The Contention-Friendly Tree

Tyler Crain[1], Vincent Gramoli[2], and Michel Raynal[1,3]

[1] IRISA, Université de Rennes 1, France
[2] NICTA and University of Sydney, Australia
[3] Institut Universitaire de France
tyler.crain@irisa.fr,vincent.gramoli@sydney.edu.au,raynal@irisa.fr

Abstract. This paper proposes a new lock-based concurrent binary tree using a methodology for writing concurrent data structures. This methodology limits the high contention induced by today's multicore environments to come up with efficient alternatives to the most widely used search structures.
Data structures are generally constrained to guarantee a big-oh step complexity even in the presence of concurrency. By contrast our methodology guarantees the big-oh complexity only in the absence of contention and limits the contention when concurrency appears. The key concept lies in dividing update operations within an *eager abstract access* that returns rapidly for efficiency reason and a *lazy structural adaptation* that may be postponed to diminish contention. Our evaluation clearly shows that our lock-based tree is up to 2.2× faster than the most recent lock-based tree algorithm we are aware of.

Keywords: Binary tree, Concurrent data structures, Efficient implementation.

1 Introduction and Related Work

Today's processors tend to embed more and more cores. Concurrent data structures, which implement popular abstractions such as key-value stores [1], are thus becoming a bottleneck building block of a wide variety of concurrent applications. Maintaining the invariants of such structures, like the balance of a tree, induces contention. This is especially visible when using speculative synchronization techniques as it boils down to restarting operations [2]. In this paper we describe how to cope with the contention problem when it affects a non-speculative technique.

As a widely used and studied data structure in the sequential context, binary trees provide logarithmic access time complexity given that they are balanced, meaning that among all downward paths from the root to a leaf, the length of the shortest path is not far apart the length of the longest path. Upon tree update, if the height difference exceeds a given threshold, the structural invariant is broken and a rebalancing is triggered to restructure accordingly. This threshold depends on the considered algorithm: AVL trees [3] do not tolerate the longest length to exceed the shortest by 2 whereas red-black trees [4] tolerate the longest to be twice the shortest, thus restructuring less frequently. Yet in both cases the restructuring is triggered immediately when the threshold is reached to hide the imbalance from further operations. In a concurrent context, slightly weakened balance requirements have been suggested [5], but they still require immediate restructuring as part of update operations to the abstractions.

We introduce the *contention-friendly* tree as a tree that transiently breaks its balance structural invariant without hampering the abstraction consistency in order to reduce contention and speed up concurrent operations that access (or modify) the abstraction. More specifically, we propose a partially internal binary search tree data structure implementing a key-value store, decoupling the operations that modify the abstraction (we call these *abstract operations*) from operations that modify the tree structure itself but not the abstraction (we call these *structural operations*). An abstract operation either searches for, logically deletes, or inserts an element from the abstraction where in certain cases the insertion might also modify the tree structure. Separately, some structural operations rebalance the tree by executing a distributed rotation mechanism as well as physically removing nodes that have been logically deleted.

Context. On the one hand, the decoupling of update and rebalancing dates back from the 70's [6] and was exclusively applied to trees, including B-trees [7], $\{2,3\}$-trees [8], AVL trees [9] and red-black trees [10] (resulting in largely studied chromatic trees [11, 12] whose operations cannot return before reaching a leaf). On the other hand, the decoupling of the removals in logical and physical phases is more recent [13] but was applied to various structures: linked lists [13, 14], hash tables [15], skip lists [16], binary search trees [2, 17] and lazy lists [18]. Our methodology generalizes both kinds of decoupling by distinguishing an abstract update from a structural modification.

The guarantees of some data structures, like list-based stack, are relaxed to tolerate the high concurrency induced by multicores [19]. This idea is quite different from ours. It aims at avoiding performance of some highly contended structures to drop below sequential ones whereas we aim at designing highly-concurrent structures that leverage multi-/many-cores. Finally, the corresponding solution lies in trading off atomicity for quiescent consistency, guaranteeing that the last-in-first-out policy of an access is only with respect to preceding calls when no other accesses execute concurrently. By contrast, our solution guarantees atomicity even in concurrent executions.

We have recently observed the performance benefit of decoupling accesses while preserving atomicity. Our recent speculation-friendly tree splits updates into separate transactions to avoid a conflict with a rotation from rolling back the preceding insertion/removal [2]. While it benefits from the reusability and efficiency of elastic transactions [20], it suffers from the overhead of bookkeeping accesses with software transactional memory. The goal was to bound the asymptotic step complexity of speculative accesses to make it comparable to the complexity of pessimistic lock-based ones. Although this complexity is low in pessimistic executions, our new result shows that the performance of a lock-based binary search tree greatly benefits from this decoupling.

Content of the paper. We present a contention-friendly methodology which lies essentially in splitting accesses into an eager abstract access and a lazy structural adaptation. We illustrate our methodology with a contention-friendly binary search tree. In particular, we compare its performance against the performance of the most recent practical binary search tree we are aware of [21]. Although both algorithms are lock-based binary search trees, ours speeds up the other by $2.2\times$.

2 Overview

In this section, we give an overview of the Contention-Friendly (CF) methodology by describing how to write contention-friendly data structures as we did to design a lock-free CF skip-list [22, 23]. The following section will describe how this is specifically done for the binary search tree.

The CF methodology aims at modifying the implementation of existing data structures using two simple rules without relaxing their correctness. The correctness criterion ensured here is linearizability [24]. The data structures considered are *search structures* because they organize a set of items, referred to as *elements*, in a way that allows to retrieve the unique position of an element in the structure given its value. The typical abstraction implemented by such structures is a collection of elements that can be specialized into various sub-abstractions like a set (without duplicates) or a map (that maps each element to some value). We consider *insert*, *delete* and *contains* operations that, respectively, inserts a new element associated to a given value, removes the element associated to a given value or leaves the structure unchanged if no such element is present, and returns the element associated to a given value or $\perp$ if such an element is absent. Both inserts and deletes are considered *updates*, even though they may not modify the structure.

The key rule of the methodology is to decouple each update into an *eager abstract modification* and a *lazy structural adaptation*. The secondary rule is to make the removal of nodes selective and tentatively affect the less loaded nodes of the data structure. These rules induce slight changes to the original data structures that result in a corresponding data structure that we denote using the *contention-friendly* adjective to differentiate them from their original counterpart.

2.1 Eager abstract modification

Existing search structures rely on strict invariants to guarantee their big-oh (asymptotic) complexity. Each time the structure gets updated, the invariant is checked and the structure is accordingly adapted as part of the same operation. While the update may affect a small sub-part of the abstraction, its associated restructuring can be a global modification that potentially conflicts with any concurrent update, thus increasing contention.

The CF methodology aims at minimizing such contention by returning eagerly the modifications of the update operation that make the changes to the abstraction visible. By returning eagerly, each individual process can move on to the next operation prior to adapting the structure. It is noteworthy that executing multiple abstract modifications without adapting the structure does no longer guarantee the big-oh step complexity of the accesses, yet, as mentioned in the Introduction, such complexity may not be the predominant factor in contended executions.

A second advantage is that removing the structural adaption from the abstract modification makes the cost of each operation more predictable as operations share similar cost and create similar amount of contention. More importantly the completion of the abstract operation does not depend on the structural adaptation (like they do in existing algorithms), so the structural adaptation can be performed differently, for example, using global information or being performed by separate, unused resources of the system.

2.2 Lazy structural adaptation

The purpose of decoupling the structural adaptation from the preceding abstract modification is to enable its postponing (by, for example, dedicating a separate thread to this task, performing adaptations when observed to be necessary), hence the term "lazy" structural adaptation. The main intuition here is that this structural adaptation is intended to ensure the big-oh complexity rather than to ensure correctness of the state of the abstraction. Therefore the linearization point of the update operation belongs to the execution of the abstract modification and not the structural adaptation and postponing the structural adaptation does not change the effectiveness of operations.

This postponing has several advantages whose prominent one is to enable merging of multiple adaptations in one simplified step. Only one adaptation might be necessary for several abstract modifications and minimizing the number of adaptations decreases accordingly the induced contention. Furthermore, several adaptations can compensate each other as the combination of two restructuring can be idempotent. For example, a left rotation executing before a right rotation at the same node may lead back to the initial state and executing the left rotation lazily makes it possible to identify that executing these rotations is useless. Following this, instead of performing rotations as a string of updates as part of a single abstract operation, each rotation is performed separately as a single local operation, using the most up to date balance information.

Although the structural adaptation might be executed in a distributed fashion, by each individual updater thread, one can consider centralizing it at one dedicated thread. Since these data structures are designed for architectures that use many cores, performing the structural adaptation on a dedicated single separate thread leverages hardware resources that might otherwise be left idle.

Selective removal. In addition to decoupling level adjustments, removals are preformed selectively. A node that is deleted is not removed instantaneously but is marked as deleted. The structural adaptation then selects among these marked nodes those that are suitable for removal, i.e., whose removal would not induce high contention. This selection is important to limit contention. Removing a frequently accessed node requires locking or invalidating a larger portion of the structure. Removing such a node is likely to cause much more contention than removing a less frequently accessed one. In order to prevent this, only nodes that are marked as deleted and have at least one of their children as an empty subtree are removed. Marked deleted nodes can then be added back into the abstraction by simply unmarking them during an *insert* operation. This leads to less contention, but also means that certain nodes that are marked as deleted may not be removed. In similar, partially external/internal trees, it has already been observed that only removing such nodes [2], [21] results in a similar sized structure as existing algorithms.

3 The Contention-Friendly Tree

The CF tree is a lock-based concurrent binary search tree implementing classic insert/delete/contains operations. Each of its nodes contains the following fields: a key k, pointers l and r to the left and right child nodes, a *lock* field, a *del* flag indicating

if the node has been logically deleted, a *rem* flag indicating if the node has been physically removed, and the integers *left-h*, *right-h* and *local-h* storing the estimated height of the node and its subtrees used in order to decide when to perform rotations.

This section will now describe the CF tree algorithm by first describing three specific CF modifications that reduce contention during traversal, followed by a description of the CF abstract operations.

3.1 Avoiding contention during traversal

Each abstract operation of a tree is expected to traverse $O(\log n)$ nodes when there is no contention. During an update operation, once the traversal is finished a single node is then modified in order to update the abstraction. In the case of delete, this means setting the *del* flag to true, or in the case of insert changing the child pointer of a node to point to a newly allocated node (or unmarking the *del* flag in case the node exists in the tree). Given then, that the traversal is the longest part of the operation, the CF tree algorithm tries to avoid here, as often as possible, producing contention. Traditionally, concurrent data structures often require synchronization during traversal (not even including the updates done after the traversal). For example, performing hand-over-hand locking in a tree helps ensure that the traversal remains on track during a concurrent rotation [21], or, using optimistic strategy (such as transactional memory), validation is done during the traversal, risking the operation to restart in the case of concurrent modifications [18, 25, 26].

Physical removal. As previously mentioned, the algorithm attempts to remove only nodes whose removal incurs the least contention. Specifically, removing a node n with a subtree at each child requires finding its successor node s in one of its subtrees, then replacing n with s. Therefore precautions must be taken (such as locking all the nodes) in order to ensure any concurrent traversal taking place on the path from n to s does not violate linearizability. Instead of creating contention by removing such nodes, they are left as logically deleted in the CF tree; to be removed later if one of their subtrees becomes empty, or to be unmarked if a later insert operation on the same node occurs.

In the CF tree, nodes that are logically deleted and have less than two child subtrees are physically removed lazily (cf. Algorithm 1). Since we do not want to use synchronization during traversal these removals are done slightly differently than by just unlinking the node. The operation starts by locking the node n to be removed and its parent p (line 6). Following this, the appropriate child pointer of p is then updated (lines 12-13), effectively removing n from the tree. Additionally, before the locks are released, both of n's left and right pointers are modified to point back to p and the *rem* flag of n is set to true (lines 14-15). These additional modifications allow concurrent abstract operations to keep traversing safely as they will then travel back to p before continuing their traversal, much like would be done in a solution that uses backtracking.

Rotations. Rotations are performed to rebalance the tree so that traversals execute in $O(\log n)$ time once contention decreases. As described in Section 2, the CF tree uses localized rotations in order to minimize conflicts. Methods for performing localized rotation operations in the binary trees have already been examined and proposed in

Algorithm 1 Remove and rotate operations

```
1:  remove(parent, left-child)_p:
2:    if parent.rem then return false
3:    if left-child then n ← parent.ℓ
4:    else n ← parent.r
5:    if n = ⊥ then return false
6:    lock(parent); lock(n);
7:    if ¬n.del then return false // release locks
8:    if (child ← n.ℓ) ≠ ⊥ then
9:      if n.r ≠ ⊥ then
10:       return false // release locks
11:   else child ← n.r
12:   if left-child then parent.ℓ ← child
13:   else parent.r ← child
14:   n.ℓ ← parent; n.r ← parent;
15:   n.rem ← true;
16:   // release locks
17:   update-node-heights();
18:   return true.
```

```
19: right-rotate(parent, left-child)_p:
20:   if parent.rem then return false
21:   if left-child then n ← parent.ℓ
22:   else n ← parent.r
23:   if n = ⊥ then return false
24:   ℓ ← n.ℓ;
25:   if ℓ = ⊥ then return false
26:   lock(parent); lock(n); lock(ℓ);
27:   ℓr ← l.r; r ← n.r;
28:   // allocate a node called new
29:   new.k ← n.k; new.ℓ ← ℓr;
30:   new.r ← r; ℓ.r ← new;
31:   if left-child then parent.ℓ ← ℓ
32:   else parent.r ← ℓ
33:   n.rem ← true; // by-left-rot if left rotation
34:   // release locks
35:   update-node-heights();
36:   return true
```

several works such as [5]. The main concept used here is to propagate the balance information from a leaf to the root. When a node has a $\perp$ child pointer then the node must know that this subtree has height 0 (the estimated heights of a node's subtrees are stored in the integers *left-h* and *right-h*). This information is then propagated upwards by sending the height of the child to the parent, where the value is then increased by 1 and stored in the parent's *local-h* integer. Once an imbalance of height more than 1 is discovered, a rotation is performed. Higher up in the tree the balance information might become out of date due to concurrent structural modifications, but, importantly, performing these local rotations will eventually result in a balanced tree [5].

Apart from performing rotations locally as unique operations, the specific CF rotation procedure is done differently in order to avoid using locks and aborts/rollbacks during traversals. Let us consider specifically the typical tree right-rotation operation procedure. Here we have three nodes modified during the rotation: a parent node p, its child n who will be rotated downward to the right, as well as n's left child ℓ who will be rotated upwards, thus becoming the child of p and the parent of n. Consider a concurrent traversal that is preempted on n during the rotation. Before the rotation, ℓ and its left subtree exist below n as nodes in the path of the traversal, while afterwards (given that n is rotated downwards) these are no longer in the traversal path, thus violating correctness if these nodes are in the correct path. In order to avoid this, mechanisms such as hand over hand locking [21] or keeping count of the number of operations currently traversing a node [5] have been suggested, but these solutions require traversals to make themselves visible at each node, creating contention. Instead, in the CF tree, the rotation operation is slightly modified, allowing for safe, concurrent, invisible traversals.

The rotation procedure is then performed as follows as shown in Algorithm 1: The parent p, the node to be rotated n, and n's left child ℓ are locked in order to prevent

Algorithm 2 Restructuring process

```
1:  background-struct-adaptation()_p:
2:    while true do
3:      // continuous background restructuring
4:      restructure-node(root)_s.

5:  propagate(n)_s:
6:    if n.ℓ ≠ ⊥ then n.left-h ← n.ℓ.localh
7:    else n.left-h ← 0
8:    if n.r ≠ ⊥ then n.right-h ← n.r.localh
9:    else n.right-h ← 0
10:   n.localh ← max(n.left-h, n.right-h) + 1.

11: restructure-node(node)_s:
12:   if node = ⊥ then return
13:   restructure-node(node.ℓ)_s;
14:   restructure-node(node.r)_s;
15:   if node.ℓ ≠ ⊥ ∧ node.ℓ.del then
16:     remove(node, false)
17:   if node.r ≠ ⊥ ∧ node.r.del then
18:     remove(node, true)
19:   propagate(node);
20:   if |node.left-h − node.right-h| > 1 then
21:     // Perform appropriate rotations.
```

conflicts with concurrent insert and delete operations. Next, instead of modifying n like would be done in a traditional rotation, a new node *new* is allocated to take n's place in the tree. The key, value, and *del* fields of *new* are set to be the same as n's. The left child of *new* is set to $\ell.r$ and the right child is set to $n.r$ (these are the nodes that would become the children of n after a traditional rotation). Next $\ell.r$ is set to point to *new* and p's child pointer is updated to point to ℓ (effectively removing n from the tree), completing the structural modifications of the rotation. To finish the operation $n.rem$ is set to true (or by-left-rot, in the case of a left-rotation) and the locks are released. There are two important things to notice about this rotation procedure: First, *new* is the exact copy of n and, as a result, the effect of the rotation is the same as a traditional rotation, with *new* taking n's place in the tree. Second, the child pointers of n are not modified, thus all nodes that were reachable from n before the rotation are still reachable from n after the rotation, thus, any current traversal preempted on n will still be able to reach any node that was reachable before the rotation.

Structural adaptation. As mentioned earlier, one of the advantages of performing structural adaptation lazily is that it does not need to be executed immediately as part of the abstract operations. In a highly concurrent system this gives us the possibility to use processor cores that might otherwise be idle to perform the structural adaptation, which is exactly what is done in the CF tree. A fixed structural adaption thread is then assigned the task of running the background-struct-adaptation operation which repeatably calls the restructure-node procedure on the root node, as shown in Algorithm 2, taking care of balance and physical removal. restructure-node is simply a recursive depth-first procedure that traverses the entire tree. At each node, first the operation attempts to physically remove its children if they are logically deleted. Following this, it propagates balance values from its children and if an imbalance is found, a rotation is performed.

While here we have a single thread constantly running, other posibilites such as having several structural adaptations threads, or distributing the work amongst application threads can be used. It should be noted that, in a case where there can be multiple threads performing structural adaptation, we would need to be more careful on when and how the locks are obtained.

Algorithm 3 Abstract operations

```
1:  contains(k)_p:
2:    node ← root;
3:    while true do
4:      next ← get_next(node, k);
5:      if next = ⊥ then break
6:      node ← next;
7:    result ← false;
8:    if node.k = k then
9:      if ¬node.del then result ← true
10:   return result.

11: insert(k)_p:
12:   node ← root;
13:   while true do
14:     next ← get_next(node, k);
15:     if next = ⊥ then
16:       lock(node);
17:       if validate(node, k) then break
18:       unlock(node);
19:     else node ← next
20:   result ← false;
21:   if node.k = k then
22:     if node.del then
23:       node.del ← false; result ← true
24:   else // allocate a node called new
25:     new.key ← k;
26:     if node.k > k then node.r ← new
27:     else node.ℓ ← new
28:     result ← true;
29:   unlock(node);
30:   return result.

31: delete(k)_p:
32:   node ← root
33:   while true do
34:     next ← get_next(node, k);
35:     if next = ⊥ then
36:       lock(node);
37:       if validate(node, k) then break
38:       unlock(node);
39:     else node ← next
40:   result ← false;
41:   if node.k = k then
42:     if ¬node.del then
43:       node.del ← true; result ← true
44:   unlock(node);
45:   return result.

46: get-next(node, k)_s:
47:   rem ← node.rem;
48:   if rem = by-left-rot then next ← node.r
49:   else if rem then next ← node.ℓ
50:   else if node.k > k then next ← node.r
51:   else if node.k = k then next ← ⊥
52:   else next ← node.ℓ
53:   return next.

54: validate(node, k)_s:
55:   if node.rem then return false
56:   else if node.k = k then return true
57:   else if node.k > k then next ← node.r
58:   else next ← node.ℓ
59:   if next = ⊥ then return true
60:   return false.
```

3.2 Abstract operations

The abstract operations are shown in Algorithm 3. Each of the abstract operations begin by starting their traversal from the *root* node. The traversal is then performed, without using locks, from within a while loop where each iteration of the loop calls the get-next procedure, which returns either the next node in the traversal, or $\perp$ in the case that the traversal is finished.

The get-next procedure starts by reading the *rem* flag of *node*. If the flag was set to by-left-rotate then the node was concurrently removed by a left-rotation. As we saw in the previous section, a node that is removed during rotation is the node that would be rotated downwards in a traditional rotation. Specifically, in the case of the left rotation, the removed node's right child is the node rotated upwards, therefore in this case, the get-next operation can safely travel to the right child as it contains at least as many

nodes in its path that were in the path of the *node* before the rotation. If the flag was set to true then the node was either removed by a physical removal or a right-rotation, in either case the operation can safely travel to the left child, this is because the remove operation changes both of the removed node's child pointers to point to the parent and the right-rotation is the mirror of the left-rotation. If the rem flag is false then the key of *node* is checked, if it is found to be equal to k then the traversal is finished and $\perp$ is returned. Otherwise the traversal is performed as expected, traversing to the right if the $node.k$ is bigger than k or to the left if smaller.

Given that the insert and delete operations might modify *node*, they lock it for safety once $\perp$ is returned from get-next. Before the node is locked, a concurrent modification to the tree might mean that the traversal is not yet finished (for example the node might have been physically removed before the lock was taken), thus the validate operation is called. If false is returned by validate, then the traversal must continue, otherwise the traversal is finished. Differently, given that it makes no modifications, the contains operation exits the while loop immediately when $\perp$ is returned from get-next.

The validate operation performs three checks on *node* to ensure that the traversal is finished. First it ensures that $rem =$ false, meaning that the node has not been physically removed from the tree. Then it checks if the key of the node is equal to k, in such a case the traversal is finished and true is returned immediately. If the key is different from k then the traversal is finished only if *node* has $\perp$ for the child where a node with key k would exist. In such a case true is returned, otherwise *false* is returned.

Once the traversal is complete the rest of the code is straightforward. For the contains operation, true is returned if $node.k = k$ and $node.del =$ false, false is returned otherwise. For the insert operation, if $node.k = k$ then the *del* flag is checked, if it is false, then false is returned; otherwise if the flag is true it is set to false, and true is returned. In the case that $node.k \neq k$, a new node is allocated with key k and is set to be the child of *node*. For the delete operation, if $node.k \neq k$, then false is returned. Otherwise, the *del* flag is checked, if it is true then false is returned, otherwise if the flag is false, it is set to true and true is returned.

Linearization. Given that the insert and delete operations that return false do not modify the tree and that all other operations that modify nodes only do so while owning the node's locks, these failed insert and delete operations can be linearized at any point during the time that they own the lock of the node that was successfully validated.

The successful insert (i.e., the one that returns true) operation is linearized either at the instant it changes $node.del$ to false, or when it changes the child pointer of *node* to point to *new*. In either case, k exists in the abstraction immediately after the modification. The successful delete operation is linearized at the instant it changes $node.del$ to true, resulting in k no longer being in the abstraction.

The contains operation is a bit more difficult as it does not use locks. To give an intuition of how it is linearized, first consider a system where neither rotations nor physical removals are performed. In this system, if $node.k = k$ on line 8 is true, then the linearization point is when $node.del$ is read (line 9). Otherwise if $node.k \neq k$, then the linearization point is either on line 50 or 52 of the get-next operation where $\perp$ is read as the next node (meaning at the time of this read k does not exist in the abstraction).

Now, if rotations and physical removals are performed in the system, then a contains operation who has finished its traversal might get preempted on a node that is removed from the tree. First consider the case where $node.k = k$, since neither rotations nor removals will modify the *del* flag of a node, then in this case the linearization point is simply either on line 50 or 52 of the get-next operation where the pointer to *node* was read.

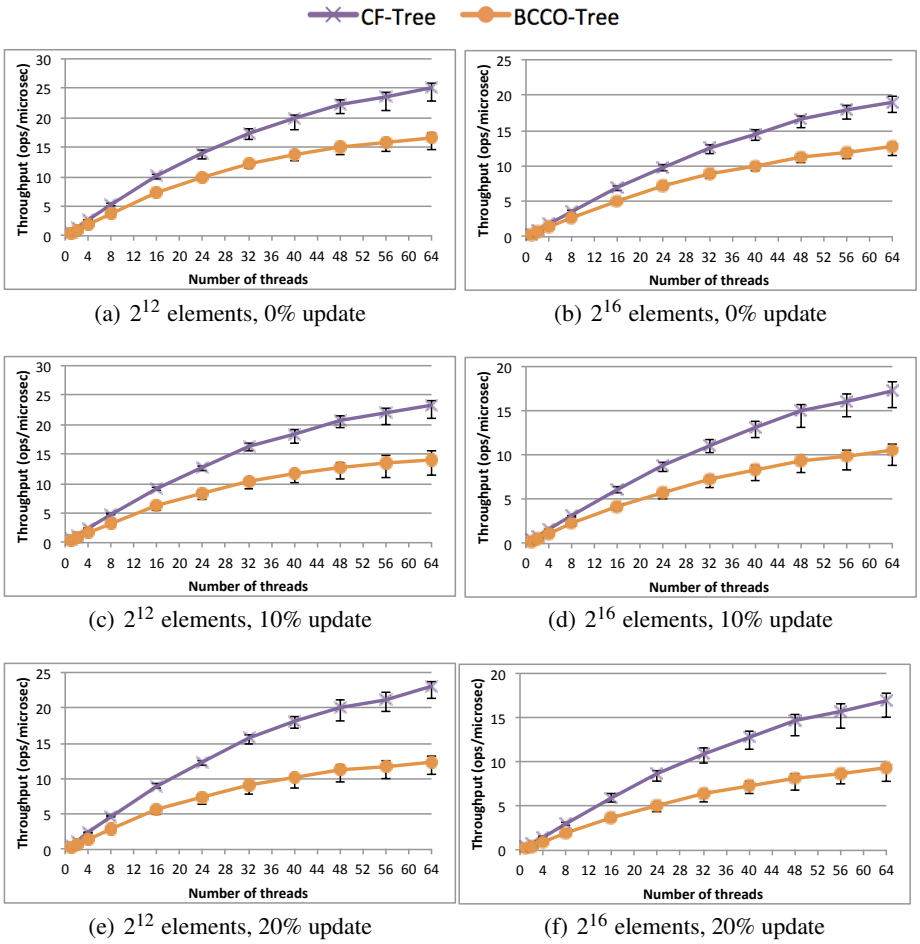
Now consider the case where $node.k \neq k$. First notice that when false is not read from node.rem (line 47 of get-next) then the traversal will always continue to another node. This is due to the facts that after a right (resp. left) rotation, the node removed from the tree will always have a non-$\perp$ left (resp. right) child (this is the child rotated upwards by the rotation) and that a node removed by a remove operation will never have a $\perp$ child pointer. Therefore if the traversal finishes on a node that has been removed from the tree, it must have read that node's *rem* flag before the rotation or removal had completed. This read will then be the linearization point of the operation. In this case, for the contains operation to complete, the next node in the traversal must be read as $\perp$ from the child pointer of *node*, meaning that the removal/rotation has not made any structural modifications to this pointer at the time of the read (this is because rotations make no modifications to the child pointers of the node they remove, and removals point the removed node's pointers towards its parent). Thus, given that removals and rotations will lock the node removed meaning no concurrent modifications will take place, effectively the contains operation has observed the state of the abstraction immediately before the removal took place.

4 Evaluation

We compare the performance of the contention-friendly tree (CF-tree) against the most recent lock-based binary search tree we are aware of (BCCO-tree [21]) on an UltraSPARC T2 with 64 hardware threads. For each run, we present the maximum, minimum, and averaged numbers of operations per microsecond over 5 runs of 5 seconds executed successively as part the same JVM for the sake of warmup. We used Java SE 1.6.0 12-ea in server mode and HotSpot JVM 11.2-b01 for both tree algorithms.

Figure 1 compares the performance of the practical BCCO tree against performance of our binary search tree with 2^{12} (left) and 2^{16} elements (right) and on a read-only workload (top) and workloads comprising up to 20% updates (bottom). The variance of our results is quite low as illustrated by the relatively short error bars we have. While both trees scale well with the number of threads, the BCCO tree is slower than its contention-friendly counterpart in all the various settings.

In particular, our CF tree is up to 2.2× faster than its BCCO counterpart. As expected, the performance benefit of our contention-friendly tree increases generally with the level of contention. (We observed this phenomenon at higher update ratios but omitted these graphs for the sake of space.) First, the performance improvement increases with the level of concurrency on Figures 1(c), 1(d), 1(e) and 1(f). As each thread updates the memory with the same (non-null) probability the contention increases with the number of concurrent threads running. Second, the performance improvement increases with the update ratio. This is not surprising as our tree relaxes the balance invariant dur-



(a) 2^{12} elements, 0% update (b) 2^{16} elements, 0% update

(c) 2^{12} elements, 10% update (d) 2^{16} elements, 10% update

(e) 2^{12} elements, 20% update (f) 2^{16} elements, 20% update

Fig. 1. Performance of our contention-friendly tree and the practical concurrent tree [21]

ing contention peaks whereas the BCCO tree induces more contention to maintain the balance invariant.

5 Concluding Remarks

To conclude, lock-based data structures can greatly benefit from the contention-friendly methodology on multicore architectures. In particular the decoupling of accesses allow the contention-friendly binary search tree to scale with a reasonably large number of hardware threads. An interesting future work would be to experimentally assess the performance gain due to applying contention-friendliness to other data structures. Additionally, a contention metric that complements the traditional asymptotic step complexity seems to be necessary to capture the cost of a multicore data structure.

References

1. Mao, Y., Kohler, E., Morris, R.T.: Cache craftiness for fast multicore key-value storage. In: EuroSys. (2012) 183–196
2. Crain, T., Gramoli, V., Raynal, M.: A speculation-friendly binary search tree. In: PPoPP. (2012) 161–170
3. Adelson-Velskii, G., Landis, E.M.: An algorithm for the organization of information. In: Proc. of the USSR Academy of Sciences. Volume 146. (1962) 263–266
4. Bayer, R.: Symmetric binary B-trees: Data structure and maintenance algorithms. Acta Informatica 1 **1**(4) (1972) 290–306
5. Bougé, L., Gabarro, J., Messeguer, X., Schabanel, N.: Height-relaxed AVL rebalancing: A unified, fine-grained approach to concurrent dictionaries. Technical Report RR1998-18, ENS Lyon (1998)
6. Guibas, L.J., Sedgewick, R.: A dichromatic framework for balanced trees. In: FOCS. (1978) 8–21
7. Bayer, R., McCreight, E.: Organization and maintenance of large ordered indices. In: Proc. of the ACM SIGFIDET Workshop on Data Description, Access and Control. (1970) 107–141
8. Huddleston, S., Mehlhorn, K.: A new data structure for representing sorted lists. Acta Inf. **17** (1982) 157–184
9. Kessels, J.L.W.: On-the-fly optimization of data structures. Commun. ACM **26**(11) (1983) 895–901
10. Nurmi, O., Soisalon-Soininen, E.: Uncoupling updating and rebalancing in chromatic binary search trees. In: PODS. (1991) 192–198
11. Nurmi, O., Soisalon-Soininen, E.: Chromatic binary search trees. A structure for concurrent rebalancing. Acta Inf. **33**(6) (1996) 547–557
12. Boyar, J., Fagerberg, R., Larsen, K.S.: Amortization results for chromatic search trees, with an application to priority queues. J. Comput. Syst. Sci. **55**(3) (1997) 504–521
13. Mohan, C.: Commit-LSN: a novel and simple method for reducing locking and latching in transaction processing systems. In: VLDB. (1990) 406–418
14. Harris, T.: A pragmatic implementation of non-blocking linked-lists. In: DISC. (2001) 300–314
15. Michael, M.M.: High performance dynamic lock-free hash tables and list-based sets. In: SPAA. (2002) 73–82
16. Fraser, K.: Practical lock freedom. PhD thesis, Cambridge University (September 2003)
17. Ellen, F., Fatourou, P., Ruppert, E., van Breugel, F.: Non-blocking binary search trees. In: PODC. (2010) 131–140
18. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kauffman (2008)
19. Shavit, N.: Data structures in the multicore age. Commun. ACM **54**(3) (2011) 76–84
20. Felber, P., Gramoli, V., Guerraoui, R.: Elastic transactions. In: DISC. (2009) 93–108
21. Bronson, N.G., Casper, J., Chafi, H., Olukotun, K.: A practical concurrent binary search tree. In: PPoPP. (2010)
22. Crain, T., Gramoli, V., Raynal, M.: Brief announcement: A contention-friendly, non-blocking skip list. In: DISC. (2012) 423–424
23. Crain, T., Gramoli, V., Raynal, M.: No hot spot non-blocking skip list. In: ICDCS. (2013)
24. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12** (July 1990) 463–492
25. Herlihy, M., Lev, Y., Luchangco, V., Shavit, N.: A simple optimistic skiplist algorithm. In: SIROCCO. (2007) 124–138
26. Raynal, M.: Concurrent Programming: Algorithms, Principles, and Foundations. Springer (2013)