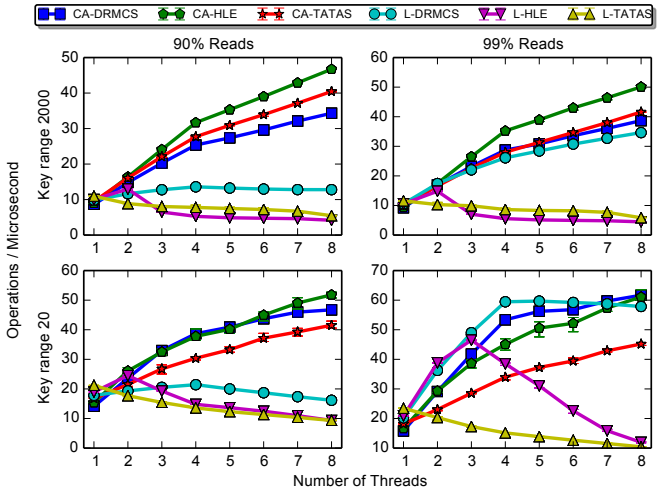

Fig. 7. Throughput for HLE (CA-HLE) and RW lock (CA-DRMCS) optimizations compared with a plain CA tree (CA-TATAS) and sequential trees protected by a TATAS lock (L-TATAS), RW lock (L-DRMCS), and an HLE lock (L-HLE).

but the data structures and the benchmark itself are implemented in C. The implementation is in C because HLE locks are currently not available for Java and because, as mentioned in Section V, RW locks have some advantages compared to sequence locks in an environment without automatic memory management. The machine we used for these experiments is also different because the previous machine does not support HLE. Here we used a machine with an Intel(R) Xeon(R) CPU E3-1230 v3 (3.30GHz), 4 cores with hyperthreading (8 logical cores), and 16GB of RAM running Ubuntu 13.10. We compiled the benchmark with GCC 4.8 using optimization level `-O3`.

Selected graphs from our experiments appear in Fig. 7. CA-DRMCS is our RW lock optimization using the write-preference RW lock with an MCS lock as mutex lock [8]. CA-HLE shows our HLE optimization and CA-TATAS is the plain CA tree algorithm using a Test-And-Test-And-Set (TATAS) lock for the statistics lock. All CA tree variants use an AVL tree as sequential data structure and a form of quiescent-state-based reclamation for routing nodes in the tree [3], [25]. We compare CA trees to a sequential AVL tree protected by an RW lock (L-DRMCS), HLE lock (L-HLE) and a TATAS lock (L-TATAS). As can be seen in the lower right corner of Fig. 7, the capacity of the hardware transactional memory is clearly limited. With a key range of 20 and 99% reads L-HLE starts to degregate after only three threads, so it is clear that HLE alone is not sufficient for good scalability. However with a key range of 20 and 99% reads CA-DRMCS, CA-HLE and L-DRMCS perform reasonably well. Going up to the graphs that show the key range 2000, it is clear that CA trees together with HLE perform well. With a key range of 2000, CA-DRMCS performs even worse than CA-TATAS. This is likely because the contention for each base node is too low to make concurrent readers at a base node common. Increasing the key range or percentage of updates even further makes the gaps between CA-HLE and CA-TATAS smaller to a point that they are almost indistinguishable.

To conclude, we have shown that, on a single-chip system with hardware transactional memory capabilities, combining HLE with CA trees can give improved performance and that

the overhead of the HLE lock seems to be small.

Finally, we mention that the code for all benchmarks as well as many additional graphs appear at http://www.it.uu.se/research/group/languages/software/ca_tree.

VIII. Concluding Remarks

In this paper, we have described a contention adapting approach to concurrent data structure design. Our description was restricted to binary search trees, but it should be easy to see that it can easily be extended to all data structures that naturally support a SPLIT and a JOIN operation. We have put forward CA trees, a new concurrent data structure for ordered sets and maps that is competitive with state-of-the-art algorithms in various scenarios with a larger set sizes. Moreover, CA trees outperform these algorithms in scenarios with smaller set sizes and very high contention as well as in scenarios with no contention at all. Their ability to perform well in a wide range of scenarios shows the strength of the contention adaptive approach to concurrent data structures.

In recent work, we have employed CA trees to increase the scalability of the `ordered_set` part of the Erlang Term Storage in-memory database [30]. CA trees are very well suited for general purpose key-value stores since they naturally adapt to a variety of scenarios and have efficient support for operations that need to access multiple keys atomically. CA trees' ability to perform well under high contention on small set sizes and their excellent uncontended performance also make them well suited for use in the hash buckets of concurrent hash tables.

Acknowledgments

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).

References

[1] G. Adelson-Velskii and E. M. Landis. An algorithm for the organization of information. In *Proceedings of the USSR Academy of Sciences*, volume 146, pages 263–266, 1962.

[2] Y. Afek, H. Kaplan, B. Korenfeld, A. Morrison, and R. E. Tarjan. CBTree: A practical concurrent self-adjusting search tree. In *International Symposium on Distributed Computing*, pages 1–15. Springer, 2012.

[3] A. Arcangeli, M. Cao, P. E. McKenney, and D. Sarma. Using read-copy-update techniques for system V IPC in the Linux 2.5 kernel. In *USENIX Annual Technical Conference, FREENIX Track*, pages 297–309. USENIX, 2003.

[4] R. Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1(4):290–306, 1972.

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

[6] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A practical concurrent binary search tree. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 257–268. ACM, 2010.

[7] T. Brown, F. Ellen, and 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, pages 329–342, New York, NY, USA, 2014. ACM.

[8] I. Calciu, D. Dice, Y. Lev, V. Luchangco, V. J. Marathe, and N. Shavit. NUMA-aware reader-writer locks. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 157–166, New York, NY, USA, 2013. ACM.

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

[10] B. Chatterjee, N. Nguyen, and P. Tsigas. Efficient lock-free binary search trees. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, pages 322–331, New York, NY, USA, 2014. ACM.

[11] T. Crain, V. Gramoli, and 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, pages 161–170, New York, NY, USA, 2012. ACM.

[12] T. Crain, V. Gramoli, and M. Raynal. A contention-friendly binary search tree. In *Euro-Par 2013 Parallel Processing - 9th International Conference*, volume 8097 of *LNCS*, pages 229–240. Springer, 2013.

[13] G. Della-Libera and N. Shavit. Reactive diffracting trees. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '97, pages 24–32, New York, NY, USA, 1997. ACM.

[14] D. Drachsler, M. Vechev, and 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, pages 343–356, New York, NY, USA, 2014. ACM.

[15] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel. Non-blocking binary search trees. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '10, pages 131–140, New York, NY, USA, 2010. ACM.

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

[17] P. H. Ha, M. Papatriantafilou, and P. Tsigas. Self-tuning reactive diffracting trees. *Journal of Parallel and Distributed Computing*, 67(6):674–694, 2007.

[18] T. E. Hart, P. E. McKenney, A. D. Brown, and J. Walpole. Performance of memory reclamation for lockless synchronization. *Journal of Parallel and Distributed Computing*, 67(12):1270–1285, 2007.

[19] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[20] S. V. Howley and J. Jones. A non-blocking internal binary search tree. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, pages 161–171, New York, NY, USA, 2012. ACM.

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

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

[23] C. Lameter. Effective synchronization on Linux/NUMA systems. In *Proc. of the Gelato Federation Meeting*, 2005.

[24] B.-H. Lim and 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, pages 25–35, New York, NY, USA, 1994. ACM.

[25] P. E. McKenney and J. D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, 1998.

[26] A. Natarajan and 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, pages 317–328, New York, NY, USA, 2014. ACM.

[27] A. Natarajan, L. H. Savoie, and N. Mittal. Concurrent wait-free red black trees. In *Stabilization, Safety, and Security of Distributed Systems*, pages 45–60. Springer, 2013.

[28] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.

[29] R. Rajwar and 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, pages 294–305, Washington, DC, USA, 2001. IEEE Computer Society.

[30] K. Sagonas and K. Winblad. More scalable ordered set for ETS using adaptation. In *ACM Erlang Workshop*, pages 3–11. ACM, Sept. 2014.

[31] K. Sagonas and K. Winblad. Efficient support for range queries and range updates using contention adapting search trees. Tech. Report, available in [9], 2015.

[32] N. Shavit and A. Zemach. Combining funnels: a dynamic approach to software combining. *Journal of Parallel and Distributed Computing*, 60(11):1355–1387, 2000.

[33] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.