Concurrent Updates with RCU: Search Tree as an Example[*]

(Version with Additional Proofs)

Maya Arbel
Department of Computer Science
Technion
mayaarl@cs.technion.ac.il

Hagit Attiya
Department of Computer Science
Technion
hagit@cs.technion.ac.il

ABSTRACT

Read copy update (RCU) is a novel synchronization mechanism, in which the burden of synchronization falls completely on the updaters, by having them wait for all pre-existing readers to finish their read-side critical section. This paper presents CITRUS, a concurrent binary search tree (BST) with a wait-free contains operation, using RCU synchronization and fine-grained locking for synchronization among updaters. This is the first RCU-based data structure that allows concurrent updaters. While there are methodologies for using RCU to coordinate between readers and updaters, they do not address the issue of coordination among updaters, and indeed, all existing RCU-based data structures rely on coarse-grained synchronization between updaters.

Experimental evaluation shows that CITRUS beats previous RCU-based search trees, even under mild update contention, and compares well with the best-known concurrent dictionaries.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Concurrent programming*; F.1.2 [**Computation By Abstract Devices**]: Modes of Computation—*Parallelism and concurrency*

General Terms

Algorithms, Architecture

Keywords

Shared memory; internal search tree; read-copy-update

1. INTRODUCTION

The traditional approach to synchronization between readers and writers allows concurrency among readers, but excludes readers when a writer is executing, e.g., through a *readers/writer lock*. An alternative, contemporary approach is presented by *read copy update* (*RCU*) [24], a distinctive synchronization mechanism favoring read-only operations even further, by allowing them to proceed even while writers are modifying the data they are reading. Instead, the *read-side critical section* is wrapped by the rcu_read_lock and rcu_read_unlock functions; an additional synchronize_rcu function can be used by a writer as a barrier ensuring that all preceding read-side critical sections complete. In typical RCU usage, the burden of synchronization is placed on updates, who can wait for all pre-existing readers to finish their read-side critical section.

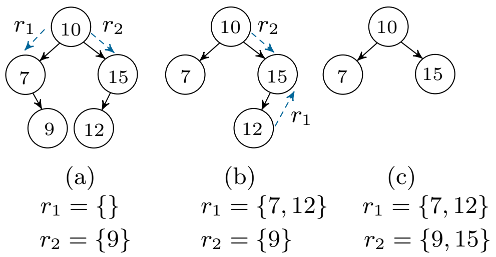
RCU is used extensively within the Linux kernel [22], mostly to facilitate memory reclamation [14, 23]. However, its potential for concurrent programming remained unexploited. Although several RCU-based data structures have been proposed, for example, search trees and hash tables [7, 20, 28, 29], they all do not allow concurrent updates, either pessimistically, using a coarse-grained lock [7], or optimistically, using transactional memory [20]. At best, the data structure is partitioned into segments, e.g., buckets in a hash table [29], each guarded by a single lock.

This leaves unanswered the question of coordination *between concurrent updates* to the data structure, while using RCU, which is the topic of this paper.

A natural approach for supporting concurrent updates is to employ fine-grained locks, acquired and released according to some locking policy, e.g., *two-phase* locking [12] or *hand-over-hand* locking [3, 27]. We found that these fine-grained approaches fail to ensure consistent views for read-only operations that access several items in the data structure, e.g., partial snapshots [2] or other iterators [26]. Figure 1 shows an example in which two readers, r_1 and r_2, attempt to collect the leaves, by in-order traversal, while two updates delete leaves 9 and 12. In (a), r_2 already traversed the left sub-tree while r_1 has not yet started its traversal. After 9 is deleted (b), r_1 finishes traversing the tree while r_2 did not take additional steps, and 12 is deleted before both readers complete (c). Since each reader may observe a different permutation of the writes to the data structure, the values returned by r_1 and r_2 are such that they observed the updates in different order. This means that without additional iterations or interaction, there is no consistent way to order the updates and the readers, complicating the task of designing a concurrent data structure.

[*]This research is supported by Yad-HaNadiv foundation and the Israel Science Foundation (grant number 1227/10).

PODC'14, July 15–18, 2014, Paris, France.
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$15.00.



(a) $r_1 = \{\}$ $r_2 = \{9\}$

(b) $r_1 = \{7, 12\}$ $r_2 = \{9\}$

(c) $r_1 = \{7, 12\}$ $r_2 = \{9, 15\}$

Figure 1: An example in which RCU readers may observe a different permutation of the writes to the data structure.

Nevertheless, this example (and others that we found) hints that the difficulty is in having read-only operations that need to atomically access several locations. The counter-examples do not hold when the read-only operation only searches for a particular data item, e.g., in a dictionary.

This observation led to the design of Citrus, a binary search tree implementing a dictionary with concurrent updates (insert and delete) and contains queries. Updates synchronize using fine-grained locks, while contains proceeds in a wait-free manner, in parallel with updates, relying on RCU to ensure correctness. The combination of RCU synchronization with fine-grained locking of modified nodes yields a simple design, greatly resembling the sequential algorithm, and leading to a (relatively) simple proof of correctness, only several pages long.

Citrus implements an *internal* search tree, with keys stored in all nodes, and it is *unbalanced.* Searching for a key, either in a contains operation or at the beginning of an update, is done in a wait-free manner, as in the sequential data structure, but inside an *RCU read-side critical section.*

Updates start by searching for their key, in a manner similar to contains. An unsuccessful update (insert that finds its key or delete that does not find its key), returns immediately. Otherwise, the update is located where the change should be done.

A new node is inserted as a leaf, requiring little synchronization, but delete may need to remove an internal node. When the node has two children, this is done by replacing the node with its *successor* in the tree. This scenario requires coordination with concurrent searches that could miss the successor in both its previous location and in its new location, necessitating delicate synchronization in some concurrent search trees [5, 8, 10, 21]. Citrus easily circumvents this pitfall by copying the successor to the new location, and using the RCU synchronization mechanism to wait for on-going searches, before removing the successor from its previous location.

Experimental evaluation of Citrus shows that its performance beats previous RCU-based search trees [7, 20], even under mild update contention. It also compares well with the best-available concurrent dictionaries, e.g., [5, 17, 25].

The evaluation also reveals that the user-level RCU implementation [9] is ill-suited for workloads in which many updates concurrently synchronize through it. We sketch a new implementation that avoids these pitfalls and scales better with growing number of updates. In the new implementation, a thread indicates that it starts a read-side critical section by incrementing a counter and setting a flag to true; the flag is set to false at the end of the section. To synchronize, an update waits until every other thread either increases its counter or sets its flag to false.

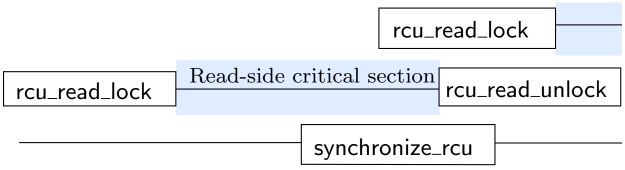


Figure 2: The semantics of synchronize_rcu

2. PRELIMINARIES

A concurrent system consists of a set of threads, communicating by applying *primitives* to shared variables. A shared data structure *implementation* provides a set of operations, each invoked with possible parameters and returning with a response. The invocation of an operation is a local step of a thread, leading to the execution of an *algorithm*, directing the thread to execute a sequence of *atomic* steps. Each atomic step is either an instance of a shared memory primitive or computation on variables that are local to the thread. Returning from an operation is a local computation step.

A *configuration* is an instantaneous snapshot of the system describing the state of all local and shared variables. In the *initial* configuration, all variables hold an initial value.

An *execution* π is an alternating sequence of configurations and steps, $C_0, s_1, ..., s_i, C_i, ...$, where C_0 is the initial configuration, and each configuration C_i is the result of executing the step s_i in configuration C_{i-1}. A *prefix* σ of π is a sub-sequence of π starting in C_0 and ending with a configuration. An *interval* of π is a sub-sequence that starts with a step and ends with a configuration. The *interval of an operation* op starts with the invocation step of op and ends with the configuration following the response of op or the end of π, if there is no such response.

The API for *read copy update* (RCU) provides several functions, three of which are used in Citrus to synchronize between readers and updates: rcu_read_lock, rcu_read_unlock and synchronize_rcu. A *read-side critical section* is the interval starting with the step returning from rcu_read_lock and ending with the configuration after the invocation of rcu_read_unlock. The implementations of rcu_read_lock and rcu_read_unlock must be wait-free. The *RCU property* (Figure 2) ensures that if a step of a read-side critical section precedes the invocation of synchronize_rcu, then all steps of this critical section precede the return from synchronize_rcu [14, 15].

A *dictionary* is a set of key-value pairs, with totally ordered keys, with the following operations:

insert(k,val) adds (k, val) to the set; returns true if k is not in the set and false otherwise.

delete(k) removes (k, val) from the set; returns true if k is in the set and false otherwise.

contains(k) returns val if (k, val) is in the set; otherwise, it returns false.

An *update* is either an insert or a delete.

A binary search tree (Figure 3) implements a dictionary; it is *internal* if key-value pairs are stored in all nodes. A node v

of the tree stores a key, $Key(v)$, which never changes. Two dummy key values $-1, \infty$ are used to avoid corner cases, when the tree has fewer than two nodes; for every key k, $-1 < k < \infty$. The root of the tree always points to a node with key -1, this node has a right child with key ∞; all other nodes are in the left sub-tree of ∞.

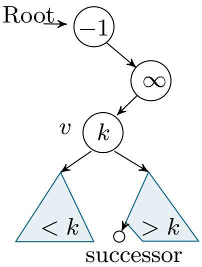


Figure 3: Search tree with dummy nodes.

In a *binary search tree* (BST), all descendants in the left sub-tree of v have keys smaller than the key of v, and all descendants in the right sub-tree of v have keys larger than the key of v.

The *successor* of node v is the node u with the smallest key among the nodes with keys larger than or equal to $Key(v)$. In a BST, u is the leftmost node in the right sub-tree of v.

3. THE CITRUS TREE ALGORITHM

Overview. A primary goal of CITRUS is to avoid locking when searching for a node, either in contains or at the beginning of an update. This is implemented in an auxiliary procedure get, which starts at the root and searches down the tree in a manner similar to the sequential algorithm, except that it is performed inside a read-side critical section, wrapped with rcu_read_lock and rcu_read_unlock.

A contains simply invokes get to find the key; if the key is found, it returns the value stored in the node returned by get; otherwise, it returns false.

An insert invokes get, and returns false if get finds the key. Otherwise, a new node with the key is inserted as a leaf, added to the tree as the child of the last node in get's search.

A delete invokes get, and returns false if get does not find the key. If the key is found in node v, then there are two cases. If v has has at most one child, the node is removed by redirecting the child field of v's parent to point to v's child (see Figure 4(a) and (b)); later in the proof this is called *bypassing*. Otherwise, v has two children and it is replaced with its successor in the tree, which is stored in the leftmost node in the right sub-tree of v.

Moving the successor, when v has two children, requires coordination with a concurrent get searching for the successor, which may return a false negative response (Figure 5). To overcome this problem, delete first inserts a copy of the successor node in the deleted key's location. Then the delete waits for concurrent searches by executing synchronize_rcu, and only then, removes the old successor node. Searches that start before synchronize_rcu starts, find the successor in its previous location, where it remains until they complete (Figure 4(c)-(e)). Searches that start after synchronize_rcu starts, find the new copy of the successor.

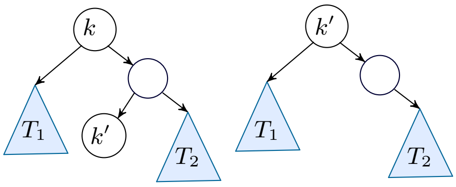


Figure 5: Search for key k' returns a false negative response due to an overlapping delete.

Note that during a delete, there might be two copies of the successor—in the original and in the new locations. This motivates the following *weak BST* (WBST) property (formally defined in Definition 1): all descendants in the left sub-tree of v have keys smaller than the key of v, and all descendants in the right sub-tree of v have keys larger than or equal to the key of v.

The WBST property allows multiple nodes with the same key. If all nodes with the same key hold the same value, preserving the WBST property ensures that contains is correct, as it may return the value of some duplicate node, and ignore the others.

Updates synchronize among themselves by acquiring locks on nodes returned by get. To avoid RCU deadlocks, locks are acquired outside the read-side critical section. This creates a region of uncertainty, for example, if an insert reaches the node with key k, but an overlapping delete operation removes this node from the tree before insert locks it (see Figure 6). If insert continues its operation, the new key will not be part of the tree.

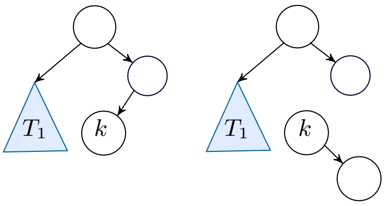
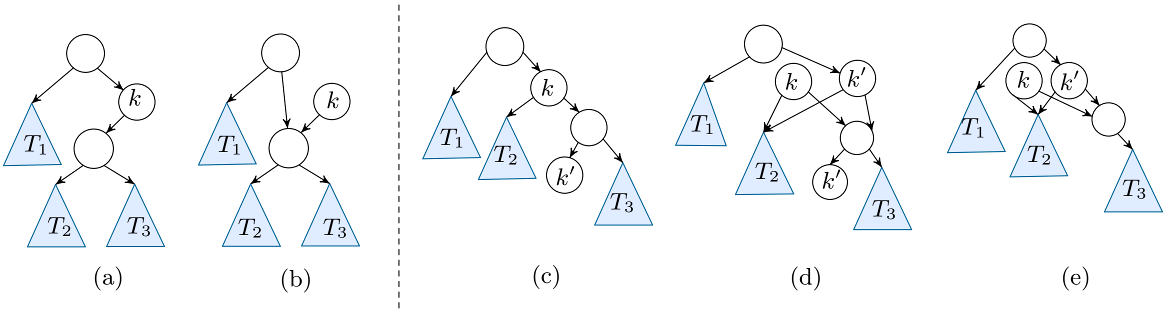


Figure 6: insert can add a node to an incorrect location due to an overlapping delete.

We overcome this problem by *validating* the nodes after locking them, and restarting the operation if validation fails. An update restarts either because the locked nodes no longer have a parent-child relation or because one (or both) of the two nodes was deleted from the tree. Validating the parent-child relation is done locally by checking the child pointer of the parent node. When the operation validating is an insert, the child pointer of the parent should be $\perp$ (null pointer). A *tag* is added to each child field, in order to avoid an ABA problem (a child pointer changed to non-$\perp$ when a leaf is inserted, then back to $\perp$, when the leaf is moved by a delete). (A total of two tag fields in each node, one for each child.) A tag field is initialized to zero, and incremented every time the corresponding child field is set to $\perp$. Validating that a node was not removed is also done locally, using a *marked* field indicating that the node was deleted (in a manner similar to [16]). If validation fails, the operation releases all locks and restarts from the root.



(a) (b) (c) (d) (e)

Figure 4: (a) and (b) show delete(k) when a node has one child. (c)-(e) show delete(k) that replaces a node with two children with its successor; synchronize_rcu is invoked between (d) and (e).

Detailed description. All operations use procedure get to search for a key k, which starts from the root, and returns two nodes, curr and prev, which is the parent of curr (Line 15). The procedure also returns a *direction*, indicating whether curr is the left or right child of prev, and the tag of prev that is associated with direction. If the key k exists in the sub-tree rooted at start, then curr is the node containing k. If the key does not exist, curr $=\perp$ and prev is the last node found in the search. All operations executed by get are wrapped with rcu_read_lock and rcu_read_unlock creating a read-side critical section (Lines 2 and 14).

Citrus get function

```
 1 function get(key)
 2   rcu_read_lock
 3   prev ← root
 4   curr ← prev.child[right]                    ▷ root is never ⊥
 5   currentKey ← curr.key ▷ root's right child is never ⊥
 6   direction ← right
 7   while curr $\neq \perp$ and currentKey $\neq$ key do
 8     prev ← curr
 9     direction ← (currentKey > key ? left : right)
10     curr ← prev.child[direction]
11     if curr $\neq \perp$ then
12       currentKey ← curr.key
13   tag = prev.tag[direction]
                  ▷ Save tag inside read-side critical section
14   rcu_read_unlock
15   return (prev,tag,curr,direction)
```

The contains operation simply invokes get. If the key is not in the tree, it returns false (Line 19). If the key is found, it returns the value associated with it (Line 20).

Citrus contains function

```
16 function contains(key)
17   (-,-,curr,-) ← get(key)
18   if curr $= \perp$ then              ▷ The key was not found
19     return false
20   return curr.value
```

The insert operation invokes get, if the key is found, it returns false (Line 25). If the key is not in the tree, insert locks prev (Line 26) and then validates it (Line 27). If validation fails, the operation starts over; otherwise, the new node is created (Line 28) and added to the tree (Line 29).

Citrus insert function

```
21 function insert(key,value)
22   loop
23     (prev,tag,curr,direction) ← get(key)
24     if curr $\neq \perp$ then               ▷ The key was found
25       return false
26     lock(prev)
27     if validate(prev,tag,$\perp$,direction) then
28       node ← new(key,value,$\perp$,$\perp$)
                                   ▷ Create a new leaf node
29       prev.child[direction] ← node
30       unlock(prev)
31       return true
32     unlock(prev)
                    ▷ Validation failed, release locks and retry
```

The validate procedure is a simple check of local properties of the given nodes.

Citrus validate function

```
33 function validate(prev,tag,curr,direction)
34   if prev.marked or prev.child[direction]$\neq$curr then
35     return false
36   if curr $\neq \perp$ then
                  ▷ If curr $\neq \perp$, validate curr's marked bit
37     return !curr.marked
38   return prev.tag[direction] = tag
                                  ▷ Otherwise validate tag
```

The incrementTag procedure receives a node and a direction, if the child of node in this direction is $\perp$, it increments the tag associated with this direction.

Citrus incrementTag function

```
39 function incrementTag(node,direction)
40   if node.child[direction] = $\perp$ then
41     node.tag[direction] ← node.tag[direction]+1
```

The delete operation invokes get. If the key is not found, it returns false (Line 46). If the key is found, prev and curr are locked (Lines 47, 48) and validated (Line 49). If validation fails, the operation unlocks prev and curr and starts over. If curr has only one child, delete does not use the successor (Line 50), curr is marked (Line 51) and deleted (Line 53).

Citrus delete function

```
42 function delete(key)
43   loop
44     (prev,-,curr,direction) ← get(key)
45     if curr=⊥ then                        ▷ The key was not found
46       return false
47     lock(prev)
48     lock(curr)
49     if validate(prev,-,curr,direction) then
50       if curr.child[left] =⊥ or curr.child[right] =⊥ then
                                              ▷ curr has a single child
51         curr.marked ← true
52         notNoneChild ←
                      (curr.child[left] ≠⊥ ? left : right)
53         prev.child[direction] ←
                                   curr.child[notNoneChild]
54         incrementTag(prev,direction)
55         release all locks
56         return true
57       else                                 ▷ curr has two children
58         prevSucc ← curr   ▷ Searching for the successor
59         succ ← curr.child[right]
60         next ← succ.child[left]                       ▷ succ ≠⊥
61         while next ≠⊥ do
62           prevSucc ← succ
63           succ ← next
64           next ← next.child[left]
65         succDirection ← (curr = prevSucc ? right : left)
66         if curr ≠ prevSucc then   ▷ Do not lock twice
67           lock(prevSucc)
68         lock(succ)
69         if validate(prevSucc,-,succ,succDirection) and
                  validate(succ,succ.tag[left],⊥,left) then
70           node ← new(succ.key,succ.value,curr.child[left],
                                          curr.child[right])
71           lock(node)
72           curr.marked ← true
73           prev.child[direction] ← node
74           synchronize_rcu                 ▷ Wait for readers
75           succ.marked ← true
                                  ▷ Remove the old successor
76           if prevSucc = curr then
                                ▷ succ is the right child of curr
77             node.child[right] ← succ.child[right]
78             incrementTag(node,right)
79           else
80             prevSucc.child[left] ← succ.child[right]
81             incrementTag(prevSucc,left)
82           release all locks
83           return true
84     release all locks
                       ▷ Validation failed, release locks and retry
```

If curr has two children (Line 57), the operation tries to delete curr by using a successor. It finds the successor succ and its parent prevSucc, by traversing the leftmost branch of the sub-tree rooted at curr (Lines 58-64). This traversal does not need a read-side critical section since the keys of nodes traversed do not impact the direction. Once found, prevSucc and succ are locked (Lines 67, 68) and validated (Line 69). If validation fails, all nodes are unlocked and the operation starts over; otherwise, a deletion using a successor is executed (Lines 72 - 83). A new node with succ's key and curr's children is created (Line 70) and locked (Line 71). Then, curr is marked (Line 72) and replaced by node (Line 73). Next, the operation waits for all pre-existing readers, by invoking synchronize_rcu (Line 74). When the operation returns, it removes the old successor from the tree (Lines 75-80); note that succ might be the right child of curr (Line 76). The operation completes by unlocking all the nodes and returning true (Line 83). After removing a node from the tree (in one of Lines 53, 73, 77 or 80), incrementTag is called in case the tag should be updated.

4. LINEARIZABILITY OF Citrus

Fix an execution π of the Citrus algorithm. Roughly speaking, an implementation is *linearizable* [19] if it is possible to identify, for each operation, a *linearization point*, inside its interval, so that the responses of the operations are *consistent*: that is, they are the same as if the operations were performed sequentially in their linearization points.

The following notation is used in the proof. Let u, v be two nodes in the tree, $v \rightarrow u$ indicates that u is a child of v, $v \xrightarrow{left} u$ when u is the left child of v and $v \xrightarrow{right} u$ when u is the right child of v; generally, $v \xrightarrow{d} u$, $d = left$ means that u is the left child of v, and analogously if $d = right$. There is a *path* from node v to node u in configuration C, if there is a sequence of nodes $v = v_0, v_1, ..., v_m = u$, $m > 0$, such that for every $i, 0 \leq i < m$, $v_i \rightarrow v_{i+1}$ in C. We denote $\rho_C(v,u)$=$v_0, ..., v_m$. A node v is *reachable* in configuration C if there is a path from the root to v in C. A key k is reachable in configuration C if there is a reachable node v such that k =$Key(v)$; recall that $Key(v)$ never changes. The set of reachable keys stored in the sub-tree rooted at v, in a configuration C, is denoted $Set_C(v)$; $Set_C(root)$ is the set of reachable keys in the whole tree. We define the WBST property:

Definition 1. The weak BST (*WBST*) property holds *in configuration C if for every internal node v, if u is a node such that $v \xrightarrow{left} u$ and $k \in Set_C(u)$ then $k <Key(v)$ and if $v \xrightarrow{right} u$ and $k \in Set_C(u)$ then $k \geq Key(v)$*. The WBST property holds in *an execution prefix σ if it holds in every configuration C in σ.*

If the WBST property holds in an execution prefix σ that ends in configuration C, then the range of keys in the subtree of node v, denoted $Range_C(v)$, can be defined by induction on $\rho_C(root, v)$: $Range_C(root) = (-\infty, \infty)$; if the parent of v is node u with k =$Key(u)$ and $Range_C(u)$ = $[min_u, max_u)$ or $Range_C(u) = (min_u, max_u)$, then $Range_C(v) = [min_u, k)$ or $Range_C(v) = (min_u, k)$ if $u \xrightarrow{left} v$, and $Range_C(v) = [k, max_u)$ if $u \xrightarrow{right} v$.

Let u, v, w be nodes such that $u \xrightarrow{d} v \rightarrow w$, $d \in \{left, right\}$, and node v has only one child, in configuration $C \in \pi$. A primitive write operation s that immediately follows C is a *bypass* of node v if $u \xrightarrow{d} w$ in the configuration that follows s in π (Figure 4(a) and (b)). The write in Line 53 is a bypass of the node curr, and the writes in Line 77 and Line 80 are a bypass of the node succ.

In order to distinguish between variables of different operations, we denote by $\mathsf{var}_{\mathsf{op}}$ the variable var of operation op; the operation is omitted when it is clear from the context.

An operation op *accesses* a node v when op reads one of v's fields. Only updates execute primitive writes, and when an update op writes to a field of node v, v is locked by op. Therefore, the fields of node v cannot be modified by another operation until the lock on v is released.

Updates use validate to ensure that they operate on nodes in the tree. The next lemma argues that if a node is not reachable in a configuration, then its marked bit is true. Note that a node is unmarked when it is added to the tree, and marked before it is removed. Since updates operate on locked nodes, this suffices to prove the lemma, even though marking and removing a node are not performed atomically.

LEMMA 1. *If a node v is reachable in configuration $C \in \pi$ and v is unmarked in configuration $C' \in \pi$ that follows C, then v is reachable in C'.*

PROOF. Assume, by way of contradiction, that the property does not hold and let C be the last configuration in π in which the property holds. Let s be the step by operation op immediately following C; s must be a primitive write to a child field of a locked node, in an update.

Case 1: insert. Line 29: By the validation in Line 27, prev has no child in direction d (where d is the value of direction), hence, s does not make an unmarked node unreachable. Since node was created by op, it is unreachable in the configuration immediately following the step in Line 28. If s makes node reachable, then node is unmarked, as required.

Case 2: delete. By the validation in Line 49, prev $\overset{d}{\to}$ curr (where d is the value of direction).

Bypassing (Lines 53, 77, 80) makes only curr or succ unreachable. The lemma holds since curr is marked before Line 53 and succ is marked before Lines 77 and 80.

Line 73: By Line 72, curr is marked and can become unreachable. Both children of curr become the children of node in Line 70. Since s adds node to the tree as a left or right child of prev (depending on d), curr is the only node that becomes unreachable. Since node is created by op, it is unreachable in the configuration immediately following the step in Line 70. If node becomes reachable by s, then it is unmarked, as required. □

To ensure that get with key k is correct, we prove that all nodes accessed by get are in the tree at some point during its interval (Lemma 2) and that if k is in the tree throughout a search from the root, then it is found (Lemma 8). Recall that an operation op accesses a node v when op reads one of v's fields.

LEMMA 2. *Let π' be the interval of operation op. If op accesses a node v in step s, then v is reachable in some configuration $C \in \pi'$ that precedes s.*

PROOF. By induction on the length of the path taken by op from the root to node v. In the base case, the root is always reachable.

For the induction step, assume that the lemma holds for paths of length $< \ell$. Let v be the ℓ'th node in the path taken by op and let u be the $(\ell - 1)$'th node in this path. Suppose, without loss of generality, that op read $u \overset{right}{\to} v$ in step s'.

By the induction hypothesis, u is reachable in some configuration during π'; let C be the last such configuration. If s' precedes C, then v is reachable in the configuration C' that immediately follow s'. Since the pointer read in step s' precedes any access to v, C' precedes s (Figure 7(a)).

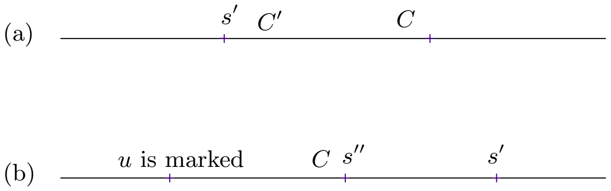


Figure 7: The proof of Lemma 2. Case (a) u is reachable when $u \to v$ is read. Case (b) u is unreachable when $u \to v$ is read

If s' does not precede C, we argue that v is the right child of u when u is removed from the tree. Suppose, by way of contradiction, that u has some other right child $w \neq v$ in C and s'' is the step removing u from the tree, i.e., the step that immediately follows C. Since op read $u \overset{right}{\to} v$, there is an update that sets the right child of u to v before s'. Since u is reachable in C and unreachable in the configuration following s'', Lemma 1 implies that u is marked before s''. The update removing u does not change u's children fields (by the code, no operation writes to child fields of curr or succ). Any other update validates all locked nodes to check that they are not marked. Since u is marked, any update that tries to write to u restarts without changing u, which is a contradiction.

When u is removed from the tree, v is the right child of u. Since C is the last configuration in π' in which u is reachable, both v and u are reachable in $C \in \pi'$, C precedes s' that precedes s, as required. □

The next lemma shows the correctness of tag validation. It shows that if a child pointer is set to $\perp$, by another operation, after the tag is read and before the node is locked, then the tag is incremented during this interval.

LEMMA 3. *Let π' be the interval starting with the step of Line 13 by op and ending with the configuration that immediately follows the lock acquisition by op. If prev$_{op}$.child[direction$_{op}$] was set to $\perp$ during the interval π' then prev$_{op}$.tag[direction$_{op}$] $\neq$ tag$_{op}$ in every configuration in the interval of op that follows π'.*

PROOF. During π', op does not hold a lock on prev$_{op}$, thus prev$_{op}$.child[direction$_{op}$] is set to $\perp$ by some other operation op'. An insert sets a child field to non $\perp$ values since new is never $\perp$, hence, op' is a delete. A delete calls incrementTag after every write to a child field, thus, if prev$_{op}$.child[direction$_{op}$] is set to $\perp$ by op' then prev$_{op}$.tag[direction$_{op}$] is incremented before op' unlocks prev$_{op}$, meaning that prev$_{op}$.tag[direction$_{op}$] is incremented during π'. This concludes the proof since op reads the value of tag$_{op}$ in line 13 that is the first step of π. □

Searches, implemented in get, follow the sequential algorithm, so it is critical to show that the tree maintains the WBST property. This is proved by induction on the prefixes of the execution π. A critical step in the proof is to show that an insert adds a node in the right location. This relies on the following lemma (Lemma 4), showing that the RCU mechanism of waiting for all pre-existing readers guarantees that if the search of an insert ends up in the wrong location, due to overlapping delete, then its validation fails.

LEMMA 4. *Let* op *be an* insert *operation with key k that successfully validates* prev *and* curr*, and let σ be the execution prefix that ends in configuration C that immediately follows the return from* validate*. If the WBST property holds in σ, then $k \in Range_C(\mathsf{prev})$.*

PROOF. If the path traversed by op is the same as $\rho_C(root, \mathsf{prev})$, then, by the WBST property, $k \in Range_C(\mathsf{prev})$. The path traversed by op is different from $\rho_C(root, \mathsf{prev})$, if it was changed by an overlapping update op'. Consider every possible write to a child field, during π', the interval of op.

Case 1: insert. Line 29: By the validation in Line 27, $\mathsf{prev}_{\mathsf{op'}}$ has no child in direction d (where d is the value of direction), Thus, op' adds a leaf to the tree, without changing the range of $\mathsf{prev}_{\mathsf{op}}$.

Case 2: delete. By the validation in Line 49, $\mathsf{prev} \overset{d}{\to} \mathsf{curr}$ (where d is the value of direction).

Bypassing (Lines 53, 77, 80) only increases the ranges of other nodes and $k \in Range_C(\mathsf{prev}_{op})$ is maintained.

Line 73: Both children of $\mathsf{curr}_{\mathsf{op'}}$ become the children of $\mathsf{node}_{\mathsf{op'}}$ in Line 70. Hence, $\mathsf{curr}_{\mathsf{op'}}$ is replaced with $\mathsf{node}_{\mathsf{op'}}$ that has the key of $\mathsf{succ}_{\mathsf{op'}}$. The node $\mathsf{succ}_{\mathsf{op'}}$ is found by traversing the leftmost branch of the sub-tree rooted at $\mathsf{curr}_{\mathsf{op'}}$. Since the WBST property holds in σ, this traversal starting at $\mathsf{curr}_{\mathsf{op'}}$ finishes at the location of the successor of $\mathsf{curr}_{\mathsf{op'}}$. Since op' accesses $\mathsf{succ}_{\mathsf{op'}}$, Lemma 2 implies that $\mathsf{succ}_{\mathsf{op'}}$ is reachable after some prefix of π ending before Line 69. By the validation in Line 69 and Lemma 1, $\mathsf{succ}_{\mathsf{op'}}$ is reachable and it has no left child, implying that $\mathsf{succ}_{\mathsf{op'}}$ is the successor of $\mathsf{curr}_{\mathsf{op'}}$. Therefore, $Key(\mathsf{succ}_{op}) = k'' \geq k'$.

If $k < k'$ or $k \geq k''$, then $k \in Range_C(\mathsf{prev}_{op})$ still holds.

Suppose otherwise that $k' \leq k < k''$ and $k \notin Range_C(\mathsf{prev}_{op})$. If the step of line 73 by op' precedes or is in the read-side critical section of op then before removing the successor $\mathsf{succ}_{\mathsf{op'}}$ from the tree, op' invokes synchronize_rcu and waits until the end of op's read side critical section, by the RCU property. Since the WBST property holds during the get of op, its search ends in $\mathsf{succ}_{\mathsf{op'}}$ ($\mathsf{succ}_{\mathsf{op'}}$ equals $\mathsf{prev}_{\mathsf{op}}$ and $\mathsf{curr}_{\mathsf{op}}$ equals $\perp$). Since op is an insert, it invokes rcu_read_unlock and locks $\mathsf{prev}_{\mathsf{op}}$ prior to validation. On the other hand, op' unlocks $\mathsf{succ}_{\mathsf{op'}}$ only after marking it in Line 75. Therefore, the validation of $\mathsf{prev}_{\mathsf{op}}$ by op finds that it is marked, and fails (Figure 8).

Otherwise, the step of Line 73 by op' is executed after the read side critical section of op and before op locks $\mathsf{prev}_{\mathsf{op}}$. If $\mathsf{prev}_{\mathsf{op}}$ equals $\mathsf{succ}_{\mathsf{op'}}$ then op' marks $\mathsf{prev}_{\mathsf{op}}$ before releasing its lock, and the validation of op fails. If not then by the WBST property the change was in the left sub-tree of $\mathsf{prev}_{\mathsf{op}}$. In this case, validation fails, because either $\mathsf{prev}_{\mathsf{op}}$.child[left] $\neq \perp$ or $\mathsf{prev}_{\mathsf{op}}$.tag[left] $\neq \mathsf{tag}_{\mathsf{op}}$ (Lemma 3). □

The WBST property now follows by a simple induction. Lemmas 1 and 2 show that the correct successor replaces a deleted node with two children (bypassing a node with one child clearly preserves the WBST property), while Lemma 4 shows that new nodes are correctly inserted.

LEMMA 5. *The WBST property holds after every prefix σ of π.*

PROOF. Assume, by way of contradiction, that the WBST property does not hold after some prefix of π. Let σ be the longest prefix of π in which the WBST property holds, ending in configuration C, and let s be the first step by operation op, that invalidates the property; op must be an update and s must be a write to a child field of a locked node. We consider all possible cases.

Case 1: insert. Line 29: The WBST property holds for σ, and by Lemma 4, $k \in Range_C(\mathsf{prev})$. Since node has the key k and is added as a child of prev, the WBST property is maintained. (The direction is correct as in the sequential algorithm.)

Case 2: delete. By the validation in Line 49, $\mathsf{prev} \overset{d}{\to} \mathsf{curr}$ (where d is the value of direction).

Bypassing (Lines 53, 77, 80) a node maintains the WBST property.

Line 73: Both children of $\mathsf{curr}_{\mathsf{op'}}$ become the children of $\mathsf{node}_{\mathsf{op'}}$, in Line 70, and $\mathsf{curr}_{\mathsf{op'}}$ is replaced with $\mathsf{node}_{\mathsf{op'}}$ that has the key of $\mathsf{succ}_{\mathsf{op'}}$. Since the WBST property holds in σ, and $\mathsf{succ}_{\mathsf{op'}}$ is found by by traversing the leftmost branch of the sub-tree rooted at $\mathsf{curr}_{\mathsf{op'}}$, the traversal ends at the location of the successor of $\mathsf{curr}_{\mathsf{op'}}$. Since op' accesses $\mathsf{succ}_{\mathsf{op'}}$, Lemma 2 implies that $\mathsf{succ}_{\mathsf{op'}}$ is reachable after some prefix of π ending before Line 69. By the validation in Line 69 and Lemma 1, $\mathsf{succ}_{\mathsf{op'}}$ is reachable and $\mathsf{succ}_{\mathsf{op'}}$ has no left child, implying that $\mathsf{succ}_{\mathsf{op'}}$ is the successor of $\mathsf{curr}_{\mathsf{op'}}$. Therefore, s replaces curr with a node that have the same key as the successor of curr, and the WBST property is maintained. □

The next lemma shows that each node has a single parent, and thus, delete makes a node unreachable in a single write to a child field. It is proved by way of contradiction, considering the first time when the property is violated.

LEMMA 6. *A node that is reachable in a configuration $C \in \pi$ has one reachable parent in C.*

PROOF. Assume, by way of contradiction, that the property does not hold and let C be the last configuration in π in which the property holds. Let s be the step by operation op immediately following C; s must be a primitive write to a child field of a locked node, in an update.

Case 1: insert. Line 29 adds a parent to node, which previously had no parent in the tree.

Case 2: delete. By the validation in Line 49, $\mathsf{prev} \overset{d}{\to} \mathsf{curr}$ (where d is the value of direction).

Bypassing (Lines 53, 77, 80): Since C is the last configuration where the property holds, the node being bypassed has at most one reachable parent in C. The bypassing operation makes the node being bypassed unreachable, and maintains the number of reachable parents for other nodes.

Line 73: Both children of curr become the children of node, which is unreachable in every configuration that precedes s, in Line 70, and curr is replaced with node (making curr unreachable), in Line 73. This does not increase the number of parents per node. □

The following lemma is used to prove that if a node v with key k remains in the tree during a get, then v can be found. The reason is that the path to v never gets longer, and that even if a node u, that used to be on the path from the root to v, is read, there is still a path from u to v.

LEMMA 7. *If there is a path from u to v in configuration $C \in \pi$ and v is reachable in a configuration $C' \in \pi$ that follows C, then there is a path from u to v in C' and $|\rho_{C'}(u, v)| \leq |\rho_C(u, v)|$.*

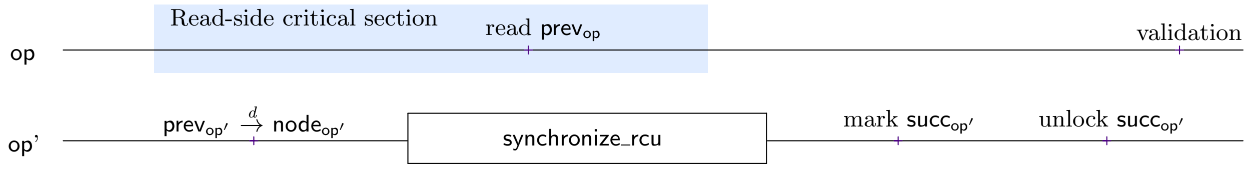

Read-side critical section
read $\mathsf{prev}_{\mathsf{op}}$
validation
op
op'
$\mathsf{prev}_{\mathsf{op'}} \xrightarrow{d} \mathsf{node}_{\mathsf{op'}}$
synchronize_rcu
mark $\mathsf{succ}_{\mathsf{op'}}$
unlock $\mathsf{succ}_{\mathsf{op'}}$

Figure 8: The execution considered in the proof of Lemma 4

Proof. Assume, by way of contradiction, that v is reachable in some configuration that follows C, but the properties are not maintained. Let C' be the last configuration in which both propertied hold. Let s be the step immediately following C'; s must be a write to a child field of a locked node, by an update op. We consider all possible cases:

Case 1: insert. Line 29: By the validation in Line 27, prev has no child in direction d (where d is the value of direction), thus, op adds a leaf to the tree, which does not change the reachability of v from u nor the length of existing paths in the tree.

Case 2: delete. By the validation in Line 49, prev $\xrightarrow{d}$ curr (where d is the value of direction).

Bypassing (Lines 53, 77, 80): By the conditions of the lemma and Lemma 6, the node being bypassed is not v (bypassing a node with a single reachable parent makes it unreachable). If the node being bypassed is u, then s does not change any child field on the path from u to v, and the path and its length are maintained. If the node being bypassed is any other node in $\rho_{C'}(u, v)$, then s only shortens the path and the lemma holds.

Line 73: Both children of curr become the children of node, in Line 70, and curr is replaced with node. Therefore, only curr may become unreachable. By Lemma 6 curr becomes unreachable, and by the conditions of the lemma $v \neq$curr. If curr $= u$, then s does not change any child field on the path from u to v, and the path and its length are maintained. If curr is any other node in $\rho_{C'}(u, v)$, and C'' is the configuration immediately following s, then node $\in\rho_{C''}(u, v)$, the path exists and its length remains the same. □

We linearize the successful updates, as follows:

- A successful insert is linearized with its primitive write in Line 29.
- A successful delete is linearized with its primitive write in Line 53 or 73. (At most one of these lines is executed.)

Let $s_1, s_2, ...$ be the primitive writes in π, in which successful updates are linearized, in their order in π. For every $i \geq 1$, let C_i be the configuration immediately preceding s_i and let C'_i be the configuration immediately following s_i. A successful insert linearized at s_i is *consistent* if $k \notin Set_{C_i}(root)$ and $k \in Set_{C'_i}(root)$, and a successful delete linearized at s_i is *consistent* if $k \in Set_{C_i}(root)$ and $k \notin Set_{C'_i}(root)$.

The next lemma is important for proving the consistency of successful inserts. It shows that if the key was inserted, by another operation, during the get of an insert then either (i) get finds the key or (ii) the key is inserted as a child of the last node in the search. Later, the first case of this lemma is used to linearize unsuccessful contains and delete.

Lemma 8. *Let π' be read-side critical section of* get *(k), and suppose that every successful update linearized before the last configuration of π' is consistent. Let C be the first configuration in π' such that $k \in Set_C(root)$ and assume that $k \in Set_{C'}(root)$, for every configuration $C' \in \pi'$ that follows C. If C precedes the last access to a child field by* get *then, (i)* get *returns* curr *such that $k = Key($*curr*$)$. Otherwise, (ii)* get *returns* prev *such that* prev $\xrightarrow{d}$ *v and $k = Key(v)$ where d is the value of* direction*.*

Proof. Let v be the closest reachable node to the root in C such that $k = Key(v)$, and let v' be the value of $\mathsf{prev}_{\mathsf{op}}$ in C or the root if C precedes Line 3 of get. since the WBST property holds in π and insert adds leaves, $v' \in\rho_C(root, v)$.

If C does not precedes the last access to a child field by get, then prev $= v' \xrightarrow{d} v$ as required.

If C precedes the last access to a child field by get, then get must reach a node with key k. Suppose otherwise that get does not reach a node with key k. If v is reachable in the last configuration of π', then the path from v' to v does not get longer (Lemma 7). Since the WBST property holds in π, get traverses this path, and hence, if a node with key k is not found, then v is unreachable in the last configuration of π'.

Let $s \in \pi'$ be the primitive write by operation op' that makes v unreachable. Let C_s be the configuration immediately preceding s in π' and let C'_s be the configuration immediately following s in π'; v is reachable in C_s and unreachable in C'_s. By the validation in Line 27, insert does not make a node unreachable, and hence, only delete can make v unreachable. Since $k \in Set_{C'_s}(root)$, op' has key $k' \neq k$. The only other way a delete makes a node unreachable is when $v =\mathsf{succ}_{\mathsf{op'}}$ and s is the primitive write of either Line 77 or Line 80.

Let $s' \in \pi'$ be the primitive write by operation op' that adds $\mathsf{node}_{\mathsf{op'}}$ to the tree. Let $C_{s'}$ be the configuration immediately preceding s' in π' and let $C'_{s'}$ be the configuration immediately following s' in π'; Since $\mathsf{succ}_{\mathsf{op'}}$ is found by traversing the leftmost branch of the sub-tree rooted at $\mathsf{curr}_{\mathsf{op'}}$, there is a path from $\mathsf{curr}_{\mathsf{op'}}$ to v in a configuration that precedes $C_{s'}$, there is a path from $\mathsf{curr}_{\mathsf{op'}}$ to v in $C_{s'}$ (Lemma 7). By Line 70, $\mathsf{node}_{\mathsf{op'}}$ has the same children as $\mathsf{curr}_{\mathsf{op'}}$, and therefore, there is a path from $\mathsf{node}_{\mathsf{op'}}$ to v in $C'_{s'}$. This means that $\mathsf{node}_{\mathsf{op'}} \in\rho_{C'_{s'}}(root, v)$, and by the choice of v, $\mathsf{node}_{\mathsf{op'}}$ is not in the tree in C. Therefore, synchronize_rcu is invoked by op' after get is invoked by op, and by the RCU property, op' executes Line 77 or Line 80 after the last configuration of π', contradicting the assumption that v is unreachable in the last configuration of π'. □

The next lemma proves that there is only one node with key k, and thus, only one node should be made unreachable in order for delete to be consistent.

Lemma 9. *Let v be a node that is successfully validated by an operation* op. *Let σ be the prefix ending with the configuration C that immediately follows the return from* validate. *If all successful updates linearized in σ are consistent, then there is no other reachable node v' such that $Key(v){=}Key(v')$.*

Proof. Since v is accessed by op during validate, Lemma 2 implies that v is reachable in some configuration that precedes C. By the validation, v is unmarked and Lemma 1 implies that v is reachable in C. Since the linearization point of insert is its first primitive write and every insert that is linearized in σ is consistent, insert does not create multiple keys.

The only other operation that can create multiple keys is a delete copying a successor. We argue that multiple keys exist only while delete holds locks on both duplicates, implying that if op locked a node v with key k, then this is the only node with key k in the tree in the configuration following the return from lock(v). Note that op invokes validate while holding the lock on v.

Let op' be delete(k') that has a successor $\mathsf{succ}_{\mathsf{op'}}$ with key k. $\mathsf{succ}_{\mathsf{op'}}$ is locked in Line 68 and $\mathsf{node}_{\mathsf{op'}}$ is locked in Line 71, before $\mathsf{node}_{\mathsf{op'}}$ is inserted into the tree (Line 73). Either Line 77 or Line 80 is executed before the locks on $\mathsf{node}_{\mathsf{op'}}$ and $\mathsf{succ}_{\mathsf{op'}}$ are released. We argue that, in both cases, op' removes $\mathsf{succ}_{\mathsf{op'}}$ from the tree. Both Line 77 and Line 80 are a bypass of $\mathsf{succ}_{\mathsf{op'}}$, which has a single parent by Lemma 6; therefore, we need to prove that $\mathsf{node}_{\mathsf{op'}}$ is reachable in the configuration immediately preceding Line 77 and $\mathsf{prevSucc}_{\mathsf{op'}}$ is reachable in the configuration immediately preceding Line 80.

Line 77: Since op' accesses $\mathsf{prev}_{\mathsf{op'}}$, Lemma 2 implies that it is reachable in some configuration preceding the step of Line 49. Furthermore, Lemma 1 implies that prev is reachable in the configuration immediately preceding Line 73, making $\mathsf{node}_{\mathsf{op'}}$ reachable in the configuration immediately preceding Line 77.

Line 80: Since op' accesses $\mathsf{prevSucc}_{\mathsf{op'}}$, Lemma 2 implies that it is reachable in some configuration preceding the step of Line 69. Furthermore, Lemma 1 implies that $\mathsf{prevSucc}_{\mathsf{op'}}$ is reachable in the configuration immediately preceding Line 80. □

Lemma 10. *All successful updates are consistent.*

Proof. By induction on the linearization points, $s_1, s_2, ...$ For every $j \geq 1$, let C_j be the configuration immediately preceding s_j and C'_j be the configuration immediately following s_j.

Base: A successful update op is linearized at its first primitive write, hence, there is no write before s_1. Let k be the key of op, and note that $k \notin Set_C(root)$ for every configuration C preceding s_1, and by Lemma 2, op cannot access a node with key k, hence, op is a successful insert. Since op accesses prev, by Lemma 2, prev is reachable in some configuration preceding the step of Line 27. Since prev is validated in Line 27 and is unmarked, Lemma 1 implies that prev is reachable in C_1 and $k \in Set_{C'_1}(root)$.

For the induction step, assume that the successful updates linearized at $s_1, s_2, ..., s_{i-1}$ are consistent, and consider the successful update op linearized at s_i.

Case 1: op is delete(k). The if statement in Line 45 implies that $k = Key(\mathsf{curr})$. By the validation in Line 49, curr and prev are both unmarked and prev $\overset{d}{\rightarrow}$curr (where d is the value of direction). Since op accesses prev and curr, Lemma 2 implies that they are reachable in some configuration preceding the step of Line 49. Furthermore, Lemma 1 implies that prev and curr are reachable in C_i. Since prev is the only reachable parent of curr (Lemma 6) curr is unreachable in C'_i. By the induction hypothesis, all successful updates linearized at $s_1, s_2, ..., s_{i-1}$ are consistent. Together with the fact that op validates curr, this implies that curr is the only reachable node with key k in C_i (Lemma 9) Therefore, $k \notin Set_{C'_i}(root)$.

Case 2: op is insert(k): Since op accesses $\mathsf{prev}_{\mathsf{op}}$, Lemma 2 implies that $\mathsf{prev}_{\mathsf{op}}$ is reachable in some configuration preceding the step of Line 27. Since $\mathsf{prev}_{\mathsf{op}}$ is validated in Line 27, it is unmarked, and Lemma 1 implies that it is reachable in C_i. Since $k = Key(\mathsf{node}_{\mathsf{op}})$, $k \in Set_{C'_i}(root)$.

Assume, by way of contradiction, that $k \in Set_{C_i}(root)$. Let $s_j, j < i$ be the last linearization point of a successful insert with key k. By the induction hypothesis, and since $k \in Set_{C_i}(root)$, no delete(k) is linearized between s_j and s_i, and for every configuration C that follows C_j and precedes C_i, $k \in Set_C(root)$.

If C'_j precedes or is in the read-side critical section of op then Lemma 8 implies that either $k = Key(\mathsf{curr}_{\mathsf{op}})$ or $\mathsf{prev}_{\mathsf{op}} \overset{d}{\rightarrow} v$ and $k = Key(v)$ (where d is the value of direction). If $k = Key(\mathsf{curr}_{\mathsf{op}})$ then $\mathsf{curr}_{\mathsf{op}} \neq \perp$ and op returns false in contradiction to op being a successful update. If $\mathsf{prev}_{\mathsf{op}} \overset{d}{\rightarrow} v$ then the validation in Line 27 fails, in contradiction to the linearization of op.

Otherwise, since the WBST property holds in π, s_j inserts the new node as the child of $\mathsf{prev}_{\mathsf{op}}$ in direction d, and the validation of op fails, because either $\mathsf{prev}_{\mathsf{op}}$.child$[d] \neq \perp$ or $\mathsf{prev}_{\mathsf{op}}$.tag$[d] \neq \mathsf{tag}_{\mathsf{op}}$ (Lemma 3). □

An operation op with interval π', of the remaining types (contains and failed updates), is linearized as follows:

- If op is a failed contains or a failed delete, then curr $=\perp$.

  If $k \in Set_C(root)$ for every configuration $C \in \pi'$, then $k \in Set_C(root)$ for every configuration $C \in \pi''$, where π'' is the read-side critical section of op. By Lemma 8(i), op reaches a node with key k, which is a contradiction. Therefore, $k \notin Set_C(root)$, for some configuration $C \in \pi'$. The linearization point of op is after last $C' \in \pi'$ such that $k \notin Set_{C'}(root)$, and before the successful insert with the same linearization point, if such insert exists. This linearization point is clearly consistent.

- If op is a successful contains that returns a node v's value or a failed insert that reached node v with key k, then it is linearized after the last configuration $C \in \pi'$ such that v is reachable in C, and before the successful delete with the same linearization point, if such delete exists.

  In op, curr $= v$, and by condition of the while loop in get (Line 7), $k = Key(v)$. By Lemma 2, curr is reachable in some configuration $C' \in \pi'$, and the linearization point exists. This linearization point is consistent since $k \in Set_C(root)$ and if op is a successful contains, it returns $v.value$.

Together with Lemma 10, this proves the linearizability of Citrus. Furthermore, if there is only a finite number of keys, then every path from the root is finite, and contains is wait-free.

THEOREM 11. *The* CITRUS *algorithm is a linearizable implementation of a binary search tree, supporting wait-free* contains.

5. EVALUATION

Setup. We implemented CITRUS in C since both the RCU implementation and the RCU-based trees are implemented in C. We considered two RCU-based trees, the *red-black* tree [20] and *Bonsai*, a balanced search tree [7], both using a global lock to synchronize among updates.[1] We also compared CITRUS to three concurrent dictionary data structures that have C implementations: The optimistic *AVL* tree[2] [5], the lock-free search tree [25] and the lock-based lazy *skiplist*[3] [17].

The experiments were run on a machine with four AMD Opteron 6376 Processors, each with 16 cores, for a total of 64 cores. All memory allocation used the `jemalloc` library, to avoid synchronization bottlenecks during memory allocation.

Experiment sets were run with two key ranges, $[0, 2 \cdot 10^5]$ and $[0, 2 \cdot 10^6]$; in both sets, the tree was pre-filled to the size of half the key range. During pre-filling memory was reclaimed. For every test, each thread ran for five seconds, continuously executing randomly chosen operations with a randomly chosen key, without performing any memory reclamation. We report the overall throughput (total number of executed operations divided by the running time). Each experiment was run five times for each configuration of operation distribution, key range and thread count, we report the arithmetic average as the final result. (The error margins, omitted for readability, are small.)

Running experiments in the unmanaged C environment revealed that memory management, and in particular cache usage, has a significant impact on performance. The size of nodes, order of fields, and their alignment inside cache lines, often influences the results more than the algorithmic aspects of the implementation.

New RCU. During our initial evaluation of CITRUS, we identified that the user-space RCU implementation [9] does not scale for workloads with many concurrent updates, due to expensive synchronization among them, which include acquiring a global lock. The left side of Figure 9 shows representative results; similar behaviour was observed under different update contention and key ranges. To show that the drop in throughput was an implementation issue, we re-implemented the subset of the RCU API used in CITRUS, in a manner similar to *epoch-based reclamation* [13]. In our implementation, each thread has a counter and flag, the counter counts the number of critical sections executed by the thread and a flag indicates if the thread is currently inside its read-side critical section. The rcu_read_lock operation increments the counter and sets the flag to true, while the rcu_read_unlock operation sets the flag to false. When a thread executes a synchronize_rcu operation, it waits for every other thread, until one of two things occurs: either the thread has increased its counter or the thread's flag is set to false. The main advantage of this implementation is that multiple threads executing synchronize_rcu need not coordinate among themselves, and they do not acquire any locks. The right side of Figure 9 shows the throughput of CITRUS with the new RCU implementation. All other experiments were run with the new RCU implementation.

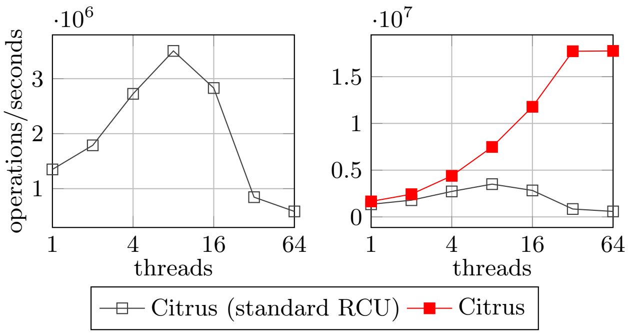


Figure 9: Impact of concurrent updates on the standard RCU implementation compared to our scalable implementation: example with operation distribution of 50% contains and key range $[0, 2 \cdot 10^5]$. Left side is a detailed view of the behaviour of the original implementation.

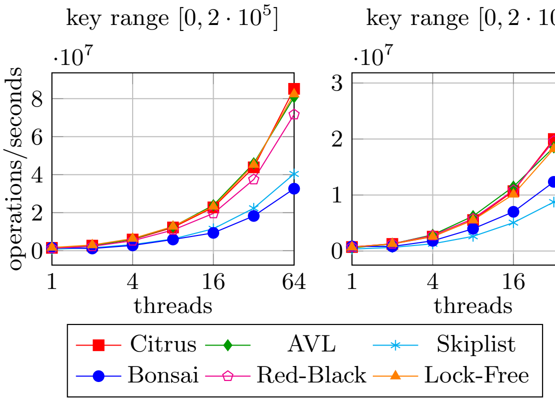


Figure 10: Throughput of the different algorithms with a single writer.

Single writer. The throughput results of the single-writer workload appear in Figure 10. This set of experiments was designed to favor the RCU-based trees, red-black tree and Bonsai: they all include a single thread executing updates (50% insert and 50% delete). Though this set favors the RCU-based trees, Bonsai does not perform well, possibly due to its functional programming style, which reconstruct parts of the tree after every update.

Other results. Figure 11 shows the results for key ranges $[0, 2 \cdot 10^5]$ and $[0, 2 \cdot 10^6]$. Experiments with 100% contains distribution (on the left) are supposed to favor the RCU-

[1] We were unable to run the red-black tree variant [20] using transactional memory to optimistically handle conflicting updates.

[2] Implemented in C by Philip W. Howard, `https://github.com/philip-w-howard/RP-Red-Black-Tree`.

[3] Implemented in C by Vincent Gramoli, `https://github.com/gramoli/synchrobench`.

based trees. As expected, the RCU-based trees show good performance, which is more visible in large key ranges.

The shortcomings of RCU-based trees with coarse-grained locks are seen already with 98% contains distribution. Both red-black and Bonsai do not scale even with a low update contention, while Citrus has similar performance to other trees. In heavy update workload of 50% contains distribution, Citrus continues to scale, though the cost of synchronize_rcu is evident. Note that unlike the AVL tree, Citrus and the lock-free tree do not pay a cost for tree balancing, which is not cost-effective when considering a uniform distribution of keys.

6. RELATED WORK

Read copy update (RCU) was introduced [24] as a solution to lock contention and synchronization overhead in read-intensive workloads. RCU can be used for explicit memory reclamation [23]. A formal semantics for RCU appears in [14, 15].

Relativistic programing is a methodology for concurrent programming using RCU, which does not assume sequentially consistent memory. It instructs readers to access items in the data structure in an order that is reverse to the order that updates modify them, but it does not deal with concurrent updates. Relativistic programming was employed in a concurrent *hash table* [28, 29] in which a lock protects each bucket. Relativistic programming was also used in a *red-black tree* [20], which allows only one concurrent writer to the tree, optimistically enforced by transactional memory [18]. RCU was used in a different way in Bonsai, a balanced tree algorithm [7]. Inspired by functional programming, Bonsai never modifies the tree in place, creating instead a new instance for the changed data structure.

Many concurrent search trees were presented in recent years, several of them using fine-grained locks, e.g., [1, 5, 8, 10]. The AVL tree of Bronson et al. [5] uses fine-grained locks, and it is partially external and relaxed balanced. Other trees are *nonblocking*, e.g., [4, 6, 11, 13, 21, 25]. These algorithms typically use *compare&swap* primitives, and in some cases, even stronger primitives on several shared variables, like *multi-word* compare&swap [13] or the customized LLX, SCX and VLX primitives [6]. Both [11, 21] use a similar technique, creating record objects for coordination between updates and inserting them into the updated node using a compare&swap operation. A different approach was used in [25], it marks edges instead of nodes, enabling insertion without helping and helping deletions without additional record object.

Several of these implementations [6, 4, 8, 10, 11, 25] also provides wait-free contains.

7. DISCUSSION

We have shown that it is possible, and even relatively simple, to design RCU-based concurrent search trees with concurrent updates. This opens up many interesting directions for future research. The obvious question is to extend Citrus to a *balanced* search tree. It is also important to integrate into Citrus two primary aspects of RCU usage, namely, efficient memory reclamation and out-of-order execution of memory instructions. A broader topic is to employ RCU in lock-free algorithms, using primitives such as *compare-and-swap* instead of locks. This will necessitate more refined mechanisms for synchronization among readers and updates, since synchronize_rcu is inherently blocking.

Acknowledgements. We would like to thank Hongseok Yang for suggesting that we look at concurrent algorithms using RCU, Noam Rinetzky for discussions of the correctness proof, Adam Morrison for his help with understanding the performance of the implementation and many insightful discussions, and finally, Yehonatan Rubin, Dana Drachsler, Alexander Libov and Eran Yahav for their help with the evaluation.

Code. Citrus source code is available at `https://bitbucket.org/mayaarl/citrus`

8. REFERENCES

[1] Yehuda Afek, Haim Kaplan, Boris Korenfeld, Adam Morrison, and Robert E. Tarjan. CBTree: A practical concurrent self-adjusting search tree. In *26th International Conference on Distributed Computing (DISC)*, pages 1–15, 2012.

[2] Hagit Attiya, Rachid Guerraoui, and Eric Ruppert. Partial snapshot objects. In *20th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 336–343, 2008.

[3] Rudolf Bayer and Mario Schkolnick. Concurrency of operations on b-trees. In Michael Stonebraker, editor, *Readings in database systems*, pages 129–139. Morgan Kaufmann Publishers Inc., 1988.

[4] Anastasia Braginsky and Erez Petrank. A lock-free B+tree. In *24th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 58–67, 2012.

[5] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In *15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 257–268, 2010.

[6] Trevor Brown, Faith Ellen, and Eric Ruppert. A general technique for non-blocking trees. In *19th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 329–342, 2014.

[7] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. Scalable address spaces using RCU balanced trees. In *7th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 199–210, 2012.

[8] Tyler Crain, Vincent Gramoli, and Michel Raynal. A contention-friendly binary search tree. In *19th International Conference on Parallel Processing (Euro-Par)*, pages 229–240, 2013.

[9] Mathieu Desnoyers, Paul E. McKenney, Alan S. Stern, Michel R. Dagenais, and Jonathan Walpole. User-level implementations of Read-Copy Update. *IEEE Transactions on Parallel and Distributed Systems*, 23(2):375–382, 2012.

[10] Dana Drachsler, Martin Vechev, and Eran Yahav. Practical concurrent binary search trees via logical ordering. In *19th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 343–356, 2014.

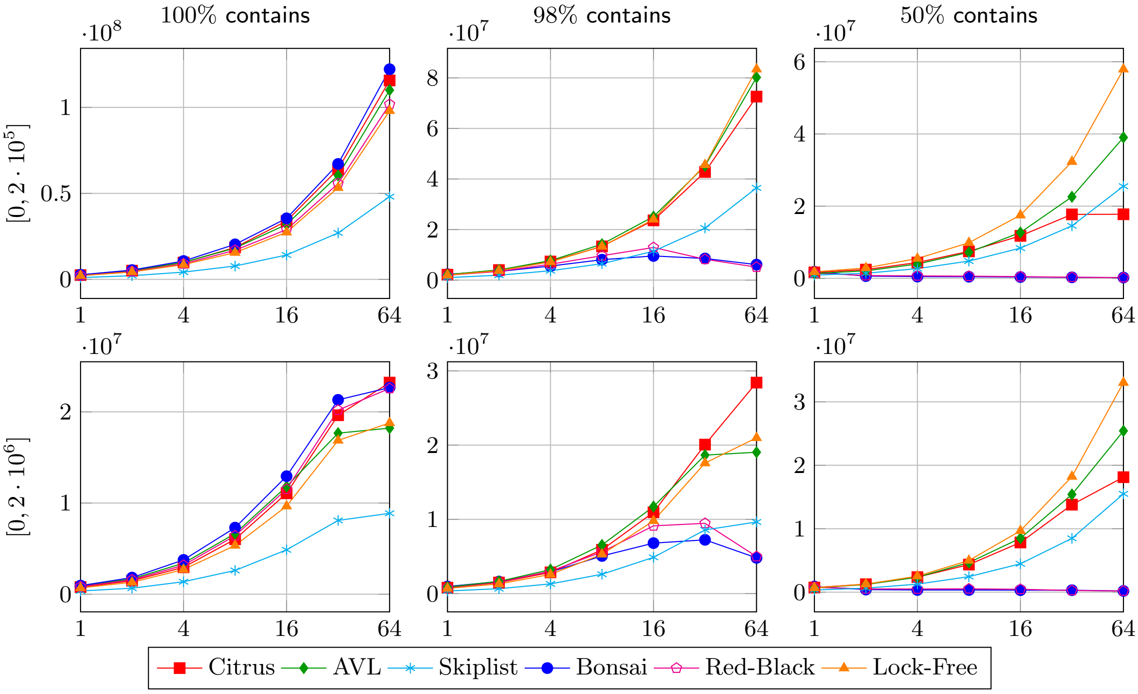


Figure 11: Throughput of the different algorithms with key range $[0, 2{\cdot}10^5]$ and $[0, 2{\cdot}10^6]$ under different operation distribution; y-axis show the throughput (operations/sec), and x-axis show the number of threads.

[11] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *29th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 131–140, 2010.

[12] Kapali P. Eswaran, Jim N. Gray, Raymond A. Lorie, and Irving L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, November 1976.

[13] Keir Fraser. *Practical lock-freedom*. PhD thesis, University of Cambridge, 2003.

[14] Alexey Gotsman, Noam Rinetzky, and Hongseok Yang. Verifying concurrent memory reclamation algorithms with grace. In *22nd European Symposium on Programming (ESOP)*, pages 249–269, 2013.

[15] Dinakar Guniguntala, Paul E. McKenney, Josh Triplett, and Jonathan Walpole. The read-copy-update mechanism for supporting real-time applications on shared-memory multiprocessor systems with Linux. *IBM Systems Journal*, 47(2):221–236, May 2008.

[16] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer III, and Nir Shavit. A lazy concurrent list-based set algorithm. In *9th International Conference on Principles of Distributed Systems (OPODIS)*, pages 3–16, 2006.

[17] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A simple optimistic skiplist algorithm. In *14th International Conference on Structural Information and Communication Complexity (SIROCCO)*, pages 124–138, 2007.

[18] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, May 1993.

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

[20] Philip W. Howard and Jonathan Walpole. Relativistic red-black trees. *Concurrency and Computation: Practice and Experience*, 2013.

[21] Shane V. Howley and Jeremy Jones. A non-blocking internal binary search tree. In *Proceedinbgs of the 24th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 161–171, 2012.

[22] Paul E. McKenney. RCU Linux usage. `http://www.rdrop.com/users/paulmck/RCU/linuxusage.html`.

[23] Paul E. McKenney. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System kernels*. PhD thesis, Oregon State University, 2004.

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

[25] Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. In *19th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 317–328, 2014.

[26] Erez Petrank and Shahar Timnat. Lock-free data-structure iterators. In *27th International Conference on Distributed Computing (DISC)*, pages 224–238, 2013.

[27] Abraham Silberschatz and Zvi Kedem. Consistency in hierarchical database systems. *J. ACM*, 27(1):72–80, 1980.

[28] Josh Triplett, Paul E. McKenney, and Jonathan Walpole. Scalable concurrent hash tables via relativistic programming. *SIGOPS Oper. Syst. Rev.*, 44(3):102–109, August 2010.

[29] Josh Triplett, Paul E. McKenney, and Jonathan Walpole. Resizable, scalable, concurrent hash tables. In *2011 USENIX conference on USENIX annual technical conference*, 2011.