# Concurrent Wait-Free Red Black Trees* †

Aravind Natarajan, Lee H. Savoie, and Neeraj Mittal

*Erik Jonsson School of Engineering and Computer Science*
*The University of Texas at Dallas*
*Richardson, TX 75080, USA*

**Abstract**

We present a new *wait-free* algorithm for concurrently accessing a red-black tree in an asynchronous shared memory system that supports search, insert, update and delete operations using single-word compare-and-swap instructions. Search operations in our algorithm are fast and execute only read and write instructions (and no atomic instructions) on the shared memory. The algorithm is obtained through a progressive sequence of modifications to an existing general framework for deriving a concurrent wait-free tree-based data structure from its sequential counterpart. Our experiments indicate that our algorithm significantly outperforms other concurrent algorithms for a red-black tree.

**Key words:** concurrent data structure; non-blocking implementation; wait-free algorithm; red-black tree; compare-and-swap instruction

## 1 Introduction

With the growing prevalence of multi-core, multi-processor systems, concurrent data structures are becoming increasingly important. In such a data structure, multiple processes may need to operate on overlapping regions of the data structure at the same time. Contention between different processes must be managed in such a way that all operations complete correctly and leave the data structure in a valid state.

Concurrency is most often managed through locks. However, locks are blocking; while a process is holding a lock, no other process can access the portion of the data structure protected by the lock. If a process stalls while it is holding a lock, then the lock may not be released for a long time. This may cause other processes to wait on the stalled process for extended periods of time. As a result, lock-based implementations of concurrent data structures are vulnerable to problems such as deadlock, priority inversion and convoying [2, 22].

Non-blocking algorithms avoid the pitfalls of locks by using hardware-supported *read-modify-write* instructions such as *load-link/store-conditional (LL/SC)* [19, 20] and *compare-and-swap (CAS)* [22]. A load-link instruction returns the current value of a memory location; a subsequent store-conditional instruction to the same memory location will store a new value only if no updates have occurred to that location since the load-link instruction was performed. A compare-and-swap

*This work was supported, in part, by the National Science Foundation (NSF) under grant number CNS-1115733.
†This work has appeared as a brief announcement in the 26th International Symposium for Distributed Computing (DISC), 2012.

instruction compares the contents of a memory location to a given value and, only if they are the same, modifies the contents of that memory location to a given new value.

Non-blocking implementations of common data structures such as queues, stacks, linked lists, hash tables, and search trees have been proposed [1, 4, 9, 11, 13, 15, 18, 20–24, 26, 30, 31, 33, 34, 38, 41]. Non-blocking algorithms may provide varying degrees of progress guarantees. Three widely accepted progress guarantees are: obstruction-freedom, lock-freedom, and wait-freedom [22]. An algorithm is *obstruction-free* if any process that executes in isolation will finish its operation in a finite number of steps. An algorithm is *lock-free* if some process will complete its operation in a finite number of steps. An algorithm is *wait-free* if every operation executed by every process will complete in a finite number of steps.

Binary search tree is one of the fundamental data structures for organizing ordered data that supports search, insert, update and delete operations [7]. Red-black tree is a type of self-balancing binary search tree that provides good worst-case time complexity for search, insert, update and delete operations. As a result, they are used in symbol table implementations within systems like C++, Java, Python and BSD Unix [36]. They are also used to implement completely fair schedulers in Linux kernel [25]. However, red-black trees have been remarkably resistant to parallelization using both lock-based and lock-free techniques. The tree structure causes the root and high level nodes to become the subject of high contention and thus become a bottleneck. This problem is only exacerbated by the introduction of balance requirements.

**Related Work:** Designing an efficient concurrent non-blocking data structure that guarantees wait-freedom is hard. Several universal constructions exist that can be used to derive a concurrent wait-free data structure from its sequential version [6, 12, 19, 20]. Due to the general nature of the constructions, when applied to a binary search tree, the resultant data structures are quite inefficient. They involve either: (a) applying operations to the data structure in a serial manner, or (b) copying the entire data structure (or parts of it that will change and any parts that directly or indirectly point to them), applying the operation to the copy and then updating the relevant part of the shared data structure to point to the copy. The first approach precludes any concurrency. The second approach, when applied to a tree, also precludes any concurrency since the root node of the tree indirectly points to every node in the tree.

Several customized non-blocking implementations of concurrent unbalanced search trees [5, 11, 24, 35], and balanced search trees such as B-tree [1] and $B^+$-tree [3] have been proposed, that are typically more efficient than those obtained using universal constructions.

In [28], Ma presented a "lock-free" algorithm for a concurrent red-black tree that only supports search and insert operations using CAS, DCAS (double-word[1] CAS) and TCAS (triple-word[1] CAS) instructions [28]. Kim *et al.* extended Ma's algorithm to support delete operations as well as eliminate the use of multi-word CAS (DCAS and TCAS) instructions [27]. However, a closer inspection of the algorithm reveals that it is actually a blocking algorithm and not lock-free. It is only lock-free in the sense that CAS instructions are used for synchronization (acquiring and releasing flags on nodes) and no "locks" are acquired. However, a process that is blocked while it is holding the flag on the root of the tree will prevent *all other processes* from making progress. Concurrent algorithms for a red-black tree based on the transactional memory framework have also been proposed in [8, 14]. The algorithm in [8] maintains a relaxed red-black tree (may violate balance requirements); the update and re-balancing operations of the tree are decoupled, and a dedicated rotator thread regularly scans the tree to correct any balance violations. In contrast to the above algorithms, our algorithm has the following desirable properties: (a) it uses only single word

[1]Words need not be adjacent.

CAS instruction, which is commonly available in most hardware architectures including Intel 64 and AMD64 architectures. (b) it does not require any additional underlying system support such as transactional memory, and (c) it never allows the tree to go out of balance.

For a tree-based data structure that supports operations executing in top-down manner using small-sized windows, Tsay and Li's framework [40] can be used to derive a concurrent wait-free data structure from its sequential version. Operations are injected into the tree at the root node, and work their way toward a leaf node by operating on small portions of the tree at a time. Wait-freedom is achieved using helping; as an operation traverses the tree, i8t helps any operation that it encounters on its way "move out" of its way. The framework requires that an operation (including a search operation) makes a copy of every node that it encounters as it traverses the tree. When compared with universal constructions, the wait-free data structure obtained using Tsay and Li's framework yields more efficient modify operations (since modify operations working on different parts of the tree can execute concurrently once their paths diverge) but less efficient search operations (since search operations also have to copy nodes). Our wait-free algorithm is based on Tsay and Li's framework, but significantly modified to (a) overcome some of its practical limitations, and (b) reduce the overhead for search and modify operations.

**Contributions:** In this paper, we present a new *wait-free* algorithm for concurrently accessing a red-black tree in an asynchronous shared memory system that supports search, insert, update and delete operations using a single-word compare-and-swap instruction. Search operations in our algorithm are fast and perform only read and write instructions (and no atomic instructions) on the shared memory. The algorithm is obtained through a progressive sequence of modifications to the Tsay and Li's framework for deriving a concurrent wait-free tree-based data structure from its sequential counterpart. We also describe a wait-free garbage collection operation for reclaiming memory allocated to "inaccessible" objects that can be run concurrently with other operations. Our experiments indicate that our algorithm *significantly outperforms* all other concurrent algorithms for maintaining a (non-relaxed) red-black tree that can be implemented directly without any additional system support.

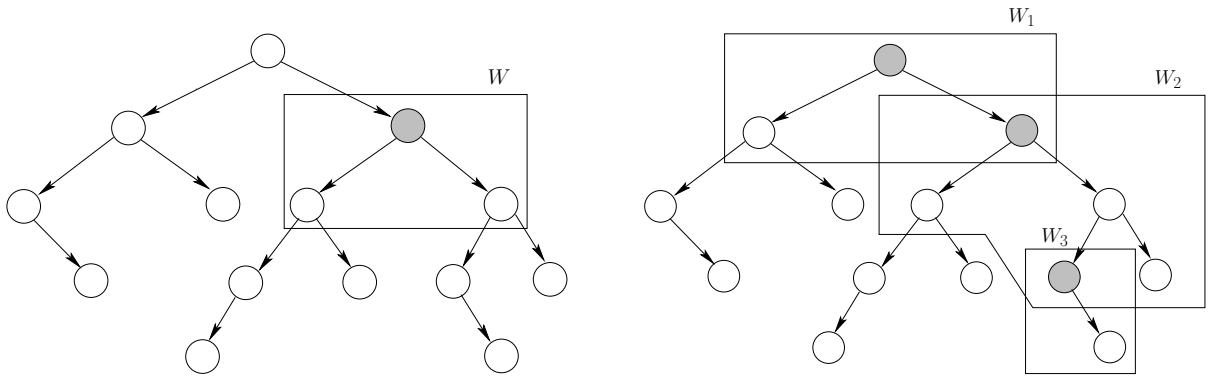
**Roadmap:** We first describe the preliminaries necessary to understand our work in Section 2. The wait-free algorithm for a concurrent red-black tree is described in Section 3 and its proof of correctness is described in Section 4. A wait-free algorithm for garbage collection is presented in Section 5. We describe our experimental results in Section 6. Finally, Section 7 concludes the paper.

# 2 Preliminaries

We first describe the Tsay and Li's framework for deriving wait-free algorithms for concurrent tree-based data structures. We then briefly discuss red-black trees and their key properties we use in our algorithm.

## 2.1 Tsay and Li's Wait-Free Framework for Concurrent Tree-Based Data Structures

Tsay and Li described a framework in [40] that can be used to develop wait-free operations for a tree-based data structure provided operations traverse (and modify) the tree in top-down manner. The framework is based on the concept of a *window*, which is simply a *rooted subtree* of the tree




**(a)** A window of a top-down operation. **(b)** Overlapping windows of a top-down operation.

**Figure 1:** Windows in Tsay and Li's framework.

structure, that is, a small, contiguous piece of the tree. We say that a window is *located* at its root node. For example, Figure 1(a) shows a window $W$ of the tree; the shaded node denotes the root node of $W$.

The execution of a top-down operation can be modeled using a sequence of windows. For example, Figure 1(b) shows a sequence of three windows $W_1$, $W_2$ and $W_3$ starting from the root of the tree and ending at a leaf of the tree; the shaded nodes denote the root node of the windows. Note that different windows of an operation may be of different shapes and sizes. We refer to actions performed by an operation as part of its window as *transaction*.

In the Tsay and Li's framework, when an operation starts, it first needs to be "injected" into the tree. This involves obtaining the owenership of (or "locking") the root of the tree. This step basically "initializes" the first window of the operation. Once an operation is injected into the tree, it then performs a sequence of window transactions until it reaches the bottom of the tree at which point it terminates. Consecutive windows of an operation always *overlap*. The root of the next window is part of the current window. For an illustration, see Figure 1(b). A process table is used to store the current state of the most recent operation of each process.

We now explain how a window transaction is performed in Tsay and Li's framework. To execute a window transaction of an operation $\alpha$ in Tsay and Li's framework, a process $p$ needs to perform four steps described as follows. Let $W_G$ denote the current window of $\alpha$.

1. *Explore-Help-And-Copy:* In this step, $p$ traverses nodes in $W_G$ starting from the root node of $W_G$. On visiting a node $X$ in $W_G$, if $p$ finds that $X$ is owned by another operation $\beta$ (implying that $\beta$ is "blocking" $\alpha$'s way), then $p$ helps $\beta$ "move out" of $\alpha$'s way by performing a window transaction on $\beta$'s behalf. Process $p$ also makes a copy of $W_G$, denoted by say $W_L$. Note that, at this point, only $p$ can access nodes in $W_L$.

2. *Transform-And-Lock:* In this step, $p$ modifies $W_L$ as needed (*e.g.*, flipping colors or performing rotations). Let $W_L^M$ denote the window obtained after aplying all transformations to $W_L$. Let $Y$ denote the node in $W_L^M$ that corresponds to the root node of the next window of $\alpha$ (recall that consecutive windows of an operation overlap). Process $p$ then obtains the ownership of node $Y$. Note that actions in this step do not require any synchronization because, at this point, only $p$ can access nodes in $W_L^M$.

3. *Install:* In this step, $p$ replaces the window $W_G$ in the tree with the window $W_L^M$ in its local memory using a synchronization instruction. If this step succeeds, then nodes in $W_L^M$ become accessible from the root of the tree and are thus visible to all processes in the system. Further,

nodes in $W_G$ are no longer accessible from the root of the tree (but some processes may still hold references to them). We refer to nodes that are reachable from the root of the tree as *active nodes*, and nodes that were once active but not any more as *passive nodes*. Note that, on performing this step, $\alpha$'s ownership of the root node of the current window in the tree is released and that of the next window in the tree gained *atomically*.

4. *Announce:* Finally, in this step, $p$ announces the location of $\alpha$'s new window to other processes in the system by updating the (process) table entry of the process that generated $\alpha$. It is possible for this step to be performed by another process (different from $p$) since $\alpha$'s new window is now visible to all processes in the system. Sufficient information is stored in the root node of the window to enable this to happen. A synchronization instruction is used to update the process table entry because multiple processes may attempt to update the same entry with different values.

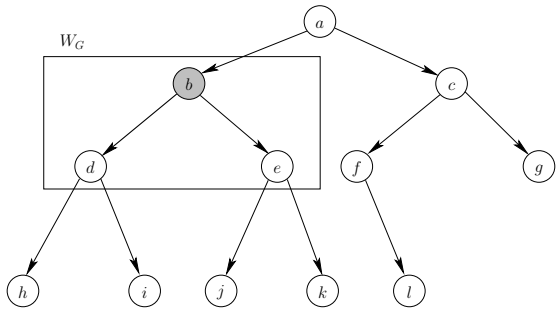
Figure 2 illustrates the first three steps of performing a window transaction. The last step (not shown in the figure) would involve updating the process table entry storing the current location of the window to point to the new location.

Consider a window rooted at some tree node, say $X$. As part of executing the window transaction, multiple attributes of $X$ (*e.g.*, color, key and/or children pointers) may need to be changed. This, in general, cannot be performed using a single synchronization instruction. To address this problem, a tree node in Tsay and Li's framework has a *dual* structure; it consists of a *pointer* node and a *data* node. The pointer node contains a reference to the data node; it also contains information about whether the tree node (it represents) is owned by some operation or is free. The data node stores all other attributes of the tree node (color, key, *etc.*). This dual structure allows a window in the tree to be replaced by replacing the data node of its root node. See Figure 3 for an example. In the figure, the tree node $X$ is represented using pointer node $A$ and data node $B$ with $A$ containing reference to $B$. The window $W_G$ is rooted at tree node $X$. Copying $W_G$ involves copying nodes $B$, $C$, $D$, $E$ and $F$ ($C$ and $E$ are pointer nodes, and $B$, $D$ and $F$ are data nodes). Let $W_L$ denotes the local window after all changes have been applied to the copy of $W_G$. Window $W_G$ can be replaced with window $W_L$ by changing the reference stored in $A$ to point to $G$ from $B$.

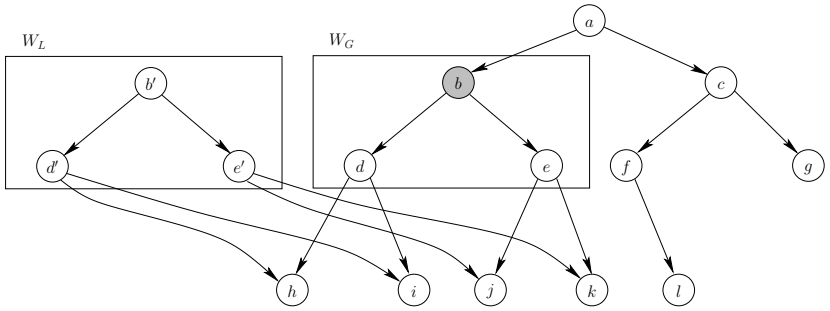
The original Tsay and Li's framework [40] has several limitations. First, the pointer node, which is a single word, needs to store two distinct addresses. Second, it assumes the availability of a special hardware instruction *check_valid* that checks if the contents of a word have changed since they were last read using an LL instruction; to our knowledge, such an instruction is not currently implemented in hardware. For our algorithm, we have modified the framework to remove both the above limitations. A pointer node in our algorithm needs to only store a single address (and a small number of bits). Further, our algorithm uses only single-word CAS instruction, which is widely available in hardware. Hereafter, we refer to Tsay and Li's framework, modified to make it more practical, as MTL-framework; our wait-free algorithm is built on top of this modified framework.

### 2.2 Red-Black Trees and Top-Down Operations

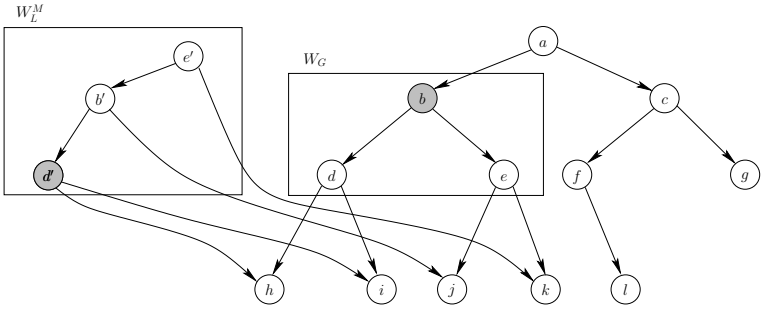
We assume that a red-black tree, which is a type of self-balancing binary search tree, implements a dictionary of *key-value pairs* and supports the following four operations. A *search* operation explores the tree for a given key and, if the key is present in the tree, returns the value associated with the key. An *insert* operation adds a given key-value pair to the tree if the key is not already present in the tree. Otherwise, it becomes an *update* operation and changes the value associated with the key to the given value. A *delete* operation removes a key from the tree if the key is present in the tree. A *modify* operation is an insert, update or delete operation.



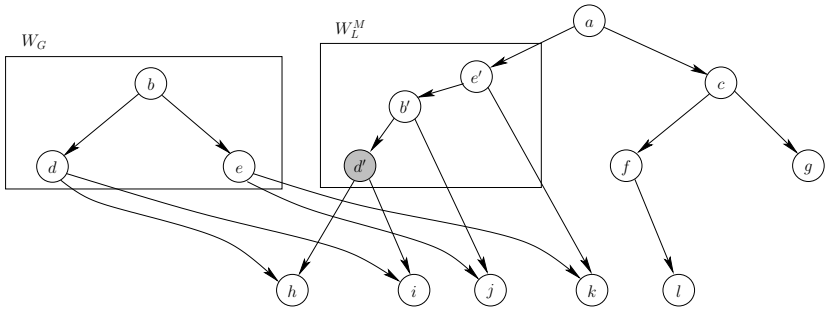

**(a)** A window on which an operation is currently working on.




**(b)** Make a local copy of the window.




**(c)** Transform the local window as needed (transformations applied depend on the specific operation).




**(d)** Replace the original window with the local window using a synchronization instruction. Nodes in the original window are no longer accessible from the root of the tree.

**Figure 2:** All illustration of the steps involved in performing a window transaction. The shaded nodes in the tree denote the root node of the windows.

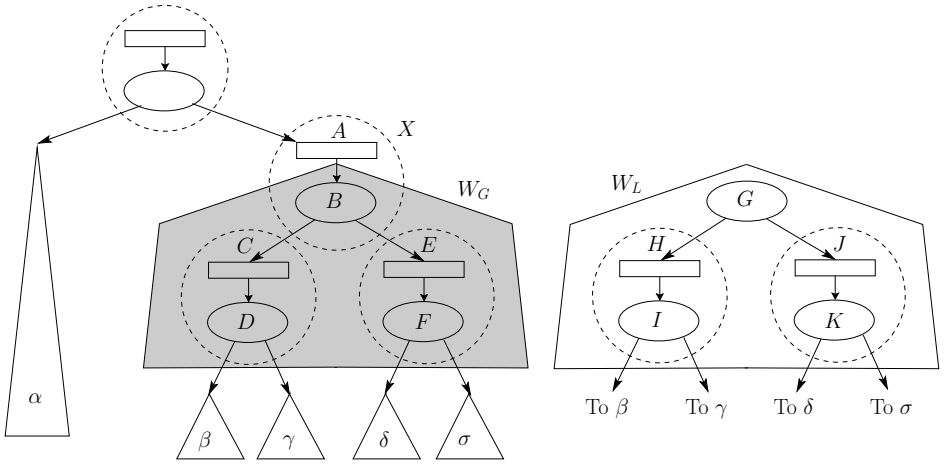



**Figure 3:** An illustration of the dual node structure of a tree node in Tsay and Li's framework.

Traditional insert and delete operations for maintaining a red-black tree do not work in a top-down manner (a top-down phase may be followed by a bottom-up phase for re-balancing the tree). Approaches for top-down insertion and deletion in a red-black tree have been proposed in [17, 39] that execute the operation in a *single pass* of the tree from the root node to a leaf node. For our algorithm, we use the top-down operations proposed by Tarjan in [39] that work on an *external* red-black tree in which all the data are stored in the leaf nodes. Basically, all operations begin at the root of the tree and traverse the tree towards the leaf nodes along a path called the *access path* using a constant-size window, while maintaining specific invariants. Broadly speaking, an insert operation maintains the invariant that the root of the current window is colored black and has a black-colored child, whereas a delete operation maintains the invariant that either the root of the current window or its child or its grandchild is colored red. For more details of insert and delete operations, please refer to [39].

# 3 A Wait-Free Algorithm for Red-Black Tree

## 3.1 Reducing the Overhead of Search Operation

Note that MTL-framework, when used with red-black tree operations that work in top-down manner [17, 39], yields a wait-free red-black tree. But the resulting concurrent data structure has a serious limitation. In the MTL-framework, every operation including search operation: (i) only "acts" on active nodes, (ii) needs to make a copy of every node that it encounters, and (iii) needs to help every stalled operation on its path before it can advance further. The copying and helping makes an operation *expensive* to execute. Additionally, every operation that is currently executing on the tree, including a search operation, owns a node in the tree, and each node can only be owned by one operation at a time. This implies that concurrently invoked search operations that follow the same path in the tree will conflict with each other, which is an unusual behavior for a concurrent algorithm.

To reduce the overhead of a search operation, we make the following observations. First, in the MTL-framework, a window transaction is *atomic* with respect to an operation; either the operation sees *all* modifications made by the transaction or *none* of the modifications made by the transaction. This is because a process makes change to its *local* window first (which is private to the process) and then installs it in the tree using a *single* CAS instruction (at which time it

becomes publicly accessible by all processes). Second, every window transaction applied to the tree maintains the *legality* of the red-black tree, that is, the set of active nodes in the tree always form a valid red-black tree. So, in our algorithm, a search operation simply traverses the tree, unaware of other operations and without helping other operations on their path complete. Clearly, a search operation can now proceed concurrently with other operations (search as well as modify) without interfering with them. Note that, as a modify operation traverses the tree from the root node to a leaf node window-by-window, it replaces all nodes in the current window with new copies before moving down. Therefore, as a search operation proceeds, it may encounter nodes that are no longer part of the tree. Nevertheless, it can be shown that the result of a search operation is still meaningful.

## 3.2 Reducing the Overhead of Modify Operation

We reduce the overhead of a modify operation in two ways. We describe them one-by-one.

### 3.2.1 Minimizing the Use of the MTL-Framework

By significantly reducing the overhead of a search operation, we can also reduce the overhead of a modify operation in practice by first using a search operation to determine whether the tree contains the key and, depending on the result, execute the modify operation using the MTL-framework if needed [22]. For example, for an insert/update operation, if the search operation finds the key, then it returns the address of the leaf node containing the key and the insert/update operation can change the value associated with the key outside the MTL-framework. Note that, in the MTL-framework, a node is replaced with a new copy whenever it happens to be in the window of a modify operation. Hence, to be able to change the value associated with a key outside the MTL-framework, the value can no longer be stored inside a node. Rather, it has to be stored *outside* a node as a separate *record* with the node containing the *address* of the record. Also, a search operation is changed to return the address of the record (containing the value) if it finds the given key in the tree.

An insert/update operation now consists of the following three phases: (a) *Phase 1:* The tree is searched for the given key using the fast search operation. (b) *Phase 2:* If the key does not exist in the tree, the expensive insert operation is executed using the MTL-framework, and the key along with its associated value are inserted into the tree. (c) *Phase 3:* If the key already exists in the tree, the record containing the value (associated with the key) is modified to update the value outside the MTL-framework. Note that an operation in phase 2 may find that the key already exists in the tree. This may happen when another operation adds the key to the tree after phase 1 of the first operation completes but before its phase 2 starts. In that case, the insert operation becomes an update operation after completing its phase 2 and then executes phase 3 as well. To accomplish, we modify the MTL-framework to return the address of the record in case the key is already present in the tree. To summarize, an insert operation consists of phase 1 followed phase 2, whereas an update operation consists of phase 1, followed by possible phase 2, followed by phase 3.

A delete operation consists of the following two phases: (a) *Phase 1:* The tree is searched for matching key using the fast search operation. If the key does not exist in the tree, no further action is required and the delete operation terminates. (b) *Phase 2:* If the key exists in the tree, the expensive delete operation is executed using the MTL-framework, and the key along with its associated value are deleted from the tree.

**Updating the Value in a Record** To modify the value associated with a key in phase 3 of an update operation, we adopt the wait-free algorithm described by Chuong *et al.* in [6]. The algorithm uses two data structures that are shared by all processes: (i) an array *announce* that is used by processes to *announce* their operations to other processes, and (ii) a variable *gate* that is used by processes to *agree* on the next operation to execute. To maximize concurrency, we use a separate instance of Chuong *et al.*'s algorithm for each record. To allow multiple instances of Chuong *et al.*'s algorithm to execute concurrently, each record can store a separate copy of *announce* and *gate* variables. This, however, causes the space complexity of our algorithm to increase by a multiplicative factor that is linear in the number of processes because *announce* array contains one entry for every process. To reduce the space-complexity, we proceed as follows. All records share the same *announce* array, but each record has its own copy of the *gate* variable. We modify Chuong *et al.*'s algorithm so that a process helps an update operation only if the operation *conflicts* with its own update operation (wants to update the value stored in the same record). This would require storing the address of the record that an update operation wants to modify in the *announce* array. Processes whose update operations conflict use the *gate* variable stored in the (target) record to decide on the next update operation to be applied to the value.

### 3.2.2 Minimizing Copying of Nodes in the MTL-Framework

There may be situations when a transaction does not need to modify the window of the tree in any way because the required invariant already holds [39]. We refer to such transactions as *trivial* transactions. Clearly, it is wasteful for a trivial transaction to copy the entire window of the tree in local memory and then replace that window with an identical copy. It is instead desirable for the window to simply *slide down* to its next root. To avoid copying a window, acquiring ownership of the next root of the window and releasing ownership of the current root of the window is no longer an atomic step as in the MTL-framework. Rather, a process first needs to acquire ownership of the next root of the window and then release the ownership of the current root of the window in two separate steps.

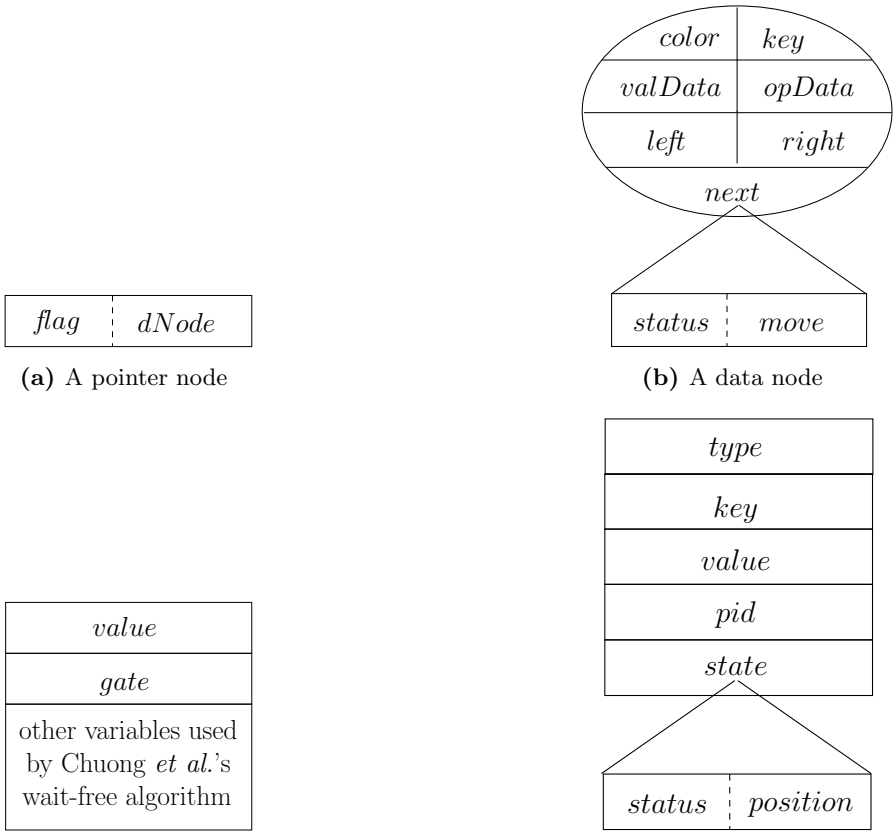
The consequence of not copying the entire window is that a modify operation can now *overtake* a search operation that started before it. In other words, a search operation may now be *able to see* the effect of a window transaction generated by a modify operation that *started later*. As a result, it is possible for a search operation to never complete if it is repeatedly overtaken by a constant stream of modify operations that continually cause the bottom of the tree to move down.

To ensure that a search operation eventually terminates, a modify operation may now have to help a search operation complete. To that end, whenever a process executes a modify operation, just before it starts executing window transactions (specifically, at the beginning of phase 2), it selects a process to help in a round-robin manner. If the search operation of the process it selected at the beginning of phase 2 is still pending at the end of phase 2, then it helps that search operation complete.

## 3.3 Data Structures Used

Our algorithm uses four major data structures: (1) *pointer node* which stores reference to the data node, (2) *data node* which stores tree node attributes, (3) *value record* which stores the value associated with the key, and (4) *operation record* which stores information about the operation such as its type, arguments and current state. They are shown in Figure 4.

A pointer node, which is a single word, contains the following fields: (a) *flag*: indicates whether the node is owned by an operation, and (b) *dNode*: the address of the data node. The *flag* field




**(a)** A pointer node **(b)** A data node

**(c)** A value record **(d)** An operation record

**Figure 4:** Data structures used by our algorithm.

has two possible values: FREE or OWNED.

A data node contains the following fields: (a) node specific attributes such as color, key, pointers to left and right children nodes, denoted by *color*, *key*, *left* and *right*, respectively, (b) *valData*: the address of the record that contains the value associated with the key, (c) *opData*: the address of an operation record (only relevant if the node was/is the root of some operation window), and (d) *next*: information about the operation after executing window transaction (only relevant if the node was the root of some operation window); it contains two sub-fields (packed into a single word): (i) *status*: the next status of the operation, and (ii) *move*: the address of the next location of the operation window. The *status* field has three possible values: WAITING (waiting to be injected into the tree), IN_PROGRESS (executing window transactions) and COMPLETED (terminated).

A value record contains the following fields: (a) *value*: the value associated with the key, and (b) variables used by the Chuong *et al.*'s wait-free algorithm (*e.g.*, *gate*).

An operation record contains the following fields: (a) operation specific attributes such as its type, arguments and process identifier, denoted by *type*, *key*, *value* and *pid*, and (b) *state*: information about the current state of the operation; it contains two sub-fields (packed into a single word): (i) *status*: the current status of the operation, and (ii) *position*: the address of the current location of the operation window. In case of search or update operation, the *position* field of its operation record is used to store the address of the record containing the value (if found).

Besides the above data structures, our algorithm also uses two tables, namely modify table, denoted by $MT$, and search table, denoted by $ST$. They are used to enable *helping* so as to ensure the wait-freedom property. Each table contains one entry for every process; the entry stores the

```
1  Value search( key )
2  begin
3      opData := create( SEARCH, key, ⊥ ) ;          // create and initialize a new operation record
4      opData→state := {IN_PROGRESS, null} ;                // initialize the operation state
5      ST[myid].opData := opData ;                        // initialize the search table entry
6      traverse( opData ) ;                                                  // traverse the tree
7      if (opData→state)⇝position ≠ null then
8          read the value stored in the record using Chuong et al.'s algorithm and return it;
9      else
10         return ⊥;

11 insertOrUpdate( key, value )
12 begin
13     valData := null;
       // phase 1:  determine if the key already exists in the tree
14     search( key );
15     valData := (ST[myid].opData→state)⇝position;
16     if valData = null then
           // phase 2:  try to add the key-value pair to the tree using the MTL-framework
           // select a search operation to help at the end of phase 2 to ensure wait-freedom
17         pid := the process selected to help in round-robin manner;
18         pidOpData := ST[pid].opData;
19         opData := create( INSERT, key, value ) ;   // create and initialize a new operation record
20         executeOperation( opData ) ;                          // add the key-value pair to the tree
21         valData := (opData→state)⇝position;
22         if pidOpData ≠ null then
23             traverse( pidOpData ) ;                 // help the selected search operation complete
24     if valData ≠ null then
           // phase 3:  update the value in the record using Chuong et al.'s algorithm

25 delete( key )
26 begin
       // phase 1:  determine if the key already exists in the tree
27     if search( key ) then
           // phase 2:  try to delete the key from the tree using the MTL-framework
           // select a search operation to help at the end of phase 2 to ensure wait-freedom
28         pid := the process selected to help in round-robin manner;
29         pidOpData := ST[pid].opData;
30         opData := create( DELETE, key, ⊥ ) ;      // create and initialize a new operation record
31         executeOperation( opData ) ;                              // remove the key from the tree
32         if pidOpData ≠ null then
33             traverse( pidOpData ) ;                 // help the selected search operation complete
```

**Figure 5:** Pseudo-code for MinimalCopy.

address of the operation record of the most recent operation generated by the process. Given a tree node $X$, let $pnode(X)$ and $dnode(X)$ denote its pointer node and data node, respectively.

### 3.4 Details of the Algorithm

A detailed pseudo-code of the algorithm is given in Figures 5-10. The pseudo-code is self-explanatory. It uses the following functions: (i) read to dereference a pointer node and extract both its fields,

```
34 traverse( opData )
35 begin
36     dCurrent := pRoot ⇝ dNode ;                          // start from the root of the tree
37     while dCurrent is not a leaf node do
           // abort the traversal if no longer needed
38         if (opData → state) ⇝ status = COMPLETED then
39             return;
           // find the next node to visit
40         if opData → key < dCurrent → key then
41             dCurrent := (dCurrent → left) ⇝ dNode;
42         else
43             dCurrent := (dCurrent → right) ⇝ dNode;
44     if dCurrent → key = opData → key then
45         valData := dCurrent → valData ;                                  // key found
46     else
47         valData := null ;                                                // key not found
48     opData → state := {COMPLETED, valData} ;        // write the result into the operation state
49 executeOperation( opData )
50 begin
51     opData → state := {WAITING, root} ;                 // initialize the operation state
52     MT[myid].opData := opData ;                        // initialize the modify table entry
       // select a modify operation to help later at the end to ensure wait-freedom
53     pid := the process selected to help in round-robin manner;
54     pidOpData := MT[pid].opData;
       // inject the operation into the tree
55     injectOperation( opData );
       // repeatedly execute transactions until the operation completes
56     {status, pCurrent} := read( opData → state );
57     while status ≠ COMPLETED do
58         dCurrent := pCurrent ⇝ dNode;
59         if dCurrent → opData = opData then
60             executeWindowTransaction( pCurrent, dCurrent );
61         {status, pCurrent} := opData → state;
62     if pidOpData ≠ null then
63         injectOperation( pidOpData ) ;                 // help inject the selected operation
```

**Figure 6:** Pseudo-code for MinimalCopy (continued).

(ii) clone to make a copy of a data node (copies all fields except *opData* and *next*), and (iii) create to allocate and initialize an operation record. Note that a data node in our algorithm is an *immutable* object. Once it becomes part of the tree, the contents of its fields never change. Thus, it can be safely copied without any issues. In the pseudo-code, we use *pRoot* to refer to the pointer node of the root of the tree, which never changes. Further, we use the convention that a variable with prefix 'p' represents a pointer node and that with prefix 'd' represents a data node.

There are three places in our algorithm where two fields are packed into a single word so that they can be read and updated atomically: pointer node, next field of data node and state field of operation record. In the pseudo-code, given a word $w$, we use the notation $w \rightsquigarrow f$ to refer to the contents of the field $f$ inside $w$. Further, for convenience, we use the same notation even if $w$ represents the address of a word (*e.g.*, address of a pointer node).

```
64 injectOperation( opData )
65 begin
     // repeatedly try until the operation is injected into the tree
66   while (opData→state)⇝status = WAITING do
67     dRoot := pRoot⇝dNode;
       // execute a window transaction, if needed
68     if dRoot→opData ≠ null then
69       executeWindowTransaction( pRoot, dRoot );
70     dNow := pRoot⇝dNode ;                    // read the address of the data node again
       // if they match, then try to inject the operation into the tree; otherwise restart
71     if dRoot = dNow then
72       dCopy := clone( dRoot );
73       dCopy→opData := opData;
         // try to obtain the ownership of the root of the tree
74       result := CAS( pRoot, {FREE, dRoot}, {OWNED, dCopy} );
75       if result then
           // the operation has been successfully injected; update the operation state
76         CAS( opData→state, {WAITING, pRoot}, {IN_PROGRESS, pRoot} );
```

**Figure 7:** Pseudo-code for MinimalCopy (continued).

For ease of exposition, we assume that there is no reclamation of the memory allocated to nodes that have become garbage and are not "accessible" by any process. This enables us to assume that all pointer and data nodes as well as all value and operation records have unique addresses. However, a *wait-free garbage collection operation* can be easily developed for our algorithm using the well-known notion of *hazard pointers* [29]. Finally, we assume that the tree is never empty and always contains at least one node. This can be ensured by using a sentinel key $\infty$ that is larger than any other key value.

Figure 5 shows the high-level routines for search (lines 1-10), insert/update (lines 11-24) and delete (lines 25-33) operations. As mentioned earlier, modify operations help search operations complete to ensure wait-freedom (lines 18, 23, 29 & 33).

Figure 6 contains the pseudo-code for traversing the tree (lines 34-48), which is used as a helper routine by search and modify operations (lines 6, 23 and 33). Figure 6 also shows the high-level routine for executing an operation in the MTL-framework (lines 49-63). It involves injecting the operation into the tree (line 55) and executing repeated window transactions until it completes (lines 56-61). Finally, to ensure wait-freedom, a modify operation is selected and helped to facilitate its injection into the tree (lines 54 & 63).

Figure 7 contains the pseudo-code for injecting an operation into the tree (lines 64-76) Assume that a process $p$ wants to inject an operation $\alpha$ into the tree and let $\mathcal{OR}(\alpha)$ denote $\alpha$'s operation record. Let $\mathbb{R}$ denote the root of the tree. To inject $\alpha$ into the tree, $p$ first helps any operation that may be sitting at $\mathbb{R}$ move out of the way (line 69). It then attempts to obtain the ownership of $\mathbb{R}$. Let the current data node of $\mathbb{R}$ be $A$. This step involves setting $pnode(\mathbb{R}).flag$ field to OWNED and replacing $A$ with a copy, say $B$, with $B.opData$ set to $\mathcal{OR}(\alpha)$ using a CAS instruction (line 74). If the CAS instruction succeeds, then $\alpha$ has been successfully injected into the tree and $\mathcal{OR}(\alpha).state$ field is updated using a CAS instruction to reflect the same (line 76).

Figure 8 contains the pseudo-code for executing one window transaction (lines 77-110). Assume that a process $p$ wants to execute one window transaction for an operation $\alpha$ whose window is currently rooted at node $X$. Process $p$ first makes a copy of $dnode(X)$ (line 86). It then traverses the window (lines 87-93), helping any operations move out of the way (line 91), and making copies

```
77 executeWindowTransaction( pNode, dNode )
78 begin
       // execute a window transaction for the operation stored in dNode
79     opData := dNode→opData;
80     {flag, dCurrent} := read( pNode ) ;                    // read the contents of pNode again
81     if dCurrent→opData = opData then
82         if flag = OWNED then
83             if pNode = pRoot then
                   // the operation may have just been injected into the tree, but the operation
                      state may not have been updated yet; update the state
84                 CAS( opData→state, {WAITING, pRoot}, {IN_PROGRESS, pRoot} );
85             if not (executeCheapWindowTransaction( pNode, dCurrent )) then
                   // traverse the window using Tarjan's algorithm, making copies as required
86                 windowSoFar := {clone( dCurrent )};
87                 while more nodes need to be added to windowSoFar do
88                     pNextToAdd := the address of the pointer node of the next tree node to be copied;
89                     dNextToAdd := pNextToAdd⇝dNode;
                       // help the operation located at this node, if any, move out of the way
90                     if dNextToAdd→opData ≠ null then
91                         executeWindowTransaction( pNextToAdd, dNextToAdd );
                       // read the address of the data node again as it may have changed
92                     dNextToAdd := pNextToAdd⇝dNode;
93                     copy pNextToAdd and dNextToAdd, and add them to windowSoFar;
94                 window has been copied; now apply transformations dictated by Tarjan' algorithm to
                   windowSoFar;
95                 dWindowRoot :=
                   the address of the data node now acting as window root in windowSoFar;
96                 if last/terminal window transaction then
97                     status := COMPLETED;
98                     pMoveTo :=
                       { the address of the record containing the value   : if an update operation;
                       { null                                             : otherwise;
99                 else
100                    status := IN_PROGRESS;
101                    pMoveTo :=
                       the address of the pointer node of the node in windowSoFar to which the oper-
                       ation will now move;
102                    pMoveTo⇝flag := OWNED;
103                    dMoveTo := pMoveTo⇝dNode;
104                    dMoveTo→opData := opData;
105                dWindowRoot→opData := opData;
106                dWindowRoot→next := {status, pMoveTo};
                   // replace the tree window with the local copy and release the ownership
107                CAS( pNode, {OWNED, dCurrent}, {FREE, dWindowRoot} );
           // at this point, no operation should own pNode; may still need to update the
              operation state with the new position of the operation window
108        dNow := pNode⇝dNode;
109        if dNow→opData = opData then
110            CAS( opData→state, {IN_PROGRESS, pNode}, dNow→next );
```

**Figure 8:** Pseudo-code for MinimalCopy (continued).

```
111 Boolean executeCheapWindowTransaction( pNode, dNode )
112 begin
113     opData := dNode→opDate;
114     pid := opData→pid;
        // traverse the tree window using Tarjan's algorithm
115     while traversal not complete do
116         pNextToVisit := the address of the pointer node of the next tree node to be visited;
117         dNextToVisit := pNextToVisit⇝dNode;
118         if (opData→state)⇝position ≠ pNode then
119             return true;                                   // abort; transaction already executed
            // if there is an operation residing at the node, then help it move out of the way
120         if dNextToVisit→opData ≠ null then
                // there are several cases to consider
121             if (dNextToVisit→opData)→pid ≠ pid then
                    // the operation residing at the node belongs to a different process
122                 executeWindowTransaction( pNextToVisit, dNextToVisit );
                    // read the address of the data node again as it may have changed
123                 dNextToVisit := pNextToVisit⇝dNode;
124                 if (opData→state)⇝position ≠ pNode then
125                     return true;                           // abort; transaction already executed
126             else if dNextToVisit→opData = dNode→opData then
                    // partial window transaction has already been executed; complete it if needed
127                 if (opData→state)⇝position = pNode then
128                     slideWindowDown( pNode, dNode, pNextToVisit, dNextToVisit );
129                 return true;
130             else if MT[pid].opData ≠ opData then
131                 return true;                               // abort; transaction already executed
132             visit dNextToVisit;
133     if no transformation needs to be applied to the tree window then
134         if last/terminal window transaction then
135             pMoveTo := { the address of the record containing the value : if an update operation;
                           { null                                          : otherwise;
136             dMoveTo := null;
137         else
138             pMoveTo := the address of the pointer node of the node in the tree to which the operation
                           will now move;
139             dMoveTo := pMoveTo⇝dNode;
140         if (opData→state)⇝position = pNode then
141             slideWindowDown( pNode, dNode, pMoveTo, dMoveTo );
142         return true;
143     else
144         return false;
```

**Figure 9:** Pseudo-code for MinimalCopy (continued).

of the nodes it visits (line 93). It then applies any required transformations to the local window copy (line 94). Finally, it slides the window down. This involves two steps. Process $p$ first acquires the ownership of the next root of the window, say $Y$ (line 102). Note that, at this time, $Y$ is still part of $p$'s local window and no CAS instruction is required. Process $p$ then releases the ownership of the current root of the window $X$. This involves setting $pnode(X).flag$ field to FREE and updating $pnode(X).dNode$ field with the address of the new data node using a CAS instruction (line 107).

```
145 slideWindowDown( pMoveFrom, dMoveFrom, pMoveTo, dMoveTo )
146 begin
147     opData = dMoveFrom→opData;
        // copy the data node of the current window location
148     dCopyMoveFrom := clone( dMoveFrom );
149     dCopyMoveFrom→opData := opData;
150     if dMoveTo ≠ null then
151         dCopyMoveFrom→next := {IN_PROGRESS, pMoveTo};
152     else
153         dCopyMoveFrom→next := {COMPLETED, pMoveTo};
        // copy the data node of the next window location, if needed
154     if dMoveTo ≠ null then
155         if dMoveTo→opData ≠ opData then
156             dCopyMoveTo := clone( dMoveTo );
157             dCopyMoveTo→opData := opData;
                // acquire the ownership of the next window location
158             CAS( pMoveTo, {FREE, dMoveTo}, {OWNED, dCopyMoveTo} );
        // release the ownership of the current window location and update the operation state
159     CAS( pMoveFrom, {OWNED, dMoveFrom}, {FREE, dCopyMoveFrom} );
160     CAS( opData→state, {IN_PROGRESS, pMoveFrom}, dCopyMoveFrom→next );
```

**Figure 10:** Pseudo-code for MinimalCopy (continued).

If the CAS instruction succeeds, then the window has moved down. However, $\mathcal{OR}(\alpha).state$ still contains the address of $pnode(X)$ because only one variable can be updated using a single-word CAS instruction. The last step is to update $\mathcal{OR}(\alpha).state$ field to store the address of $pnode(Y)$ using a CAS instruction (line 110). To enable any other process to perform the last step, the address of $pnode(Y)$ is stored in the *next* field of the new data node of $X$ (line 106).

Figure 9 contains the pseudo-code for executing a window transaction without copying the window. To execute a cheap window transaction, a process first traverses the window, helping any operations in the way move out of the window (lines 111-144). The process aborts the traversal if it realizes the transaction has already been executed (lines 119, 125 & 131). When the traversal completes and the process determines that no changes need to be made to the window, then it slides the window down.

Figure 10 shows the pseudo-code for sliding the window of an operation down the tree (lines 148-159). It involves acquiring the ownership of the next root of the window and releasing the ownership of the current root of the window . Finally, the process updates the operation's state with the new location of the window (line 160).

We refer to our wait-free algorithm for concurrent red-black tree as MinimalCopy.

# 4 Proof of Correctness

We first show that our algorithm is wait-free. We then show that it only generates linearizable executions.

## 4.1 All Executions are Wait-Free

We first prove that search operations are wait-free. For convenience, we refer to a modify operation that executes phase 2 as *non-trivial* modify operation; otherwise it is a *trivial* modify operation.

Let $N$ denote the number of processes in the system.

**Lemma 1.** *Consider a search operation $\alpha$ belonging to process $p$. Assume that some process, say $q$, completes at least $N$ non-trivial modify operations after $\alpha$ started, and let $\beta$ denote the $N$-th non-trivial modify operation of $q$. Then, by the time $\beta$ completes, $\alpha$ is no longer outstanding.*

*Proof.* Note that, as part of executing phase 2 of a modify operation, a process has to help a search operation of another process that is in progress at the beginning of phase 2 complete. The process selected to help is chosen in a round-robin manner.

Now, consider the non-trivial modify operations executed by $q$ after $\alpha$ started. The first operation of $q$ may have started before $\alpha$ started. However, the next $N-1$ operations of $q$ are guaranteed to have started after $\alpha$ started. Clearly, as part of one of these operations, say $\gamma$, $q$ selects $p$ to help. Clearly, if, at the beginning of phase 2 of $\gamma$, $\alpha$ is still outstanding, then $\gamma$ can complete only after $\alpha$ no longer needs any help. □

The next two lemmas describe the behavior of the system if some search operation were to never complete.

**Lemma 2.** *If a search operation never completes, then eventually no more window transactions are executed in the system.*

*Proof.* Assume that a search operation $\alpha$ never completes. Using the contrapositive of Lemma 1, it follows that, after $\alpha$ has started, no process executes more than $N$ non-trivial modify operations (the last of which may not complete). Clearly, a modify operation generates only a finite number of window transactions during phase 2. Combining the two, it follows that only a finite number of window transactions are executed in the system. □

Let $\mathcal{T}_I$ denote the red-black tree when $\alpha$ dereferences the pointer node of the root of the tree $\mathbb{R}$. Note that, as $\alpha$ proceeds, it may only see a subset of window transactions that are applied to the red-black tree. Specifically, it will only see a window transaction if the window is rooted at the node that is reachable from the node $\alpha$ is currently visiting. Let $visible(\alpha)$ denote the set of window transactions that are applied to $\mathcal{T}_I$ *and are visible to* $\alpha$.

**Lemma 3.** *If a search operation never completes, then the structure traversed by the operation is a finite tree.*

*Proof.* First, note that any window transaction, when applied to a tree, yields a tree. Now, assume that a search operation $\alpha$ never completes. Using Lemma 2, it follows that the set of window transactions visible to $\alpha$, given by $visible(\alpha)$, is finite. This in turn implies that the tree $\mathcal{T}_F(\alpha)$, which is obtained by applying the window transactions in $visible(\alpha)$ (in order in which they are executed in real-time) to the initial tree $\mathcal{T}_I$, is a finite tree. □

We are now ready to show that search operations are wait-free.

**Theorem 4.** *A search operation is wait-free.*

*Proof.* Assume, by the way of contradiction, that some search operation never completes. Then, from Lemma 3, the operation traverses a finite tree. This implies that the operation neither encounters a cycle nor an infinite chain of nodes. This, in turn, implies that the operation eventually reaches a leaf node at which point it terminates—a contradiction. □

Finally, we show that modify operations are wait-free.

**Theorem 5.** *A modify operation is wait-free.*

*Proof.* A modify operation consists of up to three phase. Theorem 4 implies that phase 1 is wait-free. Phase 2 of a modify operation now consists of two sub-phases: executing the operation in the MTL-framework (Phase 2a) and helping a search operation complete (Phase 2b). Tsay and Li's proved in [40] that their framework is wait-free. The main idea is to first show that a modify operation is eventually injected into the tree, and then show that an injected operation eventually completes. The first property holds because of helping and the second property holds because injected modify operations do not overtake each other. The same proof can be easily extended to show that the modified Tsay and Li's framework is also wait-free. This, in turn, implies that phase 2a is wait-free. Clearly, Theorem 4 implies that phase 2b is wait-free. Finally, as proved in [6], phase 3 is also wait-free. □

## 4.2 All Executions are Linearizable

Note that we have four types of operations: search, insert, update and delete. Search operations can be further categorized into two types depending on whether the operation finds the key in the tree or not; we refer to them as *search-hit* and *search-miss*, respectively. A delete operation that does not remove the key from the tree is treated as a search-miss operation.

Consider an execution history of our algorithm. To show that the history is linearizable, we use the *locality property* of linearizability and prove that the history is linearizable on a per key basis. So, given a key, consider the projection of the history on the given key. To show that an execution history (projected on a key) is linearizable, we only consider those operations that have "completed" and remove all other operations from the history. An insert or delete operation is said to have completed once it has executed its terminal window transaction. An update operation is said to have completed once it is deemed to have completed by Chuong *et al.*'s algorithm. Finally, a search operation is said to have completed once the function call associated with the operation returns.

Our proof works as follows. Given a concurrent history $H$ (with incomplete operations removed), we show how to construct an equivalent sequential history $S$ that is legal and respects the real-time order of operations in $H$ (*i.e.*, preserves the relative ordering of non-overlapping operations). To that end, we assign a sequence number to every operation as follows. We assign sequence numbers to insert and delete operations based on the order in which they complete their last window transaction. The first operation in the order is assigned the sequence number one, the second operation is assigned the sequence number two, and so on. The sequence number of an update or search-hit operation is the sequence number of the insert operation that was responsible for creating the record that the (update or search-hit) operation acts on. The sequence number of a search-miss operation is the sequence number of the most recent delete operation to have completed when the search operation completes. (If no such delete operation exists, then the search-miss is assigned the sequence number of zero.) Note that all insert and delete operations have a unique sequence number, but multiple search and/or update operations may have the same sequence number.

We next construct a sequential history $S$ as follows. We first add all insert and delete operations to the history ordering them by their sequence number. We then group all search and update operations based on their sequence number. For a group consisting of search-hit and update operations, we use the linearization order chosen by Chuong *et al.*'s algorithm. (Note that this linearization order will be consistent with the real-time order among its operations.) For a group consisting of search-miss operations, we serialize its operations in any order that is consistent with

the real-time order among its operations. Finally, for each group, we add all its operations to the history immediately after the insert or delete operation with the matching sequence number (or in the beginning if no such operation exists). Hereafter, for convenience, we refer to insert and delete operations as *anchor* operations. So search and update operations are non-anchor operations; they depend on anchor operations for their linearization. Clearly, by our construction, $S$ satisfies the following property:

**Proposition 1.** *The sequential history $S$ contains operations in non-decreasing order of their sequence numbers.*

Note that $H$ and $S$ are trivially equivalent since they contain exactly the same set of operations. We now argue that $S$ is legal and respects the real-time order of operations in $H$. A node is said to *active* if it is reachable from the root of the tree; otherwise, it is said to be *passive*. Tsay and Li's framework satisfies the following property:

**Lemma 6.** *A window transaction of an insert or delete operation is only performed on active nodes in the tree.*

**Lemma 7.** *The sequential history $S$ is legal.*

*Proof.* From Lemma 6, an anchor operation with sequence number $s + 1$ reads-from the anchor operation with sequence number $s$ with $s \geq 1$. This ensures that every anchor operation in $S$ is legal. Further, by our construction, every non-anchor operation is inserted after the anchor operation it depends on in appropriate order with no other anchor operation in between. This implies that every non-anchor operation is legal as well. □

For an operation $\alpha$, let its sequence number be denoted by $seqno(\alpha)$. We have:

**Lemma 8.** *Consider two operations $\alpha$ and $\beta$. We have:*

$$(\alpha \text{ completes before } \beta \text{ starts in } H) \wedge (\alpha \text{ is an anchor operation}) \implies seqno(\alpha) \leq seqno(\beta)$$

*Proof.* The lemma clearly holds if $\beta$ is an anchor operation. In that case, by our construction, $seqno(\beta)$ is guaranteed to be larger than $seqno(\alpha)$. So assume that $\beta$ is a non-anchor (search or update) operation. There are two cases to consider depending on whether $\beta$ is an update/search-hit or search-miss operation.

- **Case 1 ($\beta$ is an update or search-hit operation):** If $\alpha$ is an insert operation, then clearly the record on which $\beta$ acts was created by an insert operation with sequence number at least $seqno(\alpha)$. If $\alpha$ is a delete operation, then the record $\beta$ finds must have been added to the tree by some insert operation that completes after $\alpha$.
- **Case 2 ($\beta$ is a search-miss operation):** If $\alpha$ is a delete operation, then, by our construction, $\beta$ either reads-from $\alpha$ or some other delete operation that completes after $\alpha$. If $\alpha$ is an insert operation, then we have to argue that some delete operation must have completed between when $\alpha$ completes, say $t_\alpha$, and when $\beta$ completes, say $t_\beta$. If no such delete operation exists, then the key that $\beta$ is looking for is present in the tree *continuously* during the time period $[t_\alpha, t_\beta]$. This contradicts with the assumption that $\beta$ was not able to find the key in the tree.

This establishes the lemma. □

We now have:

**Lemma 9.** *The sequential history $S$ preserves the relative ordering of non-overlapping operations in the concurrent history $H$.*

*Proof.* Consider two operations $\alpha$ and $\beta$. We have to show that:

$$\alpha \text{ completes before } \beta \text{ starts in } H \quad \Longrightarrow \quad \alpha \text{ occurs before } \beta \text{ in } S$$

Assume that $\alpha$ completes before $\beta$ starts in $H$. There are two cases depending on whether $\alpha$ is an anchor operation or not.

- **Case 1 ($\alpha$ is an anchor operation:)** Using Lemma 8, $seqno(\alpha) \leq seqno(\beta)$. If $seqno(\alpha) < seqno(\beta)$ then, from Lemma 1, $\alpha$ occurs before $\beta$ in $S$. On the other hand, if $seqno(\alpha) = seqno(\beta)$, then $\beta$ must be a non-anchor operation (that depends on $\alpha$). By our construction, $\beta$ is placed after $\alpha$ in $S$.
- **Case 2 ($\alpha$ is an non-anchor operation:)** Let $\gamma$ be the anchor operation that $\alpha$ depends on. Note that $seqno(\alpha) = seqno(\gamma)$. Clearly, by our construction, $\gamma$ completes before $\alpha$ completes in $H$. Therefore, $\gamma$ completes before $\beta$ starts in $H$. Using Lemma 8, $seqno(\gamma) \leq seqno(\beta)$. This in turn implies that $seqno(\alpha) \leq seqno(\beta)$. As before, if $seqno(\alpha) < seqno(\beta)$ then, from Lemma 1, $\alpha$ occurs before $\beta$ in $S$. On the other hand, if $seqno(\alpha) = seqno(\beta)$, then $\beta$ must be a non-anchor operation as well. By our construction, when we add $\alpha$ and $\beta$ to $S$ (note both operations will be added to $S$ as part of the same group), we ensure that we respect the real-time order between $\alpha$ and $\beta$.

This establishes the lemma. □

Therefore, we have the following result:

**Theorem 10.** *Our wait-free algorithm only generates linearizable executions.*

# 5 A Wait-Free Algorithm for Memory Reclamation

In our wait-free algorithms for a red-black tree, as modify operations traverse the tree, they replace the existing nodes in the tree with new copies. A node, on being replaced, becomes passive, is no longer reachable from the root of the tree, and therefore not accessible to any operation that starts thereafter. However, such nodes still occupy memory, which needs to be reclaimed to enable the data structure to be used for longer periods. Otherwise, the system will run out of memory fairly quickly.

Note that nodes that never become active because they are part of a local window of a failed transaction can be garbage collected immediately. A transaction fails if the window of the tree it is trying to replace has already been replaced by some other process. Reclaiming memory of nodes that were active earlier but are now passive is tricky because search operations rely on such nodes to reduce their overhead. Further, although modify operations during phase 2 only "act" on active nodes, it is still possible for a modify operation to dereference a passive node. This happens when a process is making a copy of a window of the tree but the window has already been replaced by a new window by some other process. So, all nodes in the original window have become passive.

Our algorithm for garbage collection is based on the notion of *hazard pointers* introduced by Michael in [29]. The algorithm is designed with the objective that a search operation incurs minimal overhead and process invoking it does not have to execute any atomic (synchronization) instruction.

We now describe the modifications that need to be made to search and modify operations to support memory reclamation of passive nodes. We then describe a wait-free garbage collection algorithm.

### 5.1 Maintaining Hazard Pointer Lists

Each process maintains a separate hazard pointer list that contains addresses of nodes that the process may dereference in the future and hence should not be garbage collected. A hazard pointer list is implemented as a doubly linked list of nodes; nodes can be added to but not removed from the list. Each node either stores an address of a (pointer, data or record) node or is empty (in which case it stores the sentinel value $\perp$). (Two bits in the address can be used to distinguish between the three types of addresses.) Since a list can only be modified by its owner (but can be read by any process), it can be manipulated using simple read and write instructions.

Our algorithm (explained later) works in a way that addresses are removed from a hazard pointer list in the reverse order of which they were added (last-in-first-out). As a result, a hazard pointer list basically has the following structure: a list of non-empty nodes followed by a list of empty nodes. We need to support the following operations on a hazard pointer list: (i) add an address to the list, (ii) remove the last added address from the list, and (iii) take a snapshot of all non-empty nodes in the list. The first two operations are performed by search and modify operations, where as the last one is performed by garbage collection operations. To support the three operations, a process maintains a pointer to the first empty node in the list, referred to as *current*. All nodes to the left of *current* are non-empty and all nodes to the right of *current* (including the one pointed to by *current*) are empty. Initially, the list contains a single empty node with *current* pointing to that node.

Adding an address to a list involves the following steps: (i) store the address in the node pointed to by *current*, (ii) insert a new node into the list if the node pointed to by *current* is the last node in the list, and (iii) move *current* to the right by one node. Clearly, an add operation has time complexity of $O(1)$.

Removing the last added address from a list involves the following steps: (i) move *current* to the left by one node, and (ii) store sentinel value in the node pointed to by *current*. Clearly, a remove operation has time complexity of $O(1)$.

Finally, taking a snapshot of all non-empty nodes in a list involves the following steps: (i) read the value of *current*, and (ii) read the address field of all nodes to the left of *current*. Clearly, a snapshot operation has time complexity linear in the number of non-empty nodes in the list at the time it reads the value of *current*.

### 5.2 Changes Made to Search and Modify Operations

The main idea is that make sure that a process never dereferences a passive node whose memory has been reclaimed. Note that a process needs to dereference a node as part of performing some action (*e.g.* to help a search operation complete, to help a window transaction of a modify operation complete in the TL-framework). So, before a process dereferences a node, it first adds its address to its hazard pointer list, ascertains that the the action still needs to be performed by *testing* for some conditions, and dereferences the node only if the test succeeds. This works because, when a process retires a node, it would have already completed the action whose execution requires the node to be dereferenced. For a search operation, the address of a node can be removed from the hazard pointer list after it has been dereferenced. For a window transaction of a modify operation, addresses of all nodes in the window can be removed after the window transaction has completed.

We now discuss the test that a process, say $q$, can use to determine if an action of an operation, say $\alpha$, has not yet completed. Assume that the operation $\alpha$ belongs to process $p$. If $\alpha$ is a search operation, then $q$ examines the entry for $p$ in the search table and ascertains that the address of the operation record in the entry matches that of $\alpha$ and the status field in the operation record has

the value WAITING. If $\alpha$ is a modify operation, then there are two cases depending on the type of node that $q$ wants to dereference: a pointer node or a data node. For a pointer node, we can further categorize the actions performed on behalf of a modify operation into three types: (i) $q$ is trying to inject $\alpha$ into the tree (and thus needs to dereference the pointer node of the root of the tree), (ii) $q$ is trying to copy nodes in $\alpha$'s current window (so that it can execute a window transaction and slide $\alpha$'s window down), or (iii) $q$ trying to obtain the current location of $\alpha$'s window (in which case $q$ is same as $p$). The third case occurs because some other process, different from $p$, may execute one or more of $\alpha$'s window transactions, so now $p$ has to "find its way back" into the tree. Process $q$ determines that a given action of $\alpha$ has not yet been performed as follows. In the first case, $q$ can safely dereference the pointer node because the pointer node of the root of the tree is never retired. In the second case, let $R$ denote the pointer node currently owned by $\alpha$ (and hence corresponds to the current location of $\alpha$'s window). Note that $q$ would have already read the contents of $R$ once and stored them in its local memory. Process $q$ reads the contents of $R$ again once to ascertain that they still match the contents stored in its local memory. Further, $q$ examines $p$'s modify table entry to ascertain that the address of the operation record still mataches that of $\alpha$. Finally, in the third case, $p$ (which is same as $q$) examines its own entry in the modify table to ascertain that the status field in $\alpha$'s operation record has not changed. For a data node, there is an extra test condition in addition to those described above. Let the data node be $A$. Note that $q$ would have already dereferenced the pointer node associated with $A$, say $B$. The extra test condition is that the contents of $B$ have not changed.

Note that the changes described above change the time-complexity of search and modify operations by a constant factor only.

### 5.3 Reclaiming Memory of Garbage Nodes

When an active node becomes passive, we say that the node has *retired*. As in [29], the process that retires a node is responsible for reclaiming the memory occupied by the node. Each process maintains a list of all nodes that it has retired but whose memory has not been reclaimed yet. Once the length of the list exceeds a certain threshold at a process, it invokes a garbage collection operation. The operation consists of the following steps:

1. Help all search operations currently in-progress complete using the approach described in Section 3.

2. Take a snapshot of the following: (a) hazard pointer lists of all processes, (b) modify table entries of all processes, (c) search table entries of all processes, and (d) update table entries of all processes. Note that any node whose address is contained in one of the snapshots cannot be garbage collected. Further, note that, if a data node cannot be garbage collected, then the record whose address is stored in that data node cannot be garbage collected as well.

3. Reclaim the memory of all nodes in the retired list except for those that cannot be garbage collected as described above.

Note that each of the above steps takes a finite amount of time to execute. This implies that every garbage collection operation eventually terminates and is therefore wait-free.

Note that, in practice, instead of freeing the memory of garbage collected nodes, a process can *recycle* them and use them when executing modify operations.

```
161 reclaimMemory( )
162 begin
        // help all search operations complete
163     foreach process p do
164         s := ST[p].seqno;
165         k := ST[p].key;
166         if ST[p].help and ST[p].seqno = s then
167             traverse( k, p, s );
        // scan all hazard pointer lists
168     foreach process p do
169         foreach address a in p's hazard pointer list do
170             add a to doNotReclaimList;
171             if a is the address of a data node then add a→record to doNotReclaimList;
172             ;
        // scan all entries in the search, modify and update tables
173     foreach process p do
174         add ST[p].result to doNotReclaimList;
175         add MT[p].address to doNotReclaimList;
176         add UT[p].record to doNotReclaimList;
            // UT denotes the update table
        // perform garbage collection
177     reclaim memory of all nodes whose addresses are present in retiredList but not in doNotReclaimList;
```

**Figure 11:** Garbage collection operation.

### 5.4 Proof of Correctness

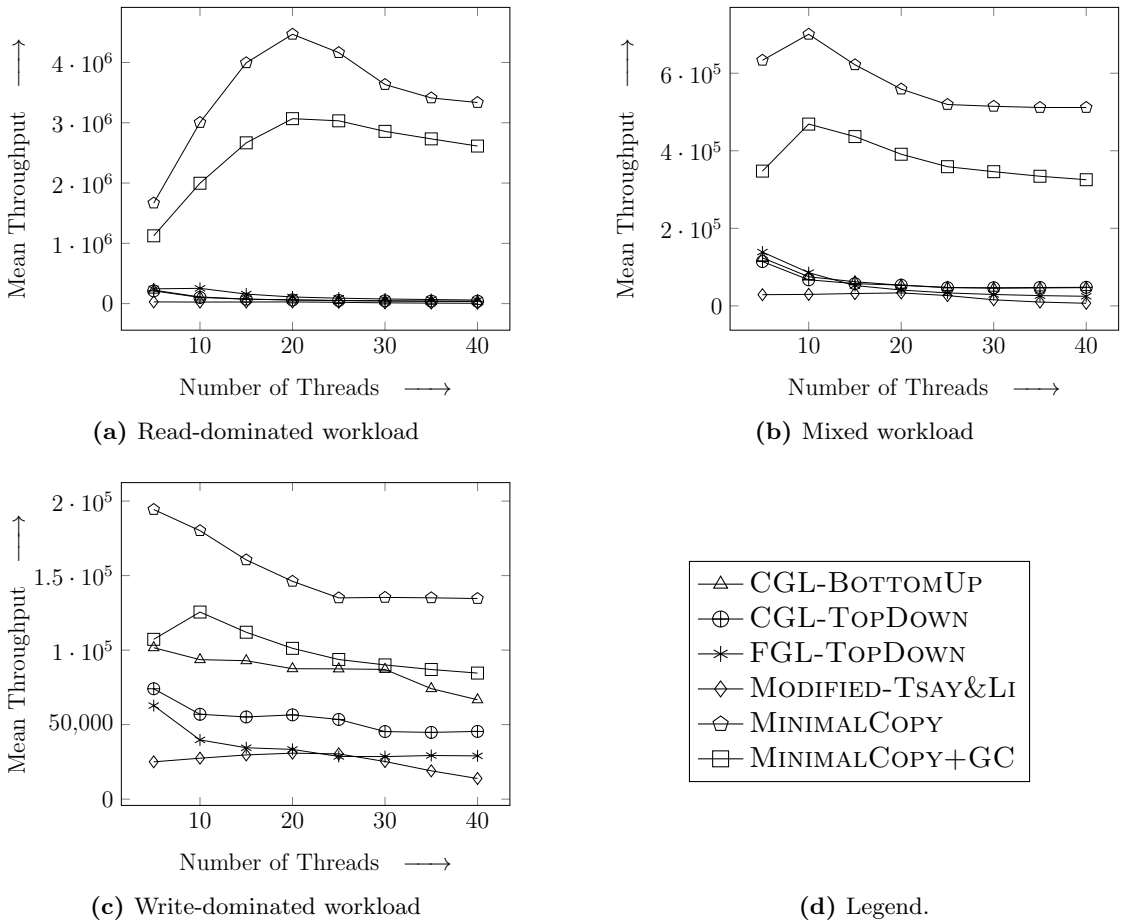
We show that a process never dereferences a garbage-collected node. We have:

**Theorem 11.** *A process never dereferences a node whose memory has already been reclaimed.*

*Proof Sketch.* Consider a process $p$ that needs to dereference a node $X$ in order to perform an action. Assume that node $X$ has been retired by process $q$. Further, assume that process $q$ invokes a garbage collection operation $\alpha$ sometime after retiring $X$. As explained earlier, before dereferencing $X$, $p$ adds the address of $X$ to its hazard pointer list. As part of executing $\alpha$, $q$ takes a snapshot of $p$'s hazard pointer list. If $q$ finds the address of $X$ in the list, then $q$ does not garbage collect $X$. So, assume that $q$ does not find the address of $X$ in the list. It implies that $q$'s snapshot either finished before $p$ was able to add the address of $X$ to its list or started after $p$ removed the address of $X$ from its list. In the first case, $p$ would detect that the action has already been performed and therefore will not dereference $X$. In the second case, $p$ dereferences $X$ before $q$ reclaims $X$'s memory. □

## 6 Experimental Evaluation

**Other Concurrent Red-Black Tree Implementations:** For our experiments, we considered four other implementations of concurrent red-black tree besides the two based on MinimalCopy and MinimalCopy+GC: (i) two based on coarse-grained-locking (using the standard bottom-up and the Tarjan's top-down approaches), denoted by CGL-BottomUp and CGL-TopDown, (ii) one based on fine-grained-locking (using the Tarjan's top-down approach), denoted by FGL-TopDown and (iii) one based on Tsay and Li's framework (modified to use one-word pointer




**(a)** Read-dominated workload

**(b)** Mixed workload

**(c)** Write-dominated workload

**(d)** Legend.

**Figure 12:** Comparison of throughput (in operations per second) of different implementations of red-black tree.

nodes and CAS instructions), denoted by Modified-Tsay&Li. We did not implement Kim *et al.*'s algorithm for concurrent red-black tree because some important details about the algorithm are missing in the description given in [27]. For example, it is not clear how a search operation works. It appears that it cannot simply traverse the tree as in [28] because the tree is modified in-place using multiple CAS instructions and thus may be in an inconsistent state at times. Besides these, we are not aware of any other practical direct implementation of a concurrent red-black tree.

In all three lock-based implementations, a tree node is a singular entity and not split into pointer and data nodes, and the value associated with a key is stored inside a node and not outside in a record. Windows are modified in-place. Note that all the above changes improve the performance of lock-based implementations by reducing indirection and reducing copying. In CGL-BottomUp and CGL-TopDown, the entire tree is protected using a single lock that supports two different locking modes: shared (for search operations) and exclusive (for modify operations). Thus the only concurrency is due to search operations; modify operations execute in serial manner. In FGL-TopDown, a lock is associated with each tree node, which again supports shared and exclusive locking modes. A search operation starts by locking the root of the tree in shared mode, and releases the lock on the currently locked node after locking the next node in its path in shared mode or upon terminating. A modify operation starts by locking the root of the tree in exclusive mode, and releases the lock on the root of the current window after locking the root of the next

window in exclusive mode or upon terminating. Finally, in both top-down implementations CGL-TopDown and FGL-TopDown, as in MinimalCopy, search operations are used to speedup modify operations.

**Experimental Setup:** We conducted our experiments on a dual-processor AMD Opteron 6180 SE 2.5 GHz machine, with 12 cores per processor (yielding 24 cores in total), 64 GB of RAM and 300 GB of hard disk, running 64-bit Linux operating system. All implementations were written in C++. We used Boost C++ library [10] for locks and Google's TCMalloc library [16] for dynamic memory allocation.

To compare the performance of different implementations, we considered the following parameters:

1. **Maximum Tree Size:** This depends on the size of the key space. We considered three different key space sizes of 10,000 (10K), 100,000 (100K) and 1 million (1M) keys. To ensure consistent results, as in [37], rather than starting with an empty tree, we populated the tree to a 50% *full* value prior to starting the simulation run.
2. **Relative Distribution of Various Operations:** We considered three different workload distributions: (a) *Read-dominated workload:* 90% search, 9% insert/update and 1% delete (b) *Mixed workload:* 70% search, 20% insert/update and 10% delete (c) *Write-dominated workload:* 20% search, 40% insert/update and 40% delete
3. **Maximum Degree of Contention:** This depends on the number of threads. We varied the number of threads from 5 to 40 in steps of 5.

We compared the performance of different implementations with respect to system throughput, which is given by the number of operations executed per unit time.

**Evaluation Results:** The results of our experiments are shown in Figure 12. Each test was carried out for 60 seconds and the results were averaged over multiple runs to obtain values within 99% confidence interval. The results for the three key space sizes are very similar to each other; due to space limitations, we only show the results for the 100K key space size.

As the graphs show, MinimalCopy and MinimalCopy+GC are the top two performers among all implementations for all workloads. Between the two, MinimalCopy+GC has 20%-45% lower throughput than MinimalCopy indicating that garbage collection has significant overhead. (In our experiments, we set the threshold for garbage collection at 25,000 nodes per thread.) For read-dominated workloads , the third best performer is FGL-TopDown, whereas for mixed and write-dominated workloads, it is CGL-BottomUp. For read-dominated workloads, MinimalCopy+GC has 450%-4,500% (or 4.5-45 *times*) better throughput than FGL-TopDown. For mixed workloads, MinimalCopy+GC has 250%-760% (or 2.5-7.6 *times*) better throughput than CGL-BottomUp. Finally, for write-dominated workloads, the gap between MinimalCopy+GC and CGL-BottomUp is much less; MinimalCopy+GC has only 3.4%-34% better throughput than CGL-BottomUp. For both read-dominated and mixed workloads, search as well as modify operations of MinimalCopy+GC are several times faster than those of the next best performer. Finally, we would like to note that Modified-Tsay&Li has the *worst-performance* among all implementations for all workloads.

# 7 Conclusion

In this paper, we have presented an efficient wait-free algorithm for a concurrent red-black tree that minimizes copying of nodes and supports fast search operations. A search operation can not only

execute concurrently with other search operations but with modify operations as well. Further, unlike as in universal constructions, modify operations working on different parts of the tree can execute concurrently after their paths in the tree have diverged. Our experiments indicate that our algorithm has much better performance than other concurrent algorithms for a red-black tree. Its utility can be enhanced by augmenting it with a wait-free garbage collection operation that reclaims memory allocated to nodes that are no longer "accessible" by any process [32].

# References


[1] M. A. Bender et al. "Concurrent Cache-Oblivious B-Trees". In: *Proceedings of the 17th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. Las Vegas, Nevada, USA, July 2005, pp. 228–237.

[2] M. Blasgen et al. "The Convoy Phenomenon". In: *Operating Systems Review (OSR)* 13.2 (Apr. 1979), pp. 20–25.

[3] A. Braginsky and E. Petrank. "A Lock-Free B+tree". In: *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. Pittsburgh, Pennsylvania, USA, 2012, pp. 58–67.

[4] N. G. Bronson et al. "A Practical Concurrent Binay Search Tree". In: *Proceedings of the 15th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*. Bangalore, India, Jan. 2010, pp. 257–268.

[5] T. Brown and J. Helga. "Non-Blocking k-ary Search Trees". In: *Proceedings of the International Conference on Principles of Distributed Systems (OPODIS)*. 2011, pp. 207–221.

[6] P. Chuong, F. Ellen, and V. Ramachandran. "A Universal Construction for Wait-Free Transaction Friendly Data Structures". In: *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. Thira, Santorini, Greece, 2010, pp. 335–344.

[7] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. Cambridge, Massachusetts, USA: The MIT Press, 1991.

[8] T. Crain, V. Gramoli, and M. Raynal. "A Speculation-Friendly Binary Search Tree". In: *Proceedings of the 17th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*. New Orleans, Louisiana, USA, 2012, pp. 161–170.

[9] M. David. "A Single-Enqueuer Wait-Free Queue Implementation". In: *Proceedings of the Symposium on Distributed Computing (DISC)*. Amsterdam, The Netherlands, Oct. 2004, pp. 132–143.

[10] B. Dawes and D. Abrahams. *Boost C++ Libraries*. URL: `http://www.boost.org`.

[11] F. Ellen et al. "Non-Blocking Binary Search Trees". In: *Proceedings of the 29th ACM Symposium on Principles of Distributed Computing (PODC)*. Zurich, Switzerland, July 2010, pp. 131–140.

[12] P. Fatourou and N. D. Kallimanis. "A Highly-Efficient Wait-Free Universal Construction". In: *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. San Jose, California, USA, 2011, pp. 325–334.

[13] M. Fomitchev and E. Ruppert. "Lock-Free Linked Lists and Skiplists". In: *Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing (PODC)*. St. John's, Newfoundland, Canada, July 2004, pp. 50–59.

[14] K. Fraser. "Practical Lock-Freedom". PhD thesis. University of Cambridge, Feb. 2004.

[15] K. Fraser and T. L. Harris. "Concurrent Programming Without Locks". In: *ACM Transactions on Computer Systems* 25.2 (May 2007).

[16] S. Ghemawat and P. Menage. *TCMalloc: Thread-Caching Malloc*. URL: `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`.

[17] L. J. Guibas and R. Sedgewick. "A Dichromatic Framework for Balanced Trees". In: *Proceedings of the 19th Annual Symposium on Foundations of Computer Science (FOCS)*. Oct. 1978, pp. 8–21.

[18] T. Harris. "A Pragmatic Implementation of Non-blocking Linked-lists". In: *Distributed Computing (DC)* (2001), pp. 300–314.

[19] M. Herlihy. "A Methodology for Implementing Highly Concurrent Objects". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 15.5 (1993), pp. 745–770.

[20] M. Herlihy. "Wait-Free Synchronization". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13.1 (Jan. 1991), pp. 124–149.

[21] M. Herlihy, V. Luchangco, and M. Moir. "Obstruction-Free Synchronization: Double-Ended Queues as an Example". In: *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems (ICDCS)*. 2003, pp. 522–529.

[22] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008. ISBN: 0123705916, 9780123705914.

[23] M. Herlihy and J. M. Wing. "Axioms for Concurrent Objects". In: *Proceedings of the 14th ACM Symposium on Principles of Programming Languages (POPL)*. Munich, West Germany, 1987, pp. 13–26.

[24] S. V. Howley and J. Jones. "A Non-Blocking Internal Binary Search Tree". In: *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. Pittsburgh, Pennsylvania, USA, June 2012, pp. 161–171.

[25] M. T. Jones. *Inside the Linux 2.6 Completely Fair Scheduler*. Dec. 2009. URL: `http://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler/`.

[26] J. H. Kim. "High-Concurrency Lock-Free Algorithms for Red-Black Trees". MA thesis. The University of Manitoba, Canada, Aug. 2005.

[27] J. H. Kim, H. Cameron, and P. Graham. "Lock-Free Red-Black Trees Using CAS". In: *Concurrency and Computation: Practice and Experience* (2006), pp. 1–40.

[28] J. Ma. "Lock-Free Insertions on Red-Black Trees". MA thesis. The University of Manitoba, Canada, Oct. 2003.

[29] M. M. Michael. "Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects". In: *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 15.6 (2004), pp. 491–504.

[30] M. M. Michael. "High Performance Dynamic Lock-Free Hash Tables and List-based Sets". In: *Proceedings of the 14th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. Winnipeg, Manitoba, Canada, 2002, pp. 73–82.

[31] M. M. Michael and M. L. Scott. "Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms". In: *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing (PODC)*. Philadelphia, Pennsylvania, USA, 1996, pp. 267–275.

[32] A. Natarajan, L. Savoie, and N. Mittal. *Concurrent Wait-Free Red Black Trees*. Tech. rep. UTDCS-16-12. Department of Computer Science, The University of Texas at Dallas, Oct. 2012. URL: `http://www.utdallas.edu/~aravindn/waitfree-rbt.pdf`.

[33] S. Prakash, Y.-H. Lee, and T. Johnson. "A Nonblocking Algorithm for Shared Queues using Compare-And-Swap". In: *IEEE Transactions on Computers* 43.5 (May 1994), pp. 548–559.

[34] S. Prakash, Y.-H. Lee, and T. Johnson. *Non-Blocking Algorithms for Concurrent Data Structures*. Tech. rep. 91-002. University of Florida, 1991.

[35] A. Prokopec et al. "Concurrent Tries with Efficient Non-Blocking Snapshots". In: *Proceedings of the 17th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*. New Orleans, Louisiana, USA, 2012, pp. 151–160.

[36] R. Sedgewick. *Left-leaning Red-Black Trees*. URL: `http://www.cs.princeton.edu/~rs/talks/LLRB/LLRB.pdf`.

[37] M. F. Spear et al. "Transactional Mutex Locks". In: *Proceedings of the 4th Workshop on Transactional Computing (TRANSACT)*. Feb. 2009.

[38] H. Sundell and P. Tsigas. "Scalable and Lock-Free Concurrent Dictionaries". In: *Proceedings of the 19th Annual Symposium on Selected Areas in Cryptography*. Nicosia, Cyprus, Mar. 2004, pp. 1438–1445.

[39] R. E. Tarjan. *Efficient Top-Down Updating of Red-Black Trees*. Tech. rep. TR-006-85. Department of Computer Science, Princeton University, 1985.

[40] J.-J. Tsay and H.-C. Li. "Lock-Free Concurrent Tree Structures for Multiprocessor Systems". In: *Proceedings of the International Conference on Parallel and Distributed Systems (ICPADS)*. Hsinchi, Taiwan, Dec. 1994, pp. 544–549.

[41] J. D. Valois. "Lock-Free Linked Lists using Compare-And-Swap". In: *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing (PODC)*. Atlanta, Georgia, USA, 1999, pp. 214–222.