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) [22], 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 [20], mostly to facilitate memory reclamation [12, 21]. However, its potential for concurrent programming remained unexploited. Although several RCU-based data structures have been proposed, for example, search trees and hash tables [6, 18, 25, 26], they all do not allow concurrent updates, either pessimistically, using a coarse-grained lock [6], or optimistically, using transactional memory [18]. At best, the data structure is partitioned into segments, e.g., buckets in a hash table [25, 26], 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 or hand-over-hand locking. 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 [24]. Figure 1 shows an example in which two readers, r1 and r2, attempt to collect the leaves, by in-order traversal, while two updates delete leaves 9 and 12. In (a), r2 already traversed the left sub-tree while r1 has not yet started its traversal. After 9 is deleted (b), r1 finishes traversing the tree while r2 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 r1 and r2 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.

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 in the tree. This scenario requires coordination with concurrent searches, before removing the successor from its previous location.

Experimental evaluation of Citrus shows that its performance beats previous RCU-based search trees [6, 18], even under mild update contention. It also compares well with the best-available concurrent dictionaries, e.g., [4, 15, 23].

The evaluation also reveals that the user-level RCU implementation [8] 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.

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.

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 \texttt{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 \texttt{rcu\_read\_lock} and \texttt{rcu\_read\_unlock}.

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

An \texttt{insert} invokes \texttt{get}, and returns \texttt{false} if \texttt{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 \texttt{get}'s search.

A \texttt{delete} invokes \texttt{get}, and returns \texttt{false} if \texttt{get} does not find the key. If the key is found in node \( v \), then there are two cases. If \( v \) 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 3(a) and (b)); later in the proof this is called \textit{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 \texttt{get} searching for the successor, which may return a false negative response (Figure 4). To overcome this problem, \texttt{delete} first inserts a copy of the successor node in the deleted key’s location. Then the \texttt{delete} waits for concurrent searches by executing \texttt{synchronize\_rcu}, and only then, removes the old successor node. Searches that start before \texttt{synchronize\_rcu} starts, find the successor in its previous location, where it remains until they complete (Figure 3(c)-(e)). Searches that start after \texttt{synchronize\_rcu} starts, find the new copy of the successor.

Note that during a \texttt{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 \texttt{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 \texttt{get}. To avoid RCU deadlocks, locks are acquired outside the read-side critical section. This creates a region of uncertainty, for example, if an \texttt{insert} reaches the node with key \( k \), but an overlapping \texttt{delete} operation removes this node from the tree before \texttt{insert} locks it (see Figure 5). If \texttt{insert} continues its operation, the new key will not be part of the tree.

We overcome this problem by \texttt{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 \texttt{insert}, the child pointer of the parent should be \( \bot \) (null pointer). A \texttt{tag} is added to each child field, in order to avoid an ABA problem (a child pointer changed to non-\( \bot \) when a leaf is inserted, then back to \( \bot \), when the leaf is moved by a \texttt{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 \( \bot \). Validating that a node was not removed is also done locally, using a \texttt{marked} field indicating that the node was deleted (in a manner similar to [14]). If validation fails, the operation releases all locks and restarts from the root.

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

![Figure 3](image1.png)  
(a) and (b) show \texttt{delete}(k) when a node has one child. (c)-(e) show \texttt{delete}(k) that replaces a node with two children with its successor; \texttt{synchronize\_rcu} is invoked between (d) and (e).

![Figure 4](image2.png)  
Figure 4: Search for key \( k' \) returns a false negative response due to an overlapping \texttt{delete}.

![Figure 5](image3.png)  
Figure 5: \texttt{insert} can add a node to an incorrect location due to an overlapping \texttt{delete}.
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).

```
function contains(key)
  (-.-,curr,-) ← get(key)
  if curr = ⊥ then ▷ The key was not found
  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).

```
function insert(key,value)
  loop
    (prev,tag,curr,direction) ← get(key)
    if curr = ⊥ then ▷ The key was found
    return false
  lock(prev)
  if validate(prev,tag,...,direction) then
    node ← new(key,value,...,⊥) ▷ Create a new leaf node
    prev.child[direction] ← node
  unlock(prev)
  return true ▷ Validation failed, release locks and retry

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

```
function validate(prev,tag,curr,direction)
  if prev.marked or prev.child[direction] ≠ curr then
    return false
  if curr ≠ ⊥ then ▷ If curr ≠ ⊥ validate curr’s marked bit
    return curr.marked
  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 ⊥, it increments the tag associated with this direction.

```
function incrementTag(node,direction)
  if node.child[direction] = ⊥ then
    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).

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 operation 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 `root` (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, 77 or 80), `incrementTag` is called in case the tag should be updated.

4. **LINEARIZABILITY OF CITRUS**

Fix an execution $\pi$ of the CITRUS algorithm. Roughly speaking, an implementation is linearizable [17] 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 $v$, $v \leftarrow u$ when $u$ is the left child of $v$, and $v \uparrow u$ when $u$ is the right child of $v$; generally, $v \leftarrow u, d = left$ 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 $v \leftarrow u$ and $k \in Set_C(u)$ then $k \leq Key(v)$ and if $v \uparrow u$ and $k \in Set_C(u)$ then $k \geq Key(v)$. The WBST property holds in an execution prefix $\sigma$ if it holds in every configuration $C$ in $\sigma$. 
CITRUS delete function

function delete(key)
loop
(prev, curr, direction) ← get(key)
if curr = nil then return false
lock(curr)
if validate(prev, curr, direction) then
if curr.child[<left>] = nil or curr.child[<right>] = nil then
curr.marked ← true
notNoneChild ← curr.child[<left>] ≠ nil ? left : right
prev.child[<direction>] ← curr.child[notNoneChild]
incrementTag(prev, direction)
release all locks
return true
else
prevSucc ← curr
succ ← curr.child[<right>]
nexth ← succ.child[<left>]
while next ≠ nil do
prevSucc ← succ
succ ← next
next ← next.child[<left>]
succDirection ← (curr = prevSucc ? right : left)
if curr = prevSucc then
Do not lock twice
lock(prevSucc)
lock(succ)
if validate(prevSucc, succ, succDirection) and
validate(succ, succ.tag[<left>], nil, nil) then
node ← new(succ.key, succ.value, curr.child[<left>],
curr.child[<right>])
lock(node)
curr.marked ← true
syncronize_rcu
wait for readers
succ.marked ← true
remove the old successor
if prevSucc = curr then
succ is the right child of curr
node.child[<right>] ← succ.child[<right>]
incrementTag(node, right)
else
prevSucc.child[<left>] ← succ.child[<left>]
incrementTag(prevSucc, left)
release all locks
return true
else
release all locks
Validation failed, release locks and retry

If the WBST property holds in an execution prefix σ that ends in
configuration C, then the range of keys in the sub-tree of node v,
denoted Range(v), can be defined by induction on nC(root, v):
RangeC(root) = (∞, ∞); if the parent of v is node u with
k = Key(u) and RangeC(u) = [minu, maxu] or RangeC(u) =
(minu, maxu), then RangeC(v) = (minu, k) or RangeC(v) =
(minu, k) if u = v, and RangeC(v) = [k, maxv] if u ≠ v.

Let u, v, w be nodes such that u → v → w, d ∈ {left, right},
and node v has only one child, in configuration C ∈ π. A primitive
write operation s that immediately follows C is a bypass of node v
if u → w in the configuration that follows s in π (Figure 3(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 v har the variable 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 asserts that if a node is not reachable
in a configuration, then its marked bit is true. Note that a node
is not reachable 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 ∈ π
and v is not reachable in configuration C’ ∈ π 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 not reachable, as required.

Case 2: Delete. By the validation in Line 49, prev → 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 (depen-
don 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 not reachable, 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
(Line 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 configura-
tion C ∈ π’ 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 ≤ ℓ. Let v be the ℓth node in the path taken by op and
let u be the (ℓ − 1)th node in this path. Suppose, without loss of
generality, that op read u → v in step s’.

By the induction hypothesis, u is reachable in some configura-
tion during π’; let C be the last such configuration. If s’ precedes
C, then v is reachable in the configuration C’ that immediately fol-
low s’. Since the reader read in step s’ precedes any access to v,
C’ precedes s (Figure 6(a)).
Figure 6: The proof of Lemma 2. Case (a) \( u \) is reachable when \( u \rightarrow v \) is read. Case (b) \( u \) is unreachable when \( u \rightarrow v \) 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 \( \text{op} \) read \( u \), 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 \( w \)'s children fields (by the code, no operation writes to child fields of \( C \)). 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 \( \pi \) 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 \( \bot \), by another operation, after the tag is read and before the node is locked, then the tag is incremented during this interval. Due to space constraints, the proof appears in the full version.

**Lemma 3.** Let \( \pi' \) be the interval starting with the step of Line 13 by \( \text{op} \) and ending with the configuration that immediately follows the lock acquisition by \( \text{op} \). If \( \text{prev}_{\text{op}}.\text{tag}[\text{direction}_{\text{op}}] \neq \text{tag}_{\text{op}} \) in every configuration in the interval of \( \pi' \) that follows \( \pi \).

Searches, implemented in \( \text{get} \), 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 \( \pi \). A critical step in the proof is to show that an \( \text{insert} \) adds a node in the right location. This relies on the following lemma (Lemma 4),

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

**Proof.** If the path traversed by \( \text{op} \) is the same as \( \text{pc}(\text{root}, \text{prev}) \), then, by the WBST property, \( k \in \text{Range}_{C}(\text{prev}) \). The path traversed by \( \text{op} \) is different from \( \text{pc}(\text{root}, \text{prev}) \), if it was changed by an overlapping update \( \text{op}' \). Consider every possible write to a child field, during \( \pi' \), the interval of \( \text{op} \).

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

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

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

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

If \( k < k' \) or \( k \geq k' \), then \( k \in \text{Range}_{C}(\text{prev}_{\text{op}}) \) still holds.

Suppose otherwise that \( k' \leq k' \) and \( k \notin \text{Range}_{C}(\text{prev}_{\text{op}}) \). If the step of line 73 by \( \text{op}' \) precedes or is in the read-side critical section of \( \text{op} \) then before removing the successor \( \text{succ}_{\text{op}} \) from the tree, \( \text{op}' \) invokes \( \text{synchronize}_{\text{RCU}} \) and waits until the end of \( \text{op}' \)’s read side critical section, by the RCU property. Since the WBST property holds during the get of \( \text{op} \), its search ends in \( \text{succ}_{\text{op}} \) (\( \text{succ}_{\text{op}} \) equals \( \text{prev}_{\text{op}} \) and \( \text{curr}_{\text{op}} \) equals \( \bot \)). Since \( \text{op} \) is an insert, it invokes \( \text{RCU}_{\text{read} \_ \text{unlock}} \) and locks \( \text{prev}_{\text{op}} \) prior to validation. On the other hand, \( \text{op}' \) unlocks \( \text{prev}_{\text{op}} \) only after marking it in Line 75. Therefore, the validation of \( \text{prev}_{\text{op}} \) by \( \text{op} \) finds that it is marked, and fails (Figure 7).

Otherwise, the step of Line 73 by \( \text{op}' \) is executed after the read side critical section of \( \text{op} \) and before \( \text{op} \) locks \( \text{prev}_{\text{op}} \). If \( \text{prev}_{\text{op}} \) equals \( \text{succ}_{\text{op}} \) then \( \text{op}' \) marks \( \text{prev}_{\text{op}} \) before releasing its lock, and the validation of \( \text{op} \) fails. If not then by the WBST property the change was in the left sub-tree of \( \text{prev}_{\text{op}} \). In this case, validation fails, because either \( \text{prev}_{\text{op}}.\text{child}[\text{left}] \neq \bot \) or \( \text{prev}_{\text{op}}.\text{tag}[\text{left}] \neq \text{tag}_{\text{op}} \) (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. Due to space constraints, the proof appears in the full version.

**Lemma 5.** The WBST property holds after every prefix \( \sigma \) of \( \pi \).

The next lemma shows that each node has a single parent, and thus, \( \text{delete} \) 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. Due to space constraints, the proof appears in the full version.

**Lemma 6.** A node that is reachable in a configuration \( C \in \pi \) has one reachable parent in \( C \).

The following lemma is used to prove that if a node \( v \) with key \( k \) remains in the tree during a \( \text{get} \), 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 \( \pi' \) and \( \text{pc}(u, v) \) \( \leq |\text{pc}(u, v)| \).

**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 properties holds.
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 → 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 πC′(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 \not\in 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 πC′(u, v), and C′ is the configuration immediately following s, then node ∈ πC′(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 s1, s2, ... be the primitive writes in π, in which successful updates are linearized, in their order in π. For every i ≥ 1, let Ci be the configuration immediately preceding si and let C′ i be the configuration immediately following si. A successful insert linearized at si is consistent if k ∈ SetC′ i(root) and k ∈ SetC′ i(root), and a successful delete linearized at si is consistent if k ∈ SetC′ i(root) and k ∉ SetC′ 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 ∈ SetC(root) and assume that k ∈ SetC′(root), for every configuration C′ ∈ π′ 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 → 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 prev in C or the root if C precedes Line 3 of get. Since the WBST property holds in π and insert adds leaves, v′ ∈ πC(root, v)

If C does not precedes the last access to a child field by get, then prev → 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 ∈ π′ be the primitive write by operation op′ that makes v unreachable. Let C′ be the configuration immediately preceding s in π′ and let C′ be the configuration immediately following s in π′; v is reachable in C′ and unreachable in C′. By the validation in Line 27, insert does not make a node unreachable, and hence, only delete can make v unreachable. Since k ∈ SetC′(root), op′ has key k′ \neq k. The only other way a delete makes a node unreachable is when v = succ and s is the primitive write of either Line 77 or Line 80.

Let s′ ∈ π′ be the primitive write by operation op′ that adds node spoken to the tree. Let C′ be the configuration immediately preceding s′ in π′ and let C′ be the configuration immediately following s′ in π′. Since succ is found by traversing the leftmost branch of the sub-tree rooted at curr, there is a path from curr to v in a configuration that precedes C′, there is a path from curr to v in C′ (Lemma 7). By Line 70, node spoken has the same children as curr, and therefore, there is a path from node spoken to v in C′. This means that node spoken ∈ πC′(root, v), and by the choice of v, node spoken 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. Due to space constraints, the proof appears in the full version.

**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′).

**Lemma 10.** All successful updates are consistent.

**Proof.** By induction on the linearization points, s1, s2, ... For every j ≥ 1, let Cj be the configuration immediately preceding s j and Cj 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 s1. Let k be the key of op, and...
note that \( k \notin \text{Set}_C(\text{root}) \) for every configuration \( C \) preceding \( s_1 \), and by Lemma 2, \( \text{op} \) cannot access a node with key \( k \), hence, \( \text{op} \) is a successful insert. Since \( \text{op} \) accesses \( \text{prev} \), by Lemma 2, \( \text{prev} \) is reachable in some configuration preceding the step of Line 27. Since \( \text{prev} \) is validated in Line 27 and is unmarked, Lemma 1 implies that \( \text{prev} \) is reachable in \( C_i \) and \( k \in \text{Set}_{C_i}(\text{root}) \). For the induction step, assume that the successful updates linearized at \( s_1, s_2, \ldots, s_{i-1} \) are consistent, and consider the successful update \( \text{op} \) linearized at \( s_i \).

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

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

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

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

Otherwise, since the WBST property holds in \( \pi \), \( s_j \) inserts the new node as the child of \( \text{prev}_v \) in direction \( d \), and the validation of \( \text{op} \) fails, because either \( \text{prev}_v \cdot \text{child}[d] \neq \bot \) or \( \text{prev}_v \cdot \text{tag}[d] \neq \bot \) (Lemma 3).

An operation \( \text{op} \) with interval \( \pi' \), of the remaining types, is linearized as follows:

- If \( \text{op} \) is a failed contains or a failed delete, then \( \text{curr} = \bot \).
  - If \( k \in \text{Set}_C(\text{root}) \) for every configuration \( C \in \pi' \), then \( k \in \text{Set}_C(\text{root}) \) for every configuration \( C \in \pi'' \), where \( \pi'' \) is the read-side critical section of \( \text{op} \). By Lemma 8(i), \( \text{op} \) reaches a node with key \( k \), which is a contradiction. Therefore, \( k \notin \text{Set}_C(\text{root}) \), for some configuration \( C \in \pi' \). The linearization point of \( \text{op} \) is last after \( \pi'' \) such that \( k \notin \text{Set}_C(\text{root}) \), and before the successful insert with the same linearization point, if such insert exists. This linearization point is clearly consistent.
  - If \( \text{op} \) 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 at 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 \( \text{op} \), \( \text{curr} = v \), and by condition of the while loop in get (Line 7), \( k = \text{Key}(v) \). By Lemma 2, \( \text{curr} \) is reachable in some configuration \( C' \in \pi'' \), and the linearization point exists. This linearization point is consistent since \( k \in \text{Set}_C(\text{root}) \) and if \( \text{op} \) is a successful contains, it returns \( v \cdot \text{value} \).

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 [18] and Bon sai, a balanced search tree [6], both using a global lock to synchronize among updates.\(^3\) We also compared CITRUS to three concurrent dictionary data structures, which have C implementations: The optimistic AVL tree\(^4\) [4], the lock-free search tree [23] and the lock-based lazy skiplist\(^5\) [15].

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, \([0, 2 \cdot 10^3] \) and \([0, 2 \cdot 10^4] \); 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 show 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.

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 much more than the algorithmic aspects of the implementation.

**New RCU.** During our initial evaluation of CITRUS, we identified that the user-space RCU implementation [8] 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 8 shows representative results; similar behavior was observed under different update contention and key ranges. To show that the drop in throughput is just an implementation issue, we re-implemented the subset of the RCU API used in CITRUS, in a manner similar to epoch-based reclamation [11]. 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 \texttt{rcu_read_lock} operation increments the counter and sets the flag to \texttt{true}, while the \texttt{rcu_read_unlock} operation sets the flag to \texttt{false}. When a thread executes a synchronize\_rcu operation, it waits for every other thread, until one of two things occurs:

\(^3\)We were unable to run the red-black tree variant [18] using transactional memory to optimistically handle conflicting updates.


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 8 shows the throughput of CITRUS with the new RCU implementation. All other experiments were run with the new RCU implementation.

**Single writer.** The throughput results of the single-writer workload appear in Figure 9. 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 10 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-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 [22] as a solution to lock contention and synchronization overhead in read-intensive workloads. RCU can be used for explicit memory reclamation [21]. A formal semantics for RCU appears in [12, 13].

Relativistic programming 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 [25, 26] in which a lock protects each bucket. Relativistic programming was also used in a red-black tree [18], which allows only one concurrent writer to the tree, optimistically enforced by transactional memory [16]. RCU was used in a different way in Bonsai, a balanced tree algorithm [6]. 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, 4, 7, 9]. The AVL tree of Bronson et al. [4] uses fine-grained locks, and it is partially external and relaxed balanced. Other trees are nonblocking, e.g., [3, 5, 10, 11, 19, 23]. These algorithms typically use compare&swap primitives, and in some cases, even stronger primitives on several shared variables, like multi-word compare&swap [11] or the customized LLX, SCX and VLX primitives [5]. Several of these implementations [5, 3, 7, 9, 10, 23] 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.

### 8. REFERENCES


Figure 10: 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.