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 concurrent manipulation of 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 for most workloads.

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 it will 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 [1].

Non-blocking algorithms avoid the pitfalls of locks by using special (hardware-supported) *read-modify-write* instructions such as *load-link/store-conditional (LL/SC)* [1] [2] and *compare-and-swap (CAS)* [2] [1]. Non-blocking implementations of

* 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 Proceedings of the 26th International Symposium for Distributed Computing (DISC), pages 421–422, 2012.

[1] A load-link instruction returns the current value of a memory location; a subsequent store-conditional instruction to the same location will store a new value only if no updates have occurred to that location since the load-link was performed.

[2] A compare-and-swap instruction compares the contents of a memory location to a given value and, only if they are the same, modifies the contents of that location to a given new value.

T. Higashino et al. (Eds.): SSS 2013, LNCS 8255, pp. 45–60, 2013.

common data structures such as queues, stacks, linked lists, hash tables, and search trees have been proposed [1, 3–8].

Non-blocking algorithms may provide varying degrees of progress guarantees [1]. Three widely accepted progress guarantees are: obstruction-freedom, lock-freedom, and wait-freedom. An algorithm is said to be *obstruction-free* if any process that executes in isolation will finish its operation in a finite number of steps. It is said to be *lock-free* if some process will complete its operation in a finite number of steps. Finally, it is said to be *wait-free* if every process will complete its every operation 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 [9]. Red-black tree is a type of self-balancing binary search tree that provides good worst-case time complexity for all tree operations. As a result, they are used in symbol table implementations within systems like C++, Java, Python and BSD Unix [10]. They are also used to implement completely fair schedulers in Linux kernel [11]. 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 [1, 12, 13]. Due to the general nature of the constructions, when applied to a binary search tree, the resultant data structure is quite inefficient. This is because universal constructions 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 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 [4–7], and balanced search trees such as B-tree [3] and B^+-tree [8] have been proposed, that are more efficient than those obtained using universal constructions.

In [14], Ma presented a "lock-free" algorithm for a concurrent red-black tree that supports search and insert operations using CAS, DCAS (double-word[3] CAS) and TCAS (triple-word[3] CAS) instructions [14]. Kim *et al.* extended Ma's algorithm to support delete operations as well as eliminate the use of multi-word CAS instructions [15]. However, a closer inspection of the algorithm reveals that it is actually a blocking algorithm. It is only lock-free in the sense that CAS instructions are used for synchronization (setting and unset-ting flags at nodes) and no "locks" are used. But, if a process blocks while holding the flag on the root of the tree, *all other processes* will be prevented from making progress. Concurrent algorithms for a red-black tree based on the transactional memory framework have also been proposed (*e.g.*, [16, 17]). The algorithm in [17] maintains a relaxed red-black tree in which the balance requirements of a red-

[3] Words need not be adjacent.

black tree may be violated temporarily. In contrast to the aforementioned algorithms, our algorithm has the following desirable properties: (a) it uses only single word CAS instruction, which is commonly available in many hardware architectures including Intel 64 and AMD64, (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 [18] 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, it 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. 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.

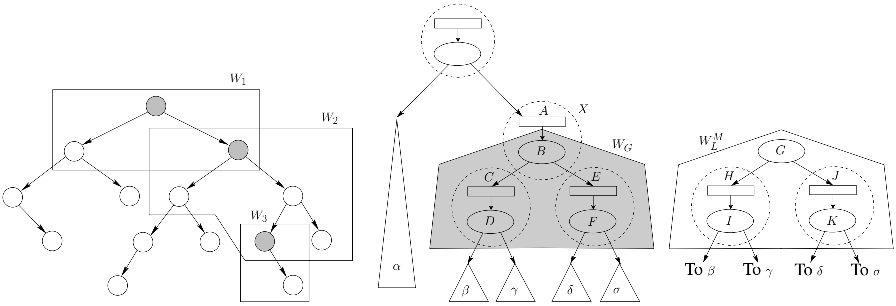
Contributions: In this paper, we present a new *wait-free* algorithm for concurrent manipulation of a red-black tree in an asynchronous shared memory system that supports search, insert, update and delete operations using single-word CAS instructions. Search operations in our algorithm are fast and perform 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. 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.

2 Preliminaries

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

Tsay and Li described a framework in [18] (or TL-framework for short) that can be used to develop wait-free operations for a tree-based data structure provided operations work on 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 structure, that is, a small, contiguous piece of the tree. We say that a window is *located* at its root node. The execution of a top-down operation can be modeled using a sequence of windows starting from the root and ending at a leaf of the tree. For example, Fig. 1(a) shows a sequence of three windows W_1, W_2 and W_3; the shaded nodes denote the root node of the respective 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 TL-framework, when an operation starts, it first needs to be "injected" into the tree. This involves obtaining the ownership of the root of the tree. This step "initializes" the first window of the operation. Thereafter the operation performs a sequence of window transactions until it reaches the bottom of the tree at which point the it terminates. Consecutive windows of an operation always *overlap*. The root of the next window is



Overlapping windows of a top-down operation.

Dual node structure of a tree node.

Fig. 1. Windows in Tsay and Li's framework

part of the current window. For an example, see Fig. 1(a). 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 the TL-framework. To execute a window transaction of an operation α with current window W_G in the tree, a process p needs to perform the following four steps:

1. *Explore-Help-And-Copy:* In this step, p traverses nodes in W_G starting from the root node of the window. On visiting a node X (in W_G), if p finds that X is owned by another operation β, then p helps β "move out" of α's way by performing a window transaction on β's behalf. As p traverses W_G, it also makes its copy, 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.*, performing rotations). Let W_L^M denote the window obtained after applying 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 α (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, α'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:* Let α belong to process q, where q may be p. In this step, p announces the location of α's new window to other processes in the system by updating q's (process) table entry using a synchronization instruction. It is possible for this step to be performed by another process, say r, where r may be different from both p and q, since α'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.

Consider a window rooted at some tree node, say X. A window transaction may involved changing multiple attributes of X (*e.g.*, color, key and/or children pointers). This, in general, cannot be performed using a single synchronization instruction. To address this problem, a tree node in the TL-framework has *dual* structure; it consists of a *pointer* node and a *data* node. The pointer node contains a reference to the data node, and information about whether the tree node (it represents) is owned by some operation. 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. For example, in Fig. 1(b), window W_G is rooted at tree node X with pointer and data nodes as A and B, respectively, and W_L^M denotes a transformed copy of W_G. Now, W_G can be replaced with W_L^M by changing the reference stored in A from B to G.

The TL-framework 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, no hardware currently implements such an instruction. 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 a single-word CAS instruction, which is widely available in hardware. Hereafter, we refer to the TL-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 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.

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). In [19], Tarjan proposed algorithms for insert and delete operations that work in a top-down manner 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. For more details of insert and delete operations, including invariants they maintain and various transformations they use to keep the tree balanced, please refer to [19].

3 A Wait-Free Algorithm for Red-Black Tree

We now describe how to reduce the overhead of search and modify operations in the MTL-framework to obtain a more efficient wait-free algorithm for a 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 [19], yields a wait-free red-black tree. But the resulting 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. This copying and helping makes an operation *expensive* to execute. Besides, 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 at most one operation at a time. This means that concurrently invoked search operations may 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 them. This is because a process makes change to its *local* window first and then installs it in the tree using a *single* CAS instruction at which time it becomes accessible to 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 search and modify operations without interfering with them. Note that, as a modify operation traverses the tree from from top to bottom, it replaces all nodes in the current window with new copies before moving down. Thus, as a search operation proceeds, it may encounter nodes that are no longer part of the tree. Nevertheless, we show that the result of a search operation is still meaningful, that is, our algorithm only generates *linearizable* histories [20].

3.2 Reducing the Overhead of Modify Operation

We reduce the overhead of a modify operation in two ways, which are described one-by-one as follows.

Minimizing the Use of the MTL-Framework: By reducing the overhead of a search operation, we can also reduce the overhead of a modify operation 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 [1]. 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 then changed to return the address of the record (containing the value) if it finds the given key in the tree.

An insert/update operation consists of 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, then the key along with its associated value are added to the tree using the expensive MTL-framework, (c) *Phase 3:* If the key already exists in the tree, then the value stored in the record associated with the key is updated outside the MTL-framework. Note that an operation in phase 2 may find that the key already exists in the tree due to concurrent modifications to the tree. In that case, the insert operation becomes an update operation after completing its phase 2 and then executes phase 3 as well. To accomplish this, we modify the MTL-framework to return the address of the record in case the key is already present in the tree.

A delete operation consists of 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 key and its associated value are removed from the tree using the expensive MTL-framework.

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 proposed by Chuong *et al.* in [12]. 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. However, to reduce the space-complexity, 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.

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 [19]. 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. 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

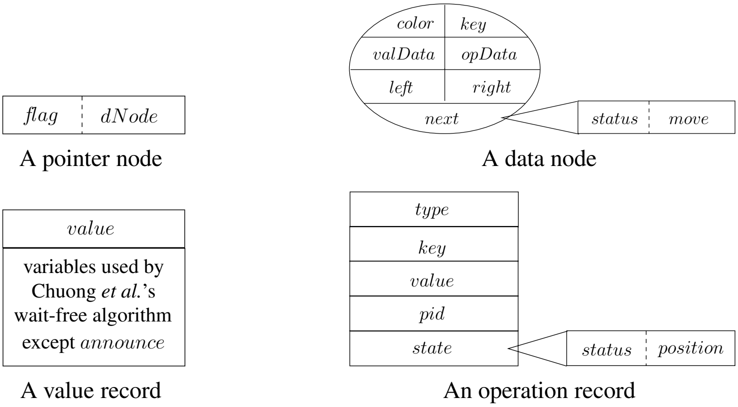


Fig. 2. Data structures used by our algorithm

to help a search operation complete. To that end, whenever a process executes a modify operation, 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 as shown in Fig. 2: (1) *pointer node* that stores reference to the data node, (2) *data node* that stores tree node attributes, (3) *value record* that stores the value associated with the key, and (4) *operation record* that stores information about the operation such as its type, arguments and current state.

A pointer node, which is a single word, contains the following fields: (a) $flag$: a bit that indicates whether the node is owned by an operation, and (b) $dNode$: the address of the data node. The $flag$ field 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 window), and (d) $next$: information about the operation after executing window transaction (only relevant if the node was the root of some window); it contains two sub-fields (packed into a single word): (i) $status$: the new status of the operation, and (ii) $move$: the address of the next location of the operation's 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's window. In case of

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

insertOrUpdate( key, value )
begin
    valData := null;
    // phase 1: determine if the key already exists in the tree
    search( key );
    valData := (ST[myid] → state) ⇝ position;
    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
        pid := the process selected to help in round-robin manner;     pidOpData := ST[pid];
        opData := create( INSERT, key, value ) ;                // create a new operation record
        executeOperation( opData ) ;                            // add the key-value pair to the tree
        valData := (opData → state) ⇝ position;
        if pidOpData ≠ null then
            traverse( pidOpData ) ;                             // help the selected search operation complete
    if valData ≠ null then
        // phase 3: update the value in the record using Chuong et al.'s algorithm

delete( key )
begin
    // phase 1: determine if the key already exists in the tree
    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
        pid := the process selected to help in round-robin manner;     pidOpData := ST[pid];
        opData := create( DELETE, key, ⊥ ) ;                    // create a new operation record
        executeOperation( opData ) ;                            // remove the key from the tree
        if pidOpData ≠ null then
            traverse( pidOpData ) ;                             // help the selected search operation complete

traverse( opData )
begin
    dCurrent := pRoot ⇝ dNode ;                                 // start from the root of the tree
    while dCurrent is not a leaf node do
        // abort the traversal if no longer needed
        if (opData → state) ⇝ status = COMPLETED then return;
        // find the next node to visit
        if opData → key < dCurrent → key then dCurrent := (dCurrent → left) ⇝ dNode;
        else dCurrent := (dCurrent → right) ⇝ dNode;
    // check if the two keys match
    if dCurrent → key = opData → key then valData := dCurrent → valData;
    else valData := null;
    opData → state := {COMPLETED, valData} ;                    // update the operation state
```

Fig. 3. Pseudo-code for MINIMALCOPY

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

```
executeOperation( opData )
begin
    opData → state := {WAITING, root} ;                         // initialize the operation state
    MT[myid] := opData ;                                        // initialize the modify table entry
    // select a modify operation to help later at the end
    pid := the process selected to help in round-robin manner;     pidOpData := MT[pid];
    // inject the operation into the tree
    injectOperation( opData );
    // repeatedly execute transactions until the operation completes
    {status, pCurrent} := read( opData → state );
    while status ≠ COMPLETED do
        dCurrent := pCurrent ⇝ dNode;
        if dCurrent → opData = opData then
            executeWindowTransaction( pCurrent, dCurrent );
        {status, pCurrent} := opData → state;
    if pidOpData ≠ null then
        injectOperation( pidOpData ) ;                          // help inject the selected operation

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

Fig. 4. Pseudo-code for MINIMALCOPY (continued)

helping so as to ensure the wait-freedom property. Each table contains one entry for every process; the entry stores the address of the operation record of the most recent operation generated by the process.

3.4 Formal Description

A detailed pseudo-code of the algorithm is given in Figs. 3-6. The pseudo-code contains extensive comments and is self-explanatory. It uses the following functions: (i) read to dereference a pointer node and extract both its fields, (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. For convenience, we assume that the tree is never empty and always contains at least one node. This can be ensured by using a sentinel key that is

```
executeWindowTransaction( pNode, dNode )
begin
    // execute a window transaction for the operation stored in dNode
    opData := dNode → opData;
    {flag, dCurrent} := read( pNode ) ;            // read the contents of pNode again
    if dCurrent → opData = opData then
        if flag = OWNED then
            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
                CAS( opData → state, {WAITING, pRoot}, {IN_PROGRESS, pRoot} );

            if not (executeCheapWindowTransaction( pNode, dCurrent )) then
                // traverse the window and make copies as required
                windowSoFar := {clone( dCurrent )};
                while more nodes need to be added to windowSoFar do
                    pNextToAdd := the address of the pointer node of the next tree node to be copied;
                    dNextToAdd := pNextToAdd ⇝ dNode;
                    // help the operation located at this node, if any, move aside
                    if dNextToAdd → opData ≠ null then
                        executeWindowTransaction( pNextToAdd, dNextToAdd );
                    // read the address of the data node again as it may have changed
                    dNextToAdd := pNextToAdd ⇝ dNode;
                    copy pNextToAdd and dNextToAdd, and add them to windowSoFar;

                window has been copied; now apply transformations dictated by Tarjan' algorithm to
                windowSoFar;
                dWindowRoot := the address of the data node now acting as window root in
                               windowSoFar;
                if last/terminal window transaction then
                    status := COMPLETED;
                    pMoveTo :=
                        { the address of the record containing the value   : if update operation;
                        { null                                              : otherwise;
                else
                    status := IN_PROGRESS;
                    pMoveTo := the address of the pointer node of the node in windowSoFar to
                               which the operation will now move;
                    pMoveTo ⇝ flag := OWNED;
                    dMoveTo := pMoveTo ⇝ dNode;
                    dMoveTo → opData := opData;

                dWindowRoot → opData := opData;
                dWindowRoot → next := {status, pMoveTo};
                // replace the tree window with the local copy and release the ownership
                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
        dNow := pNode ⇝ dNode;
        if dNow → opData = opData then
            CAS( opData → state, {IN_PROGRESS, pNode}, dNow → next );
```

Fig. 5. Pseudo-code for MINIMALCOPY (continued)

larger than any other key value. For ease of exposition, we also assume that there is no reclamation of the memory allocated to nodes that have become garbage and are not "accessible" by any process. Thus all objects will 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* [21]. More details of the garbage collection operation can be found in [22].

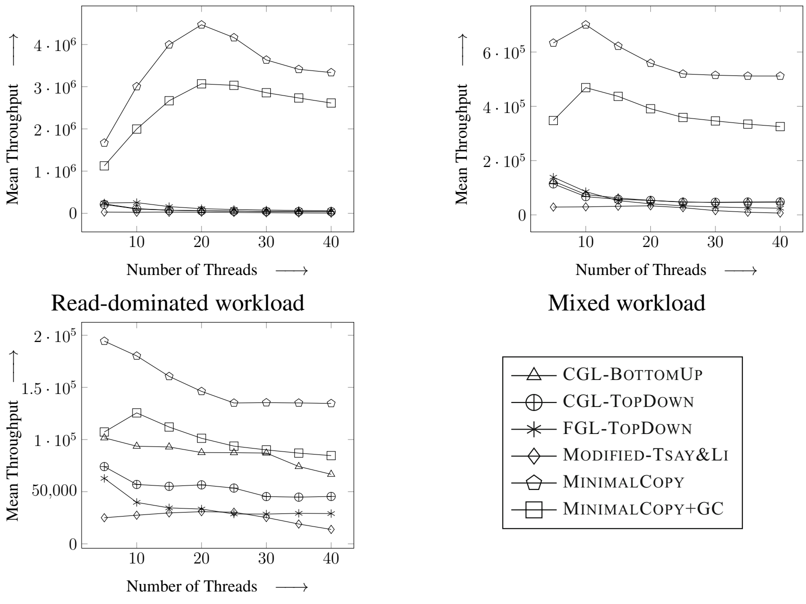
To prove the correctness of our algorithm, we show that all its execution histories are linearizable and all its operations are wait-free. For the linearizability proof, we define

```
Boolean executeCheapWindowTransaction( pNode, dNode )
begin
    opData := dNode → opDate;      pid := opData → pid;
    // traverse the tree window using Tarjan's algorithm
    while traversal not complete do
        pNextToVisit := the address of the pointer node of the next node to be visited;
        dNextToVisit := pNextToVisit ⇝ dNode;
        // abort if transaction already executed
        if (opData → state) ⇝ position ≠ pNode then return true
        // if there is an operation residing at the node, help it move out of the way
        if dNextToVisit → opData ≠ null then
            // there are several cases to consider
            if (dNextToVisit → opData) → pid ≠ pid then
                // the operation residing at the node belongs to a different process
                executeWindowTransaction( pNextToVisit, dNextToVisit );
                // read the address of the data node again as it may have changed
                dNextToVisit := pNextToVisit ⇝ dNode;
                // abort if transaction already executed
                if (opData → state) ⇝ position ≠ pNode then return true
            else if dNextToVisit → opData = dNode → opData then
                // partial window transaction has already been executed; complete it if needed
                if (opData → state) ⇝ position = pNode then
                    slideWindowDown( pNode, dNode, pNextToVisit, dNextToVisit );
                return true;
            else if MT[pid] ≠ opData then
                return true;                    // abort; transaction already executed
            visit dNextToVisit;

    if no transformation needs to be applied to the tree window then
        if last/terminal window transaction then
            pMoveTo := { the address of the record containing the value   : if an update operation;
                       { null                                             : otherwise;
            dMoveTo := null;
        else
            pMoveTo := the address of the pointer node of the node in the tree to which the operation
                       will now move;
            dMoveTo := pMoveTo ⇝ dNode;
        if (opData → state) ⇝ position = pNode then
            slideWindowDown( pNode, dNode, pMoveTo, dMoveTo );
        return true;
    else return false;

slideWindowDown( pMoveFrom, dMoveFrom, pMoveTo, dMoveTo )
begin
    opData = dMoveFrom → opData;
    // copy the data node of the current window location
    dCopyMoveFrom := clone( dMoveFrom );
    dCopyMoveFrom → opData := opData;
    if dMoveTo ≠ null then dCopyMoveFrom → next := {IN_PROGRESS, pMoveTo};
    else dCopyMoveFrom → next := {COMPLETED, pMoveTo};
    // copy the data node of the next window location, if needed
    if dMoveTo ≠ null then
        if dMoveTo → opData ≠ opData then
            dCopyMoveTo := clone( dMoveTo );      dCopyMoveTo → opData := opData;
            // acquire the ownership of the next window location
            CAS( pMoveTo, {FREE, dMoveTo}, {OWNED, dCopyMoveTo} );
    // release the ownership of the current window root and update the operation state
    CAS( pMoveFrom, {OWNED, dMoveFrom}, {FREE, dCopyMoveFrom} );
    CAS( opData → state, {IN_PROGRESS, pMoveFrom}, dCopyMoveFrom → next );
```

Fig. 6. Pseudo-code for MINIMALCOPY (continued)



Read-dominated workload

Mixed workload

Write-dominated workload

Legend.

Fig. 7. Comparison of throughput of different implementations of red-black tree

the linearization point of a "completed" operation as follows. For an insert or delete operation, the linearization point is taken to be the time when the operation performed its last window transaction. All update and search operations that act on the *same* record are linearized in the order given by Chuong *et al.*'s wait-free algorithm, and are ordered immediately after the insert operation that created that record. For a search operation that does not find the key, the linearization point is taken to be the time when the *last* terminal window transaction that is visible to the search operation is performed by some modify operation working on the same key. If no such modify operation exists, then the linearization point is taken to be the time when the operation began its traversal. We use these linearization points to construct an equivalent sequential history that respects the relative order of non-overlapping operation and in which all operations are legal. The wait-freedom of an operation follows from the helping performed by modify operations during searching, injection and execution of window transaction. More details of the correctness proof can be found in [22].

We refer to our wait-free algorithm for concurrent red-black tree as MinimalCopy and the version that supports garbage collection as MinimalCopy+GC.

4 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 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 [15]. For example, it is not clear how a search operation works. It appears that it cannot simply traverse the tree as in [14] because the tree is modified in-place using multiple CAS instructions and thus may be in an inconsistent state at times.

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 copying. Finally, in both lock-based top-down implementations CGL-TOPDOWN and FGL-TOPDOWN, search operations are used to speedup modify operations as appropriate.

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. 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 [7], rather than starting with an empty tree, we populated the tree to a certain size 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:* 0% search, 50% insert/update and 50% 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 Fig. 7. Each test was carried out for 60 seconds and the results were averaged over several runs to obtain values within 99% confidence interval. For MINIMALCOPY+GC, the garbage collection threshold was set to 25,000 nodes per thread. 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, for all the three categories of workloads, MINIMALCOPY and MINIMALCOPY+GC are the top two performers among all the implementations. Between the two, MINIMALCOPY+GC has 20%-45% lower throughput than MINIMALCOPY indicating that garbage collection has relatively significant overhead. The third best performer for read-dominated workloads is FGL-TopDown, whereas for mixed and write-dominated workloads is CGL-BottomUp. For read-dominated workloads, MinimalCopy+GC has 350%-4,300% better throughput than FGL-TopDown. For mixed workloads, MinimalCopy+GC has 150%-660% better throughput than CGL-BottomUp. For write-dominated workloads, the gap between MinimalCopy+GC and CGL-BottomUp is much smaller; MinimalCopy+GC has only 3.4%-34% better throughput than CGL-BottomUp. More details about the experiments (*e.g.*, comparison of various implementations with respect to execution times of search and modify operations) can found in [22].

5 Conclusion

In this paper, we have presented an new wait-free algorithm for a concurrent red-black tree. Our experiments indicate that our algorithm has significantly better performance than other concurrent algorithms for a red-black tree including those based on locks.

References

1. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming, Revised Reprint. Morgan Kaufmann (2012)
2. Herlihy, M.: Wait-Free Synchronization. ACM Transactions on Programming Languages and Systems (TOPLAS) 13(1), 124–149 (1991)
3. Bender, M.A., Fineman, J.T., Gilbert, S., Kuszmaul, B.C.: Concurrent Cache-Oblivious B-Trees. In: Proceedings of the 17th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), pp. 228–237 (2005)
4. Ellen, F., Fataourou, P., Ruppert, E., van Breugel, F.: Non-Blocking Binary Search Trees. In: Proceedings of the 29th ACM Symposium on Principles of Distributed Computing (PODC), pp. 131–140 (2010)
5. Brown, T., Helga, J.: Non-Blocking *k*-ary Search Trees. In: Fernàndez Anta, A., Lipari, G., Roy, M. (eds.) OPODIS 2011. LNCS, vol. 7109, pp. 207–221. Springer, Heidelberg (2011)
6. Prokopec, A., Bronson, N.G., Bagwell, P., Odersky, M.: Concurrent Tries with Efficient Non-Blocking Snapshots. In: Proceedings of the 17th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP), pp. 151–160 (2012)
7. Howley, S.V., Jones, J.: A Non-Blocking Internal Binary Search Tree. In: Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), pp. 161–171 (June 2012)
8. Braginsky, A., Petrank, E.: A Lock-Free B+tree. In: Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), pp. 58–67 (2012)
9. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introduction to Algorithms. The MIT Press (1991)
10. Sedgewick, R.: Left-leaning Red-Black Trees
11. Jones, M.T.: Inside the Linux 2.6 Completely Fair Scheduler (December 2009)
12. Chuong, P., Ellen, F., Ramachandran, V.: A universal construction for wait-free transaction friendly data structures. In: Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), pp. 335–344 (2010)

13. Fatourou, P., Kallimanis, N.D.: A Highly-Efficient Wait-Free Universal Construction. In: Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), pp. 325–334 (2011)
14. Ma, J.: Lock-Free Insertions on Red-Black Trees. Master's thesis. The University of Manitoba, Canada (October 2003)
15. Kim, J.H., Cameron, H., Graham, P.: Lock-Free Red-Black Trees Using CAS. Concurrency and Computation: Practice and Experience, 1–40 (2006)
16. Fraser, K.: Practical Lock-Freedom. PhD thesis, University of Cambridge (February 2004)
17. Crain, T., Gramoli, V., Raynal, M.: A Speculation-Friendly Binary Search Tree. In: Proceedings of the 17th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP), pp. 161–170 (2012)
18. Tsay, J.J., Li, H.C.: Lock-Free Concurrent Tree Structures for Multiprocessor Systems. In: Proceedings of the International Conference on Parallel and Distributed Systems (ICPADS), pp. 544–549 (December 1994)
19. Tarjan, R.E.: Efficient Top-Down Updating of Red-Black Trees. Technical Report TR-006-85, Department of Computer Science, Princeton University (1985)
20. Herlihy, M., Wing, J.M.: Linearizability: A Correctness Condition for Concurrent Objects. ACM Transactions on Programming Languages and Systems (TOPLAS) 12(3), 463–492 (1990)
21. Michael, M.M.: Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. IEEE Transactions on Parallel and Distributed Systems (TPDS) 15(6), 491–504 (2004)
22. Natarajan, A., Savoie, L., Mittal, N.: Concurrent Wait-Free Red Black Trees. Technical Report UTDCS-16-12, Department of Computer Science, The University of Texas at Dallas (October 2012)