N4037: Non-Transactional Implementation of Atomic Tree Move

Doc. No.: WG21/N4037
Date: 2014-05-26
Reply to: Paul E. McKenney
Email: paulmck@linux.vnet.ibm.com

May 26, 2014

1 Introduction

Concurrent search trees are well understood [9, 2], as are concurrent search trees that use lightweight read-side synchronizations mechanisms [10, 11, 17, 16, 7, 1, 8] such as read-copy update (RCU) [14, 12].

However, non-transactional-memory-based algorithms that atomically move an element from one search tree to another, while avoiding delaying lockless readers, are lacking. Such algorithms are known for hash tables [18, 19]. A challenge to find such an algorithm for trees was put forward at the 2014 C++ Standards Committee meeting at Issaquah, WA USA, and this document describes one solution. As such, it is a work in progress: Future work will implement multiple solutions and compare their performance and scalability.

A legitimate solution must meet a number of requirements:

1. Concurrent move operations involving elements that are not near each other in either the source or destination search tree must proceed without contention.

2. A reader that sees the element in its destination tree must fail to find it during any subsequent search of the source tree.

3. A reader that fails to find the element in the source tree must find it in any subsequent search of the destination tree.

4. A given move operation should proceed without contention even given concurrent insertion, deletion, or other modifications, as long as these other operations are not near the move operation in either the source or the destination tree.

5. A given move operation should proceed without contention even given concurrent relaxed search operations, even if the search operations are looking for the element being moved.

6. A given move operation should proceed without contention even given concurrent non-relaxed search operations, but only when the search operations are not near the move operation in either the source or the destination tree.

Use of a single global lock fails to meet the "proceed without contention" requirements, as does the per-data-structure lock approach. Of course, hardware transactional memory techniques can be used to elide these locks in some cases, but the current limitations of transactional-memory implementations prevent such elision from being carried out in the general case, particularly in the presence of non-irrevocable operations.

Two-phase locking results in contention on the root/header data element, also failing the "proceed without contention" requirements. It is possible to combine lockless search methods through the bulk of the data structure with two-phase locking for the area of interest, but this approach exposes the locking state to the caller and complicates the required API.

The solution presented in this paper can easily be generalized to cover moving multiple elements, and also to multiple types of data structures, including atomically moving an element from one type of data structure to another. For example, this approach can be used to atomically move an element from a search tree to a linked list.

The remainder of this paper is disorganized as follows: Section 2 gives a very brief overview of related work and a rationale for the choice of algorithm, Section 3 provides a conceptual overview of the chosen solution, Section 4 walks through selected portions of the code, Section 5 outlines a few advantages and disadvantages of this approach, and Section 6 provides concluding remarks.

2 Related Work and Rationale

There have been a great many tree algorithms put forward over the decades, but the slide set clearly showed a binary search tree, so radix trees, B trees, B* trees, 2-3 trees, and many others were excluded from consideration.

The Bonsai tree put forward by Clements et al. [1] was seriously considered, but rejected because it propagates balancing information to the root of the tree on each and every update, which conflicts with the requirement that concurrent atomic moves make mutual forward progress despite involving a common tree. And indeed, while this paper permitted multiple concurrent non-conflicting readers, it only permitted one updater to operate on a given tree at any given point in time.

Serious consideration was also given to AVL and Red-Black trees, but the rebalancing actions potentially taken on each update limit concurrency [10]. There is also some concern about the interactions between concurrent nearby

rebalancing operations. At this point, efficient and scalable rebalancing was excluded from consideration, though it is clearly an important topic for future work.

A number of concurrent tree algorithms were rejected because they relied on transactional memory [5, 4, 7, 8], which was excluded from consideration based on the nature of the challenge.

This forced the choice of a simple binary search tree without rebalancing. Concurrency considerations led to the restriction that data lives only at the leaves of the tree, which increases the probability that the node footprints of concurrent updates will not overlap.

3 Conceptual Overview of Solution

Section 3.1 gives an introduction to the concept of data-element allegiance, Section 3.2 overviews how to implement an atomic move, Section 3.3 looks at several implementations of data-element allegiance, and Section 3.4 considers sychronization design.

3.1 Data-Element Allegiance Concept

The solution is to introduce the notion of element *allegiance*, so that a given element can physically reside in two data structures at once, but logically be part of only one of them. Then if the element's allegiance can be altered atomically, it will be seen to move atomically from one enclosing data structure to another. Although allegiance could be indicated by any number of tokens or identifiers, we will instead use the address of the enclosing data structure. For example, in the case of a search tree, this address might that of the root of that tree, which in the case of a linked list, this address might be that of the list's header. Addresses can be written atomically on most modern systems, where "atomically" in this case means that concurrent reads of the location being written will see either the old value or the new value, but not a "mash-up" of the two values. This allows a single store instruction to atomically switch a given element's allegiance.

3.2 Atomic-Move Conceptual Overview

An atomic move is carried out using the following steps:

1. Allocate an allegiance structure, which is then initialized to the source data structure. Associate the allegiance structure with the data element to be moved.

2. Make a copy of the element to be moved. In cases where the element cannot be copied, introduce a level of indirection. This allows the pointer to the element to be copied, thus allowing external pointers to that element to be maintained throughout the atomic move operation.

3. Associate the allegiance structure with the copy.

4. Insert the copy into the destination data structure. Because the copy's allegiance is still to the source data structure, any search for the copy in the destination data structure will fail.

5. Update the allegiance structure to the destination data structure. At this point, searches for the element in the source data structure will start failing, while searches for the element in the destination data structure will start succeeding.

6. Delete the original element from the source data structure and deallocate it.

7. Disassociate the allegiance structure from the destination element and deallocate it.

These steps are illustrated in Figure 1, with "step 0" being the initial state of both data structures.

An allegiance structure could be permanently associated with each element, in which case move-time allocation and deallocation is unnecessary. However, we instead do move-time allocation and deallocation because it allows for more efficient allegiance checking in the common case where a given element is not being moved. In addition, this reduces memory overhead in the case where atomic moves are rare.

We will also consider approaches that restrict a given data element to being linked from at most one particular linked data structure. This data element is then either in or out, but if it is in, the identity of the data structure that it is a member of is implicit in the linkage from that data structure to this particalar data element. This allows complex multi-element atomic operations to be set up more simply. It also allows a given data element to be moved (via either copying or indirection) from one place to another within the same data structure. As we will see, it also permits a simple solution to the bank-transfer problem.

Note that the deallocation of the allegiance structure and the original element must be deferred in order to avoid disrupting concurrent readers. This deferral may be carried out via any convenient deferred-destruction mechanism, including garbage collectors, reference counters, hazard pointers, or RCU. It seems unlikely that many readers will be surprised to learn that RCU was chosen for this effort [13].

3.3 Data-Element Allegiance Implementations

This section describes three types of allegiance implementations, with the first being best when all data elements have the same source data structure and the same destination data structure, the second being required when the data elements are to move among several different data structures, and the third using implicit allegiance conditioned by an in-or-out indicator.

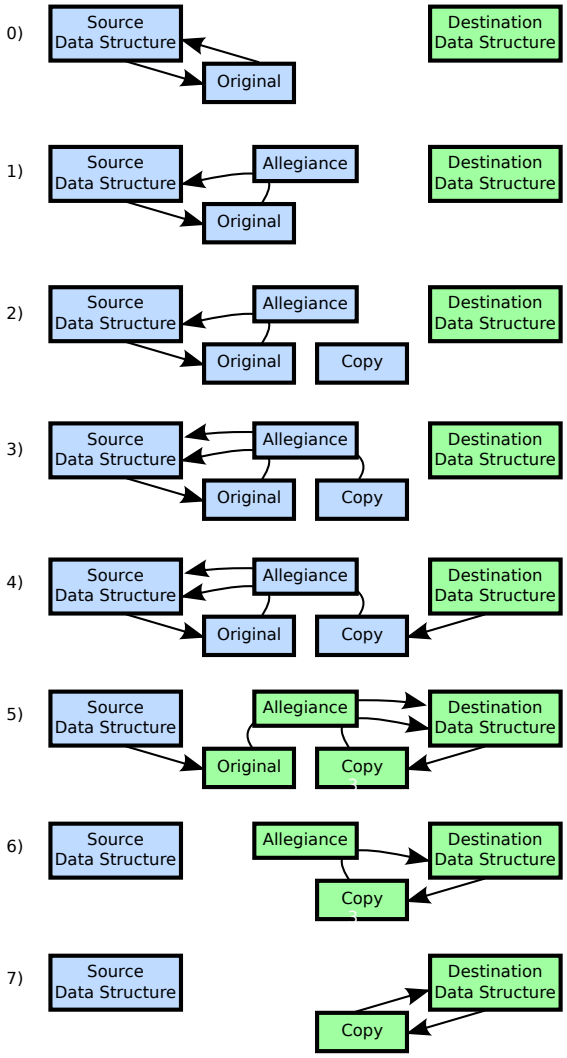

Figure 1: Conceptual Allegiance-Mediated Move

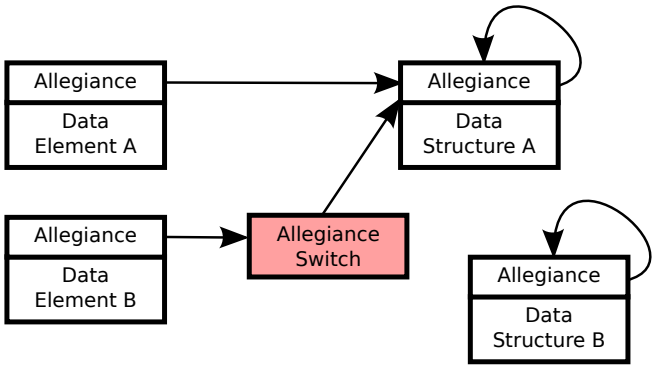

Figure 2: Single Source/Destination Allegiance Structure

3.3.1 Single Source/Destination Allegiance Implementations

In the same-source/same-destination case, each data structure has a self-referencing allegiance pointer, illustrated by the box labelled "Allegiance" above both Data Structure A and Data Structure B.

A data element's allegiance linkage can take on one of two forms. The first form, exemplified by Data Element A in Figure 2, directly references the associated data structure's allegiance pointer. This is the normal state of data elements that are not in the process of being moved. The second form, which is used to atomically move elements, is exemplified by Data Element B in the figure. Here, the data element's allegiance pointer references an allegiance switch, which is a location that can be overwritten in order to atomically switch the allegiance of a group of data elements, in this case, from Data Structure A to Data Structure B.

Note that the self-referencing nature of the allegiance pointers of the two data structures means that software need not distinguish between the two forms of data-element allegiance. In both cases, the software can proceed as follows:

1. Fetch the data element's allegiance pointer. In the first form, the result is a pointer to the enclosing data structure's allegiance field, while in the second form, the result is a pointer to the allegiance switch.

2. Dereference the pointer fetched in the preceding step. In the first form, the result is still a pointer to the enclosing data structure's allegiance field, and in the second form, the result is also a pointer to the enclosing data structure's allegiance field.

One useful optimization is to first compare the data element's allegiance pointer to the pointer to the suspected enclosing data structure's allegiance field. This optimization can eliminate cache misses that might otherwise be incurred by repeatedly fetching the enclosing data structure's allegiance field.

Given this optimization, it would in some cases be possible to make the data structure's allegiance fields be `NULL` pointers rather than self-referencing pointers, however, doing so prevents quick allegiance switches in overlapping data structures. For example, if several tree root data structures referenced the same set of nodes, then allegiance switches could provide the appearance of switching among these several trees, but without the need to undertake actual insertion, deletion, or rebalancing operations. We therefore use self-referencing allegiance fields for the enclosing data structures.[1]

This implementation fits into the atomic-move conceptual procedure described in Section 3.2, with the allegiance switch in Figure 2 taking the role of Section 3.2's allegiance structure.

Note that any number of data elements may reference the same allegiance structure, in which case all of them will atomically change allegiance simultaneously. The procedure described in Section 3.2 takes advantage of this property to simultaneously change the allegiance of the original data element and its copy, effecting an atomic move from the source data structure to the destination data structure. This can easily be elaborated in order to atomically move multiple data elements, as long as they are all moving from the same source and to the same destination.

3.3.2 Multiple Source/Destination Allegiance Implementations

Atomically moving multiple data elements among multiple data structures requires a couple additional levels of indirection, as shown in Figure 3. The first additional level of indirection is the allegiance/offset structure that selects the offset of the pointer from the offset table. This offset table is the second additional level of indirection.

Note that it is possible to place the offset directly into the data element, but doing so requires that expensive read-side memory barriers be used both when reading and updating the data element's allegiance pointer and offset. This overhead will typically only be acceptable only when the common case is optimized, that is where the data element's allegiance pointer directly references the enclosing data structure's allegiance field. However, an alternative optimization places the allegiance offsets, the allegiance switch, and the offset tables into the same memory block. This alternative optimization minimizes the impact of the separate allegiance-offset pointers, and furthermore reduces the memory footprint of the data elements. We therefore use this alternative optimization, so that the data layout is as shown in the figure.

As with the single source/destination scheme discussed in the previous section, the self-referencing nature of the allegiance pointers if the two data structures means that software need distinguish between the steady-state case represented by Data Element A and the in-motion case represented by Data Element B. In both cases, software can proceed as follows:

[1] However, frequent testing with `NULL`-valued allegiance fields for the enclosing data structures is a valuable validation measure.

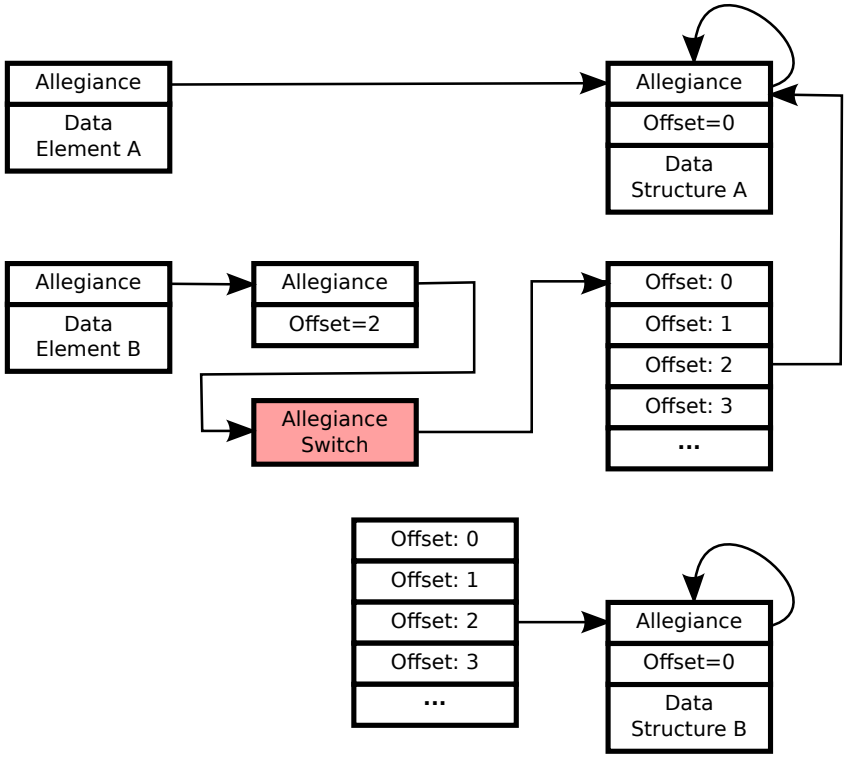

Figure 3: Multple Source/Destination Allegiance Structure

1. Fetch the data element's allegiance pointer. In the first form, the result is a pointer to the enclosing data structure's allegiance field, while in the second form, the result is a pointer to the allegiance-offset structure.

2. Given the allegiance pointer from the previous step, fetch the offset and the pointer to the allegiance switch. In the first form, the offset is zero and the pointer still references the enclosing data structure's allegiance field, while in the second form, the pointer references the allegiance switch.

3. Given the allegiance switch pointer from the previous step, fetch the allegiance switch. In the first form, the pointer still references the enclosing data structure's allegiance field, while in the second form it references the current offset table.

4. Given the offset table pointer from the previous step and the offset from the step before that, fetch the ultimate allegiance pointer. In the first form, the offset was zero, so the result still references the enclosing data structure's allegiance field, while in the second form it is the allegiance pointer, which at long last references the allegiance field of the enclosing data structure.

Given the large number of levels of indirection, the optimization of first comparing the data element's allegiance pointer to the address of the suspected enclosing data structure's allegiance field is especially valuable. The merits of a `NULL` pointer in this allegiance field are similar to those for single source/destination allegiance implementations. Also, as before, this implementation fits into the atomic-move conceptual procedure described in Section 3.2, but with the combination of the allegiance offsets, allegiance switch, and offset tables playing the role of the allegiance structure. Steps 2-4 and 6-7 of that procedure must of course be carried out for each data element being moved.

3.3.3 In-Or-Out Allegiance Implementations

An example layout for the in-or-out implementation is shown in Figure 4. A data element will normally have a `NULL` allegiance pointer, indicating that it is a member of the data structure that it is linked into. Therefore, if Data Elements A and B were to be considered to be in Data Structures A and B, respectively, both of their allegiance-pointer fields would be `NULL`.

Instead, Data Element A is leaving its data structure, and Data Element B is entering its data structure. Each therefore links to an allegiance-offset structure, each of which links to a common allegiance switch, which links two one of two allegiance-state arrays. As shown in the diagram, Data Element A has offset zero, which indexes the zero-valued element of the allegiance-state array currently indexed by the allegiance switch. This zero value indicates that Data Element A is currently to be considered a member of its data structure, namely Data Structure A. Similarly, Data Element B has offset one, which indexes the `-ENOENT`-valued element of the allegiance-state array currently indexed by the

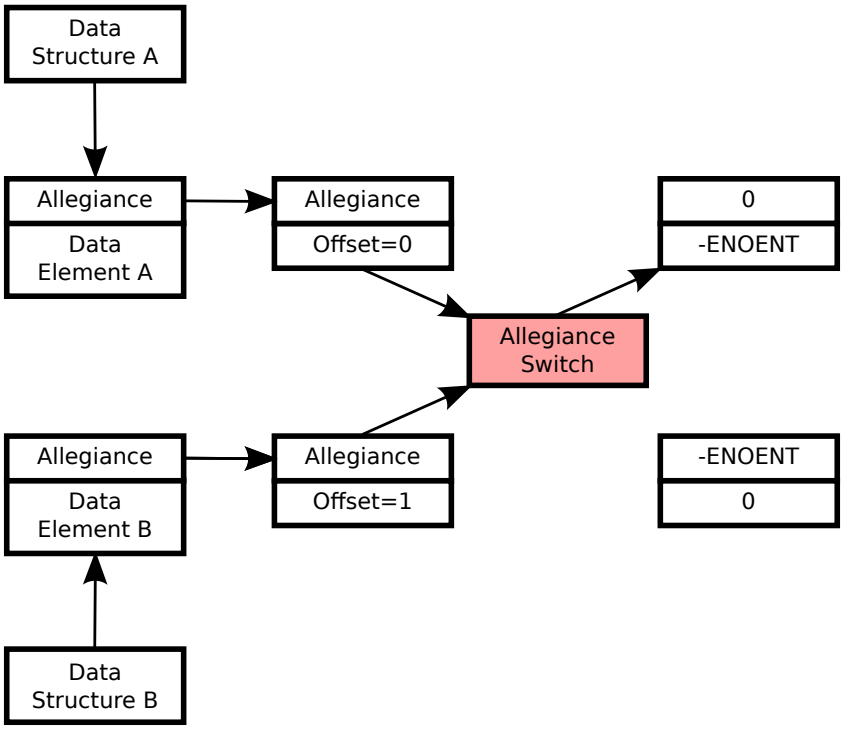

Figure 4: In-Or-Out Allegiance Structure

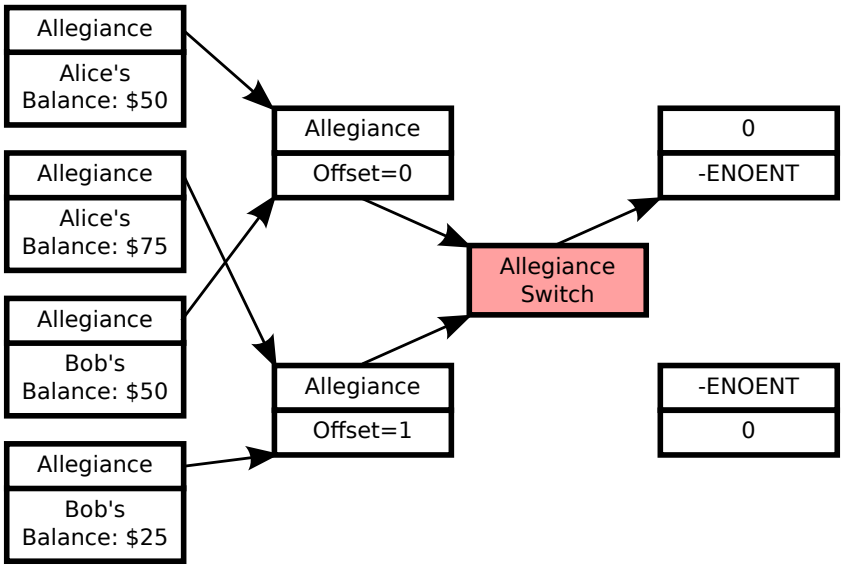

Figure 5: Transfer Using In-Or-Out Allegiance Structure

allegiance switch, which means that B is *not* to be considered a member of Data Structure B.

Updating the allegiance switch to reference the allegiance-state array at the lower right will atomically swap the status of the two data elements, in other words, it will atomically remove Data Element A and add Data Element B. Once this switch has taken place, Data Element A may be unlinked from Data Structure A and Data Element B's allegiance pointer may be set to `NULL`. The five allegiance structures are then available for reuse once all pre-existing traversals through them have completed.

Suppose that we want to use in-or-out allegiance structures to implement an atomic bank-balance transfer of $25 from Bob to Alice. This could be set up as shown in Figure 5. The records containing the old balances are linked to the upper allegiance-offset structure and those containing the new balances are linked to the lower allegiance-offset structure. The allegiance switch can then be updated to reference the allegiance array at the lower right, which atomically increases Alice's balance to $75 and decreases Bob's balance to $25.

The in-or-out allegiance approach is therefore capable of effecting surprisingly comprehensive data-structure changes, albeit at the expense of added copy operations. The amount of copying can of course be reduced by segmenting the data elements, permitting piecewise replacement/update of each data element.

Alert readers will note that this in-or-out allegiance approach bears some similarity to greatly simplified versions of some object-based software transactional memory (OSTM) commit mechanisms [4]. This simplification is made

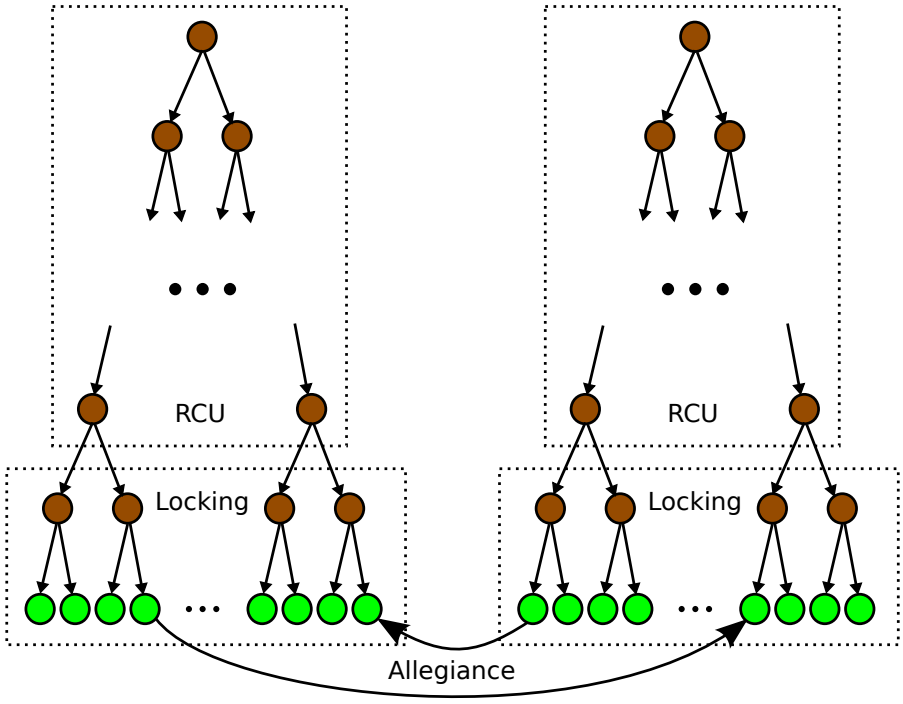

Figure 6: Locking Regions

possible by bringing multiple synchronization mechanisms to bear on this problem, and placing each such mechanism into a role for which it is well suited.

3.4 Synchronization Considerations

Although partitionable data structures can scalably use fully locked updates, attempts to take this approach with trees or linked lists will result in debilitating bottlenecks on the locks associated with the root or header nodes. Therefore, concurrent updates, including concurrent atomic moves, clearly require that locking be augmented with some other synchronization mechanism.

Fortunately, updates to these data structures are typically localized, so that the update can be divided into a search phase that locates the area to be updated and the actual update itself. We therefore use read-friendly synchronization mechanisms such as RCU to protect the search and locking to protect the update. Because the data structure can change while the locks are being acquired, there is also a validation step once the locks are acquired. If the validation step fails, the locks are released and the update is retried from the beginning. This division of responsibility between locking and RCU is depicted in Figure 6: Insertion and deletion can be carried out by locking at most the affected leaf

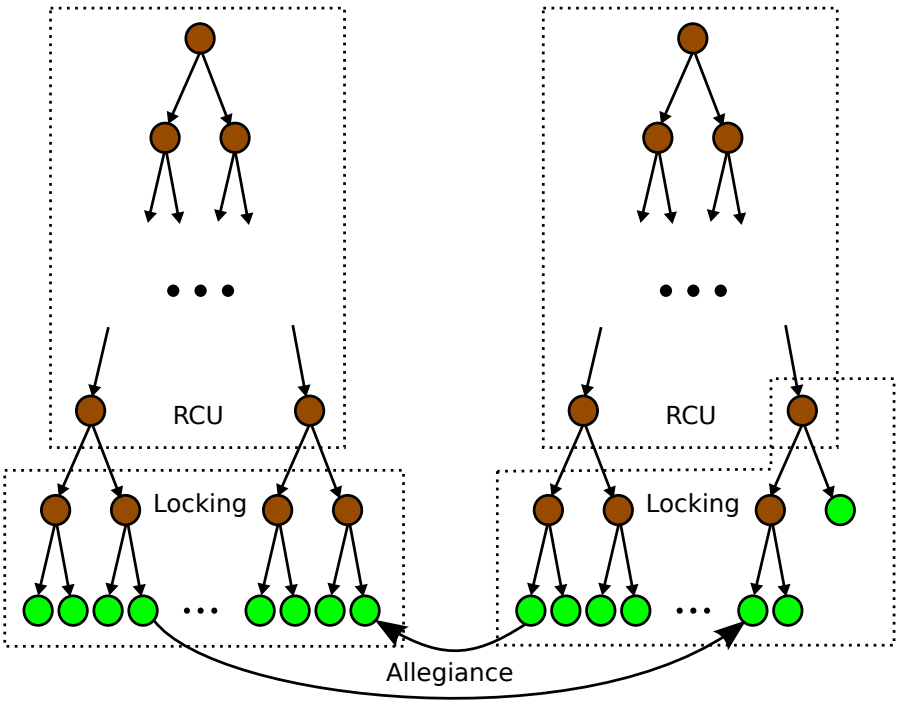

Figure 7: Locking Regions Overlap For Unbalanced Trees

Linux Kernel	C11
`smp_mb()`	`atomic_thread_fence(memory_order_seq_cst)`
a=`ACCESS_ONCE(x)`	a=`atomic_load_explicit(&x,memory_order_relaxed)`
`ACCESS_ONCE(x)=a`	`atomic_store_explicit(&x,a,memory_order_relaxed)`
`rcu_dereference(x)`	`atomic_load_explicit(memory_order_consume)`

Table 1: Approximate Correspondence Between Linux Kernel and C11

node and its parent. However, if the tree is unbalanced, the locking and RCU regions can overlap, as shown in Figure 7, where the rightmost non-leaf node is protected by RCU for updates to its left-hand descendants and by locking for updates to its right-hand descendants.

One strength of the procedure outlined in Section 3.2 is that locks need not be held across steps, which greatly simplifies usage and deadlock considerations. Nevertheless, evaluation of approaches based on two-phase locking, which would hold locks across steps, is important future work.

4 Code Walkthrough

This code walkthrough focuses on the atomic-move aspects of the invention. The source code is available for those wishing to dig more deeply.

Section 4.1 describes relevant data structures, Section 4.2 covers the atomic-move functions, Section 4.3 looks at allegiance, Section 4.4 examines lookup, and Section 4.5 covers insertion and deletion.

Note that the code uses Linux-kernel primitives. These map roughly to C11 primitives as shown in Table 1. I do apologize for inflicting this alternate syntax on the committee, and future versions will use C11.

4.1 Data Structures

Figure 8 shows the C-language data structures used in the example implementation. These provide a binary search tree, but as noted earlier, this approach works on other linked data structures as well.

The `allegiance` structure is shown on lines 1-4 of the figure. This is the simple single source/destiation version, so it consists only of a pointer and an `rcu_head` structure used for deferred freeing of the structure, which is necessary to avoid disrupting concurrent readers.

Each `treenode` structure, shown on lines 6-17 of the figure, represents a node in the search tree. This has a key, left child, right child, and pointer to user data, as is conventional for a binary search tree. Line 11 shows the allegiance pointer, and line 12 shows the eventual intended allegiance. These fields allow other operations to detect that a given node is coming or going, so that these other operations will refrain from acting on the transient node. The `->deleted` flag

```
 1 struct allegiance {
 2   void *allegiance;
 3   struct rcu_head rh;
 4 };
 5
 6 struct treenode {
 7   int key;
 8   struct treenode *lc;
 9   struct treenode *rc;
10   void *data;
11   void **allegiance;
12   void *newallegiance;
13   int deleted;
14   spinlock_t lock;
15   struct rcu_head rh;
16   int mark;
17 };
18
19 struct treeroot {
20   void *allegiance;
21   struct treenode *r;
22   spinlock_t lock;
23 };
```

Figure 8: Data Structures

on line 13 allows other operations to detect a lockless race with node deletion. These fields are all protected by the `->lock` field. Line 15 defines the `rcu_head` structure used for the deferred deletion that is necessary to avoid disrupting concurrent readers, and finally the `->mark` field defined on line 16 is used for internal consistency checking.

The `treeroot` structure, shown on lines 19-23 of the figure, represents the root of the tree. This contains a self-referencing `->allegiance` pointer on line 20, a pointer to the root node on line 21, and the lock used to guard these fields on line 22.

4.2 Atomic Move

The `tree_atomic_move()` function, which orchestrates an atomic move, is shown in Figure 9. Line 4 allocates an allegiance structure having allegiance to the source data structure, exemplified in this case by a binary search tree. If line 8 determines that line 4's allocation attempted failed, line 9 returns `-ENOMEM` to indicate memory-allocation failure. Lines 10 and 11 then invoke `tree_change_allegiance_start()` to associate the newly allocated allegiance structure with the data element corresponding to the specified key. (Another option is to also pass in a data pointer in order to distinguish among potentially many data elements having the same key.) If line 12 finds that `tree_change_allegiance_start()` returns non-zero, indicating an error, line 13 transfers to label `free_ret` to free up the unused allegiance structure and return the failure indication to the caller. Failure causes include the source data structure containing no such element and some other thread having already initiated a move for this data element.

```
 1 int tree_atomic_move(struct treeroot *srcp, struct treeroot *dstp,
 2                       int key, void **data_in)
 3 {
 4   struct allegiance *ap = alloc_allegiance(&srcp->allegiance);
 5   void *data;
 6   int ret, ret1;
 7
 8   if (ap == NULL)
 9     return -ENOMEM;
10   ret = tree_change_allegiance_start(srcp, key, ap,
11               &dstp->allegiance, &data);
12   if (ret)
13     goto free_ret;
14   ret = tree_insert_allegiance(dstp, key, data, &ap->allegiance, 0);
15   if (ret)
16     goto allegiance_end_ret;
17   smp_mb();
18   ACCESS_ONCE(ap->allegiance) = &dstp->allegiance;
19   smp_mb();
20   (void)tree_delete_allegiance(srcp, key, &data, &dstp->allegiance, 1);
21   (void)tree_change_allegiance_end(dstp, key, 0);
22   goto free_ret;
23
24 allegiance_end_ret:
25   (void)tree_change_allegiance_end(srcp, key, 1);
26 free_ret:
27   free_allegiance(ap);
28   if (data_in != NULL)
29     *data_in = ret ? NULL : data;
30   return ret;
31 }
```

Figure 9: Atomic Move Function

Line 14 then invokes `tree_insert_allegiance()`, which inserts the element into the destination data structure, but with allegiance to the source structure. This means that subsequent lookups in the destination structure will fail to find the newly inserted element due to allegiance mismatch. This insertion can fail for several reasons, including that an element with that key is already present in the destination structure, or that there was insufficient memory to allocate any needed intermediate nodes. Either way, if line 15 detects an error, line 16 transfers to label `allegiance_end_ret` in order to set the data element's allegiance back to normal, free, and return the error.

Lines 17-19 then atomically change the allegiance of the data element in both the source and destination structures. After this, the data element will no longer be accessible through the source structure, but will now be accessible via the destination structure. Note that a pair of lookups wishing to see the move as being atomic must also be separated by a memory barrier, for example, a `memory_order_acqrel` barrier in C11 or C++11. In C11 or C++11 implementations, both sets of memory barriers could be dispensed with if each element had an associated allegiance structure at all times, as the cache-coherence ordering implicit in `memory_order_relaxed` accesses would suffice. However, such an implementation would rule out atomic movement of multiple data elements, so the we include the memory barriers.

Line 20 deletes the data element from the old structure and line 21 disassociates the allegiance structure from the data element remaining in the new structure. Any errors from the deletion indicate bugs such as some other thread removing a structure undergoing an atomic move. Errors from the disassocation are normal, and can happen if some other thread deletes the element just after we move it. This situation is harmless: The other thread will deferred-free the data element, and we will deferred-free the allegiance structure.

Line 25 disassociates the allegiance structure from the source data element. This disassociation cannot fail because other operations cannot modify an element slated for a move operation.

Line 27 frees the allegiance structure, lines 28 and 29 pass back the user-data pointer, if desired, and line 30 returns status.

4.2.1 Starting The Allegiance-Change Process

Figure 10 shows the `tree_change_allegiance_start()` function, which starts the atomic move by associating a full-fledged allegiance structure with the data element to be moved.

Line 9 invokes `tree_change_allegiance_find()` locates the data element to be moved within the source tree. If line 10 determines that no such data element was available (perhaps because it is already in the process of being moved), line 11 returns the error indication to the caller.

Otherwise, line 12 assigns the allegiance structure supplied by the caller to the data element, and line 13 records its intended new allegiance. If line 14 determines that the caller wants the data element's user-supplied data pointer, line 15 assigns it to the location specified by the caller. Line 16 releases the lock

```
 1 static int tree_change_allegiance_start(struct treeroot *trp, int key,
 2                                         struct allegiance *ap,
 3                                         void *newallegiance, void **data)
 4 {
 5   struct treenode *cur;
 6   spinlock_t *lockp;
 7   int ret = 0;
 8
 9   ret = tree_change_allegiance_find(trp, key, &cur, &lockp, 0);
10   if (ret)
11     return ret;
12   cur->allegiance = &ap->allegiance;
13   cur->newallegiance = newallegiance;
14   if (data)
15     *data = cur->data;
16   spin_unlock(lockp);
17   rcu_read_unlock();
18   return ret;
19 }
```

Figure 10: Atomic Move Start

```
 1 static int tree_change_allegiance_end(struct treeroot *trp, int key, int bkout)
 2 {
 3   struct treenode *cur;
 4   spinlock_t *lockp;
 5   int ret;
 6
 7   ret = tree_change_allegiance_find(trp, key, &cur, &lockp, bkout);
 8   if (ret)
 9     return ret;
10   cur->allegiance = &trp->allegiance;
11   cur->newallegiance = &trp->allegiance;
12   spin_unlock(lockp);
13   rcu_read_unlock();
14   return 0;
15 }
```

Figure 11: Atomic Move End

that was acquired by `tree_change_allegiance_find()`, line 17 exits the RCU read-side critical section that was entered by `tree_change_allegiance_find()`, and line 18 indicates success to the caller.

4.2.2 Completing the Allegiance-Change Process

Figure 11 show the `tree_change_allegiance_end()` function, which disassociates the allegiance structure from the specified data element. Note that this element might well have been deleted after its allegiance switched to the destination tree, so it is possible for this function to fail, and such failure is OK.

Line 7 looks up the old element in the source tree, and if line 8 detects a lookup failure, line 9 returns the failure indication to the caller.

Otherwise, lines 10 and 11 reset the allegiance to the destination tree. Line 12 releases the lock that was acquired by `tree_change_allegiance_find()`, line 13 exits the RCU read-side critical section that was entered by `tree_change_allegiance_find()`, and line 14 indicates success to the caller.

```
 1 static int
 2 tree_change_allegiance_find(struct treeroot *trp, int key,
 3                              struct treenode **cur_ap,
 4           spinlock_t **lockp, int bkout)
 5 {
 6   struct treenode *cur;
 7   struct treenode *par;
 8   int ret;
 9
10 retry:
11   rcu_read_lock();
12   cur = _tree_lookup(trp, key, &par);
13   if (cur == NULL) {
14     rcu_read_unlock();
15     return -ENOENT;
16   }
17   *lockp = &cur->lock;
18   spin_lock(*lockp);
19   if (unlikely(cur->deleted)) {
20     spin_unlock(*lockp);
21     rcu_read_unlock();
22     goto retry;
23   }
24   if (!tree_leaf_node(cur) || cur->key != key) {
25     ret = -ENOENT;
26     goto unlock_ret;
27   }
28   if (bkout ||
29       (!tree_allegiance_changing(trp, cur) &&
30        !tree_wrong_cur_allegiance(trp, &trp->allegiance, cur))) {
31     *cur_ap = cur;
32     return 0;
33   }
34   ret = -EINVAL;
35 unlock_ret:
36   spin_unlock(*lockp);
37   rcu_read_unlock();
38   return ret;
39 }
```

Figure 12: Atomic Move: Allegiance-Based Lookup

4.2.3 Allegiance-Based Lookup

Figure 12 shows the `tree_change_allegiance_find()` function, which looks up a node in the specified tree with the specified key, but with the specified allegiance, which might not match that of the tree. This special-case lookup allows finding the node that was inserted into the destination tree, but with the allegiance to the source tree.

Line 11 enters an RCU read-side critical section in order to prevent any of the tree's nodes from being freed while undertaking a lockless traversal. Line 12 looks up the specified node, ignoring allegiance, returning a pointer to the desired node, and also placing a pointer to the node's parent in local variable `par`. If line 13 sees that `_tree_lookup()` could not find any such node, line 14 exits the RCU read-side critical section and line 15 returns to the caller.

Line 17 then records the address of the current node's lock for the benefit of the caller, who is responsible for releasing it. Line 18 acquires this lock. Line 18 checks to see if the current node has been concurrently deleted, and if so, line 20

```
 1 static int tree_allegiance_changing(struct treeroot *trp, struct treenode *cur)
 2 {
 3   return get_cur_allegiance(get_allegiance(&trp->allegiance),
 4                             ACCESS_ONCE(cur->allegiance)) !=
 5          cur->newallegiance;
 6 }
 7
 8 static int tree_wrong_cur_allegiance(struct treeroot *trp,
 9              void *allegiance, struct treenode *cur)
10 {
11   return allegiance !=
12        get_cur_allegiance(get_allegiance(&trp->allegiance),
13                           ACCESS_ONCE(cur->allegiance));
14 }
15
16 void *get_cur_allegiance(void *p, void **ap)
17 {
18   if ((void *)ap == p)
19     return p;
20   return rcu_dereference(*ap);
21 }
```

Figure 13: Allegiance Helper Function

releases the lock, line 21 exits the RCU read-side critical section, and line 22 branches back to line 10 to retry the lookup.

Otherwise, the node has not been deleted. Line 24 then checks that the current node is in fact a leaf with the correct key[2] If not, line 25 sets the error code to `-ENOENT` to indicate that there is no data element to move and and line 26 transfers control to line 35 to clean up and return.

Otherwise, the node is a leaf with the correct key. Line 28 therefore checks whether this is a backout operation in response to a failed move and lines 29 and 30 check whether the allegiance is unchanging adn the allegiance is what was specified by the caller. If so, line 31 stores the node address for the caller's benefit and line 32 returns indicating success. Note that in the case of success, this function returns with the node's lock held within an RCU read-side critical section. It is the caller's responsibility to release this lock and exit the RCU read-side critical section.

Otherwise, although the node is a leaf with the correct key, it's allegiance is either changing or wrong and this is not a backout operation. Line 34 therefore sets the return value to `-EINVAL` to indicate the error and control falls through to the cleanup code. Line 36 releases the lock, line 37 exits the RCU read-side critical section, and line 38 returns to the caller.

4.3 Allegiance Implementation

Figure 13 shows the allegiance helper functions for the single source/destination variant. The `tree_allegiance_changing()` function on lines 1-6 of the figure compares the `treenode` structure's current allegiance (via `get_cur_`

[2] The `_tree_lookup()` function also serves for insertion, in which case the `cur` and `par` variables will find not the node, but rather the place in the tree to insert the node.

```
 1 static struct treenode *_tree_lookup(struct treeroot *trp, int key,
 2                                       struct treenode **parent)
 3 {
 4   struct treenode *cur;
 5   struct treenode *l;
 6   struct treenode *next;
 7   struct treenode *par = NULL;
 8   struct treenode *r;
 9
10   cur = rcu_dereference(trp->r);
11   if (cur == NULL) {
12     *parent = NULL;
13     return NULL;
14   }
15   for (;;) {
16     *parent = par;
17     l = rcu_dereference(cur->lc);
18     r = rcu_dereference(cur->rc);
19     if (cur->key == key && l == NULL && r == NULL)
20       return cur;
21     if (key <= cur->key) {
22       if (l == NULL)
23         return cur;
24       par = cur;
25       cur = l;
26     } else {
27       if (r == NULL)
28         return cur;
29       par = cur;
30       cur = r;
31     }
32   }
33   /* NOTREACHED */
34 }
```

Figure 14: Lookup Helper Function

`allegiance()`) with its `->newallegiance` field, returning `true` if these differ. The `tree_wrong_cur_allegiance` function on lines 8-14 of the figure instead compares the `treenode` structure's allegiance with the caller-specified allegiance.

The `get_cur_allegiance()` function on lines 16-21 of the figure first does a fast-path check for the expected allegiance on lines 18 and 19, and if that check fails, fetches the allegiance on line 20.

4.4 Lookup

The implementation of lookups is quite typical. The following code walkthrough will therefore only point out differences from a typical tree-lookup implementation.

4.4.1 Lookup Helper Function

The only non-typical code in the `_tree_lookup()` function shown in Figure 14 are the `rcu_dereference()` invocations on lines 10, 17, and 18. The `rcu_dereference()` primitive constrains both the compiler and the CPU to avoid misorderings that might otherwise allow the subsequent code to see pre-initialization

values of the fields referenced off of the returned pointer [12, 3, 15]. These constraints allow an RCU reader to operate correctly in the face of concurrent updates that insert new data elements into the tree. In a sequentially consistent environment, `rcu_dereference()` would simply return the value of its argument.

Other than the use of `rcu_dereference()`, the `_tree_lookup()` function is a straightforward loop-based binary tree search function.

4.4.2 Lookup Function

Figure 15 shows the `tree_lookup()` function. This function uses `_tree_lookup()` to locklessly search the specified tree for the specified key, then acquires the node's lock and checks allegiance. If these checks succeed, it invokes the user-supplied function, which might acquire locks on the caller-supplied data referenced by this node.

Line 10 enters an RCU read-side critical section and line 11 does a lockless lookup. If line 12 determines that the lookup failed, line 13 exits the RCU read-side critical section, and line 14 returns a `NULL` pointer to the caller as a failure indication.

Otherwise, `_tree_lookup()` found a node. Line 16 records the address of this node's lock and line 17 acquires it. If line 18 finds that the node has been deleted, line 19 branches out in order to retry the lookup.

Otherwise, if line 20 finds that the node is either not a leaf or does not have the desired key, line 21 sets the return value to `NULL` and line 22 jumps down to clean up and return. On the other hand, if line 24 determines that the node's allegiance will soon be changing, line 25 branches out in order to retry after undergoing an appropriate delay that gives the conflicting allegiance-change operation a chance to complete. If the allegiance is not changing, line 26 checks allegiance and if it does not match that of the enclosing tree, line 27 `NULL`s out the node and line 28 branches out in order to clean up and return the error to the caller. Otherwise, if line 27 sees that a non-`NULL` function was specified, line 28 invokes it.

The unlock-return cleanup is on lines 33-37. Line 33 releases the lock and line 34 exists the RCU read-side critical section. If line 35 sees that the return value is `NULL`, line 36 returns `NULL`. Otherwise, line 37 returns a pointer to the user data.

The unlock-retry cleanup is on lines 39-46. This code releases the lock and exits the RCU read-side critical section, followed by an optional delay and a branch back to the beginning of the function.

4.4.3 Relaxed Lookup Function

Figure 16 shows `tree_lookup_relaxed()`, which does a point-in-time unsynchronized lookup. This function Takes the same series of decisions as does the `tree_lookup()` function shown in Figure 15, but avoids the locking.

```
 1 void *tree_lookup(struct treeroot *trp, int key, void (*func)(void *data))
 2 {
 3   void *allegiance = &trp->allegiance;
 4   struct treenode *cur;
 5   struct treenode *par;
 6   spinlock_t *lockp = NULL;
 7   int wantdelay = 0;
 8
 9 retry:
10   rcu_read_lock();
11   cur = _tree_lookup(trp, key, &par);
12   if (cur == NULL) { /* Empty tree */
13     rcu_read_unlock();
14     return NULL;
15   }
16   lockp = &cur->lock;
17   spin_lock(lockp);
18   if (cur->deleted)
19     goto unlock_retry;
20   if (!tree_leaf_node(cur) || cur->key != key) {
21     cur = NULL;
22     goto unlock_ret;
23   }
24   if (tree_allegiance_changing(trp, cur))
25     goto unlock_retry_delay;
26   if (tree_wrong_cur_allegiance(trp, allegiance, cur)) {
27     cur = NULL;
28     goto unlock_ret;
29   }
30   if (func)
31     func(cur->data);
32 unlock_ret:
33   spin_unlock(lockp);
34   rcu_read_unlock();
35   if (cur == NULL)
36     return NULL;
37   return cur->data;
38 unlock_retry_delay:
39   wantdelay = 1;
40 unlock_retry:
41   spin_unlock(lockp);
42   rcu_read_unlock();
43   if (wantdelay)
44     poll(NULL, 0, 1);
45   wantdelay = 0;
46   goto retry;
47 }
```

Figure 15: Lookup Function

```
 1 void *tree_lookup_relaxed(struct treeroot *trp, int key)
 2 {
 3   struct treenode *cur;
 4   struct treenode *par;
 5
 6   cur = _tree_lookup(trp, key, &par);
 7   if (cur == NULL || cur->deleted)
 8     return NULL;
 9   if (tree_leaf_node(cur) && cur->key == key &&
10       !tree_wrong_cur_allegiance(trp, &trp->allegiance, cur)) {
11     return cur->data;
12   }
13   return NULL;
14 }
```

Figure 16: Relaxed Lookup Function

4.5 Insertion and Deletion

4.5.1 Insertion

The `tree_insert_allegiance()` function is shown in Figures 17 and 18. This is a typical binary search-tree insertion, with a few changes required to accommodate the lockless search and the allegiances. It uses the `_tree_lookup()` helper function, and determines the insertion strategy based on the value of the current node (`cur`) and the parent node (`par`).

If `cur` is `NULL`, insertion into an empty tree is handled by lines 18-26 of Figure 17. The only embellishment over a textbook tree insertion is the check on lines 22-23, which handles the case where a concurrent insertion rendered the tree non-empty while the needed lock was being acquired by line 21. In this case, the search is retried, hopefully avoiding concurrent interference on the next try.

If `cur` is non-`NULL`, but `par` is `NULL`, the insertion involves the (existing) root node of the tree, which is handled by lines 27-54. This code fragment requires embellishments for both check-after-lock and allegiance. Lines 31 and 32 check to see if the root node was deleted or if some other node was interposed between the root node and the `treeroot` data structure while the locks were being acquired by line 30, and if so the search is retried, again, hopefully avoiding concurrent interference. The `spin_lock_mult()` function avoids deadlock by sorting the list of locks into address order before attempting to acquire any of them. The allegiance checks are in `insert_check_allegiance()` on line 35, which is executed in the case that a leaf node with the same key is already present in the tree. If this pre-existing leaf node has the enclosing tree's allegiance and its allegiance is not slated to change, an `-EEXIST` error is returned to the caller, otherwise the search is retried after a delay—unless the `wait` parameter was specified, in which case `-EBUSY` is returned.

If both `cur` and `par` are non-`NULL`, the insertion is to take place mid-tree, which is handled by lines 55-78 of Figure 18. This code fragment also requires embellishments for both check-after-lock and allegiance. The lock-check code on lines 58 and 59 retries if, while line 57 was acquiring the locks, either `cur` or `par`

```
 1 int tree_insert_allegiance(struct treeroot *trp, int key, void *data,
 2          void **node_allegiance, int wait)
 3 {
 4   struct treenode *cur;
 5   struct treenode *new = NULL;
 6   struct treenode *newint;
 7   struct treenode *old;
 8   struct treenode *par;
 9   int ret = 0;
10   spinlock_t *lockp[2];
11   int wantdelay = 0;
12
13   BUG_ON(data == NULL);
14   new = alloc_treenode(trp, key, data, node_allegiance);
15 retry:
16   rcu_read_lock();
17   cur = _tree_lookup(trp, key, &par);
18   if (cur == NULL) {
19     lockp[0] = &trp->lock;
20     lockp[1] = NULL;
21     spin_lock_mult(lockp, 1);
22     if (trp->r != NULL)
23       goto unlock_retry;
24     rcu_assign_pointer(trp->r, new);
25     goto unlock_ret;
26   }
27   if (par == NULL) {
28     lockp[0] = &trp->lock;
29     lockp[1] = &cur->lock;
30     spin_lock_mult(lockp, 2);
31     if (cur->deleted || trp->r != cur)
32       goto unlock_retry;
33     if (tree_leaf_node(cur)) {
34       if (cur->key == key) {
35         ret = insert_check_allegiance(trp, cur, wait);
36         if (ret == -EAGAIN)
37           goto unlock_retry_delay;
38         goto unlock_ret;
39       }
40       newint = new_internal_node(trp, key, node_allegiance,
41                cur, new);
42       if (!newint) {
43         ret = -ENOMEM;
44         goto unlock_ret;
45       }
46       rcu_assign_pointer(trp->r, newint);
47       goto unlock_ret;
48     }
49     if ((key <= cur->key && cur->lc != NULL) ||
50         (key > cur->key && cur->rc != NULL))
51       goto unlock_retry;
52     ret = parent_insert(trp, key, cur, new);
53     goto unlock_ret;
54   }
```

Figure 17: Insertion Function, 1 of 2

```
55   lockp[0] = &par->lock;
56   lockp[1] = &cur->lock;
57   spin_lock_mult(lockp, 2);
58   if (par->deleted || cur->deleted || (par->lc != cur && par->rc != cur))
59     goto unlock_retry;
60   if (tree_empty_node(cur)) {
61     if (replace_leaf(trp, cur, par, new))
62       goto unlock_retry;
63     goto unlock_ret;
64   }
65   if (tree_leaf_node(cur)) {
66     if (cur->key == key) {
67       ret = insert_check_allegiance(trp, cur, wait);
68       if (ret == -EAGAIN)
69         goto unlock_retry_delay;
70       goto unlock_ret;
71     }
72     ret = parent_insert(trp, key, par, new);
73     goto unlock_ret;
74   }
75   if ((key <= cur->key && cur->lc != NULL) ||
76       (key > cur->key && cur->rc != NULL))
77         goto unlock_retry;
78   ret = parent_insert(trp, key, cur, new);
79
80 unlock_ret:
81   spin_unlock_mult(lockp, 2);
82   rcu_read_unlock();
83   if (ret != 0)
84     free_treenode_cache(new);
85   return ret;
86
87 unlock_retry_delay:
88   wantdelay = 1;
89 unlock_retry:
90   spin_unlock_mult(lockp, 2);
91   rcu_read_unlock();
92   if (wantdelay) {
93     poll(NULL, 0, 1);
94     wantdelay = 0;
95   }
96   ret = 0;
97   goto retry;
98 }
```

Figure 18: Insertion Function, 2 of 2

were deleted on the one hand, or if `cur` is no longer an immediate descendant of `par`. The allegiance checks on line 67 operates similarly to those in the insert-at-root case discussed previously.

A one-liner wrapper function named `tree_insert()` invokes `tree_insert_allegiance()` passing in the tree's allegiance for the `node_allegiance` parameter and `1` for the `wait` parameter.

Figure 19 shows several helper functions invoked by `tree_insert_allegiance()`. The `insert_check_allegiance()` function is shown on lines 1-14. Line 7 checks to see if the current node's allegiance is slated to change, and line 8 checks to see if the current node's allegiance does not match the current specified tree. If line 9 sees that the allegiance matches the tree and is not changing, line 10 returns `-EEXIST` to indicate that there is already a node with the desired key. Otherwise, either `-EBUSY` or `-EAGAIN`, depending on whether or not the caller is willing to wait.

The `parent_insert()` function shown on lines 16-43 matches its textbook counterpart, other than the use of `rcu_assign_pointer()` on lines 24, 28, and 41 in place of the conventional assignment statement. The purpose of these `rcu_assign_pointer()` invocations is to enforce the needed memory ordering to allow the concurrent lockless reads to function correctly.

The `replace_leaf()` function shown on lines 45-61 also matches its textbook counterpart, again with `rcu_assign_pointer()` replacing assignment statements, this time on lines 49 and 55.

The `new_internal_node()` function is not show, as it exactly matches its textbook counterpart. It allocates a new internal node with the specified allegiance and with the the specified pair of nodes hanging off of it.

4.5.2 Deletion

Figure 20 shows `tree_delete_allegiance()`, which deletes a node of the specified key and allegiance, returning an error code and passing back the private data through the `data` argument. This function is quite similar to its textbook counterpart, but adds checks for changes after acquiring locks. Line 23 verifies that the data element has not be deleted and is still attached to the `treeroot` structure, while line 39 verifies that neither the data element nor its parent have been deleted and that the parent still links to the data element.

There is also a `tree_delete()` function that invokes `tree_delete_allegiance()` with the tree's allegiance and with the `wait` parameter set.

Allegiance checks are carried out in `tree_delete_leaf()` which is shown lines 1-13 of Figure 21. The allegiance checks are on lines 4-7, and report `-EAGAIN` or `-ENOENT` on changing or wrong allegiance, respectively. The only other difference from a textbook algorithm is the use of `rcu_assign_pointer()` in place of assignment on line 8.

The `tree_grew()` function checks to see if the tree grew while acquiring the locks, which provokes a retry in `tree_delete_allegiance()`.

```
 1 static int
 2 insert_check_allegiance(struct treeroot *trp, struct treenode *cur, int wait)
 3 {
 4   int c;
 5   int w;
 6
 7   c = tree_allegiance_changing(trp, cur);
 8   w = tree_wrong_cur_allegiance(trp, &trp->allegiance, cur);
 9   if (!c && !w)
10     return -EEXIST;
11   if (!wait)
12     return -EBUSY;
13   return -EAGAIN;
14 }
15
16 static int parent_insert(struct treeroot *trp, int key,
17        struct treenode *par, struct treenode *new)
18 {
19   struct treenode *newint;
20   struct treenode *old;
21   struct treenode **curp;
22
23   if (par->lc == NULL && key <= par->key) {
24     rcu_assign_pointer(par->lc, new);
25     return 0;
26   }
27   if (par->rc == NULL && key > par->key) {
28     rcu_assign_pointer(par->rc, new);
29     return 0;
30   }
31   if (key <= par->key) {
32     old = par->lc;
33     curp = &par->lc;
34   } else {
35     old = par->rc;
36     curp = &par->rc;
37   }
38   newint = new_internal_node(trp, key, par->allegiance, old, new);
39   if (!newint)
40     return -ENOMEM;
41   rcu_assign_pointer(*curp, newint);
42   return 0;
43 }
44
45 static int replace_leaf(struct treeroot *trp, struct treenode *cur,
46       struct treenode *par, struct treenode *new)
47 {
48   if (par->lc == cur) {
49     rcu_assign_pointer(par->lc, new);
50     cur->deleted = 1;
51     call_rcu(&cur->rh, tree_rcu_free_cb);
52     return 0;
53   }
54   if (par->rc == cur) {
55     rcu_assign_pointer(par->rc, new);
56     cur->deleted = 1;
57     call_rcu(&cur->rh, tree_rcu_free_cb);
58     return 0;
59   }
60   return -EINVAL;
61 }
```

Figure 19: Insertion Helper Functions

```
 1 static int tree_delete_allegiance(struct treeroot *trp, int key,
 2           void **data, void *allegiance, int wait)
 3 {
 4   struct treenode *cur;
 5   struct treenode **p;
 6   struct treenode *par;
 7   spinlock_t *lockp[2];
 8   int ret = -ENOENT;
 9   int wantdelay = 0;
10
11 retry:
12   rcu_read_lock();
13   cur = _tree_lookup(trp, key, &par);
14   *data = NULL;
15   if (cur == NULL) {
16     rcu_read_unlock();
17     return ret;
18   }
19   if (par == NULL) {
20     lockp[0] = &trp->lock;
21     lockp[1] = &cur->lock;
22     spin_lock_mult(lockp, 2);
23     if (cur->deleted || trp->r != cur)
24       goto unlock_retry;
25     if (tree_leaf_node(cur) && cur->key == key) {
26       ret = tree_delete_leaf(trp, &trp->r, cur,
27                 data, allegiance);
28       if (ret == -EAGAIN) {
29         if (!wait)
30           goto unlock_ret;
31         goto unlock_retry_delay;
32       }
33     }
34     goto unlock_ret;
35   }
36   lockp[0] = &par->lock;
37   lockp[1] = &cur->lock;
38   spin_lock_mult(lockp, 2);
39   if (par->deleted || cur->deleted || (par->lc != cur && par->rc != cur))
40     goto unlock_retry;
41   if (tree_empty_node(cur))
42     goto unlock_ret;
43   if (!tree_leaf_node(cur) || cur->key != key) {
44     if (tree_grew(cur, key))
45       goto unlock_retry;
46   } else {
47     if (par->lc == cur)
48       p = &par->lc;
49     else if (par->rc == cur)
50       p = &par->rc;
51     else
52       goto unlock_retry;
53     ret = tree_delete_leaf(trp, p, cur, data, allegiance);
54     if (ret == -EAGAIN && wait)
55       goto unlock_retry_delay;
56   }
57 unlock_ret:
58   spin_unlock_mult(lockp, 2);
59   rcu_read_unlock();
60   return ret;
61 unlock_retry_delay:
62   wantdelay = 1;
63 unlock_retry:
64   ret = -ENOENT;
65   spin_unlock_mult(lockp, 2);
66   rcu_read_unlock();
67   if (wantdelay) {
68     poll(NULL, 0, 1);
69     wantdelay = 0;
70   }
71   goto retry;
72 }
```

Figure 20: Deletion Function

```
 1 static int tree_delete_leaf(struct treeroot *trp, struct treenode **p,
 2           struct treenode *cur, void **data, void *allegiance)
 3 {
 4   if (tree_allegiance_changing(trp, cur))
 5     return -EAGAIN;
 6   if (tree_wrong_cur_allegiance(trp, allegiance, cur))
 7     return -ENOENT;
 8   rcu_assign_pointer(*p, NULL);
 9   cur->deleted = 1;
10   call_rcu(&cur->rh, tree_rcu_free_cb);
11   *data = cur->data;
12   return 0;
13 }
14
15 static int tree_grew(struct treenode *cur, int key)
16 {
17   return (key <= cur->key && cur->lc != NULL) ||
18          (key > cur->key && cur->rc != NULL);
19 }
```

Figure 21: Deletion Helper Functions

5 Advantages and Drawbacks

5.1 Reliability

This code is quite new, so although it passes significant stress testing, it likely contains numerous bugs. I do *not* yet recommend its use in production.

5.2 Ease of Use

Although allegiance-mediated update does require some modifications to standard algorithms, these changes are localized and fit well within textbook algorithms. For example, the overall flow of the tree update and search operations remained the same, but with allegiance-based code added at strategic intervals. In addition, and more important, the API for normal tree operations remains unchanged. In particular, lock acquisitions and releases are confined to individual operations, greatly simplifying deadlock avoidance compared to approaches such as two-phase locking where the locking state leaks out.

Nevertheless, allegiance-mediated update is clearly more complex than a sequential sequential implementation, so we clearly should be looking for some sort of return on the incremental investment of time and effort. The next section therefore takes a quick look at performance and scalability.

5.3 Performance and Scalability

Figure 22 shows the performance and scalability of an operation mix that is 90% lookups, 6% insertions and deletions, 3% full tree scans, and 1% moves, operating on a tree containing 256 elements. This operations mix follows Gramoli et al. [6]. Nearly linear scalability is achieved in this mix, with a throughput of 10,091 operations per millisecond at eight CPUs vs. an ideal throughput of 10,650 based on the single-CPU throughput.

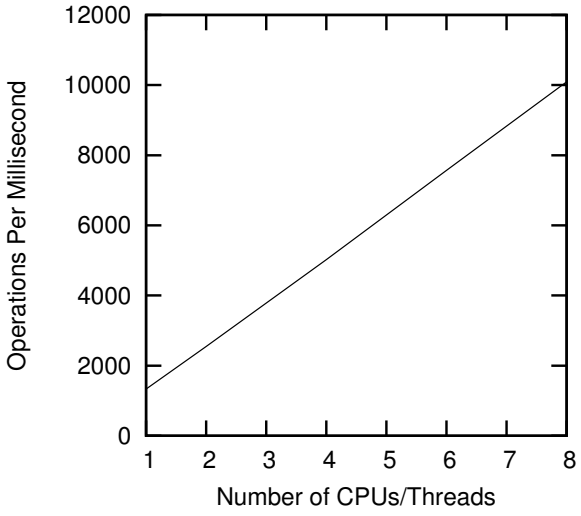

Figure 22: Performance and Scalability of Mixed Operations

Figure 23 focuses strictly on the move operation, again on a 256-element tree. Although this update-only workload does not scale as well as the read-mostly workload, its eight-CPU throughput is a semi-respectable 3.7x that of a single CPU. Note that achieving this result required a fully parallel random-number generator with per-thread state as well as (crude) per-thread caches imposed over the system implementation of `malloc()`. Further tuning would likely uncover other bottlenecks, both in the system library and in the atomic-move implementation itself.

5.4 Other Issues

Because allegiance-mediated update increases the number of memory-allocation operations, a high-quality memory allocator is critically important to its performance and scalability. Similarly, high-quality read-mostly synchronization mechanisms are also critically important.

The allegiance operations need a formal API in order to make their usage easier and less error-prone.

6 Summary

This paper has demonstrated a prototype solution to the Issaquah challenge of atomically moving data between two search trees without unnecessary contention.

Future work includes efficiently balancing the trees, evaluating other non-TM implementations, and comparing against TM implementations.

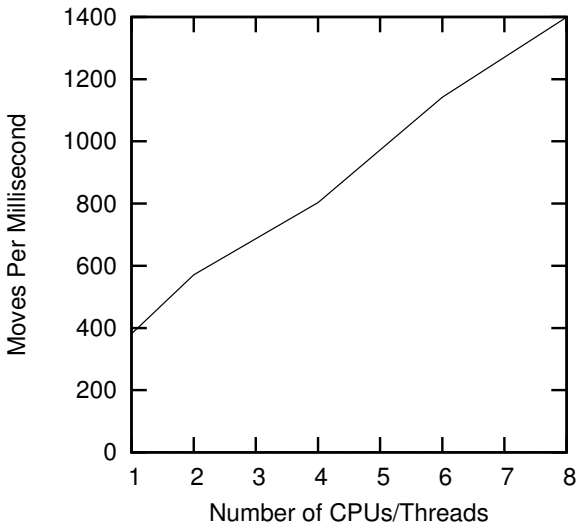

Figure 23: Performance and Scalability of Moves

References

[1] Clements, A., Kaashoek, F., and Zeldovich, N. Scalable address spaces using RCU balanced trees. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS 2012)* (London, UK, March 2012), ACM, pp. 199–210.

[2] Cormen, T., Leiserson, C., Rivest, R., and Stein, C. *Introduction to Algorithms, Second Edition*. MIT electrical engineering and computer science series. MIT Press, 2001.

[3] Desnoyers, M., McKenney, P. E., Stern, A., Dagenais, M. R., and Walpole, J. User-level implementations of read-copy update. *IEEE Transactions on Parallel and Distributed Systems 23* (2012), 375–382.

[4] Fraser, K., and Harris, T. Concurrent programming without locks. *ACM Trans. Comput. Syst. 25*, 2 (2007), 1–61.

[5] Fraser, K. A. *Practical Lock-Freedom*. PhD thesis, King's College, University of Cambridge, 2003.

[6] Gramoli, V., and Guerraoui, R. Democratizing transactional programming. *Commun. ACM 57*, 1 (Jan. 2014), 86–93.

[7] Howard, P. W., and Walpole, J. A relativistic enhancement to software transactional memory. In *Proceedings of the 3rd USENIX conference on Hot topics in parallelism* (Berkeley, CA, USA, 2011), HotPar'11, USENIX Association, pp. 1–6.

[8] Howard, P. W., and Walpole, J. Relativistic red-black trees. *Concurrency and Computation: Practice and Experience* (2013), n/a–n/a.

[9] Knuth, D. *The Art of Computer Programming*. Addison-Wesley, 1973.

[10] Kung, H. T., and Lehman, Q. Concurrent manipulation of binary search trees. *ACM Transactions on Database Systems 5*, 3 (September 1980), 354–382.

[11] Manber, U., and Ladner, R. E. Concurrency control in a dynamic search structure. *ACM Transactions on Database Systems 9*, 3 (September 1984), 439–455.

[12] McKenney, P. E. Structured deferral: synchronization via procrastination. *Commun. ACM 56*, 7 (July 2013), 40–49.

[13] McKenney, P. E. Structured deferral: Synchronization via procrastination. *Queue 11*, 5 (May 2013), 20:20–20:39.

[14] McKenney, P. E., and Slingwine, J. D. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems* (Las Vegas, NV, October 1998), pp. 509–518.

[15] McKenney, P. E., and Walpole, J. What is RCU, fundamentally? Available: `http://lwn.net/Articles/262464/` [Viewed December 27, 2007], December 2007.

[16] Piggin, N. [patch 3/3] radix-tree: RCU lockless readside. Available: `http://lkml.org/lkml/2006/6/20/238` [Viewed March 25, 2008], June 2006.

[17] Pugh, W. Concurrent maintenance of skip lists. Tech. Rep. CS-TR-2222.1, Institute of Advanced Computer Science Studies, Department of Computer Science, University of Maryland, College Park, Maryland, June 1990.

[18] Triplett, J., McKenney, P. E., and Walpole, J. Scalable concurrent hash tables via relativistic programming. *ACM Operating Systems Review 44*, 3 (July 2010).

[19] Triplett, J., McKenney, P. E., and Walpole, J. Resizable, scalable, concurrent hash tables via relativistic programming. In *Proceedings of the 2011 USENIX Annual Technical Conference* (Portland, OR USA, June 2011), The USENIX Association, pp. 145–158.