Lock-Free Red-Black Trees Using CAS

Jong Ho Kim Helen Cameron Peter Graham

October 20, 2011

Abstract

The negative side-effects of using lock-based synchronization mechanisms are well known and include unexpected scheduling anomalies such as priority inversion and convoying as well as unnecessary synchronization overhead and the potential for deadlock. Despite these drawbacks, the use of lock-free techniques has not gained widespread acceptance. The reasons for this lack of acceptance include the complexity of hand-crafting lock-free algorithms for complex data structures and the relative inefficiency of the algorithms resulting from the use of generic methods for converting sequential algorithms into lock-free concurrent ones. Given these factors, we believe that the best approach is to create a library of efficient lock-free data structures that can be easily employed by others. For such a library to be successful, however, it must be based on synchronization primitives that are generally available on a wide range of machine architectures.

In this paper, we describe the implementation of efficient lock-free algorithms for operating on a type of balanced binary search tree — red-black trees. Such tree structures are of potential importance in a wide range of search-based applications (including text mining and nucleotide sequence matching) that are being implemented on shared memory parallel machines. Our algorithms require only the widely available CAS (Compare And Swap) synchronization primitive.

1 Introduction

A red-black tree is a useful binary search tree (BST) structure with some special properties that make it an efficient structure to use in search applications. A red-black tree has height that is logarithmic in the number of nodes, making it a better choice than an ordinary binary search tree, which can have height proportional to the number of nodes. With efficient search, insert and delete algorithms that make a constant number of changes to the structure of the tree, red-black trees make a useful structure in search-intensive applications. Because they are efficient and require few structural changes in a small neighborhood for updates, red-black trees ought to make an efficient data structure for concurrent programming.

Increasingly, large-scale in-memory data structures are being employed for a range of new applications, many requiring efficient search. To provide high performance, to support multiple concurrent uses and in some cases to gain access to sufficiently large physical memory capacity, shared memory parallel machines (SMPs) are being used. In a shared memory machine, data structures are accessible to multiple concurrent processes running on different processors. To ensure correctness, access to the data structures must be synchronized. Such synchronization may be accomplished pessimistically (assuming conflicting operation will occur and always taking steps to prevent them) or optimistically (assuming few conflicts will happen and taking steps to correct them when they do).

Pessimistic techniques are by far the most common, being ubiquitous and simple to use and analyze. Most pessimistic techniques are based on the use of locks, which prevent concurrent access. A lock is acquired on some data of interest, the data is updated and then the lock is released. Other processes attempting to acquire a held lock are blocked until the lock is released. Unfortunately, the use of locks has several negative consequences, including significant fixed overhead even when contention is unlikely, possibly unnecessary limitations on concurrency and undesirable scheduling side-effects.

Optimistic techniques can be used to address the problems associated with lock-based approaches. Rather than acquiring a lock, a process makes a copy of the data it wishes to modify, changes the copy and then replaces the original data with the copy only if the original data is unchanged (i.e., no other concurrent process has changed the data). If there is no contention, the update is made with no appreciable overhead. If there is contention, all but one concurrent process accessing the shared data must "roll-back" and redo its computation using the updated data. Optimistic techniques can be divided into "lock-free" and, the stronger, "wait-free" techniques. Our focus is on lock-free techniques.

Concurrent programming, in general, is a difficult task. Experience has shown that when a programmer can make use of a library of efficient concurrent routines and/or data structures (e.g., BLAS, ScaLAPACK, the NAG libraries, etc.), the burden of concurrent programming is significantly decreased. Unfortunately, efficient *lock-free* concurrent versions of many useful data structures, including red-black trees, are generally not available to programmers.

In this paper, we describe lock-free algorithms to perform concurrent operations on a red-black tree. Further, we constrain ourselves to the direct use of synchronization primitives that have been implemented in real machines. This constraint makes our algorithm implementation significantly more challenging, but also offers the potential for better performance [1, 2]. Another challenge of implementing lock-free algorithms for red-black trees is the need to make changes in multiple parts of the tree (recolouring nodes up the insertion or deletion path, followed by a constant number of rotations in a small neighbourhood of nodes). Though specific to red-black trees, our algorithms provide useful techniques that may allow the creation of similar concurrent lock-free algorithms for other complex, multi-link data structures.

The rest of this paper is organized as follows. Section 2 provides background on lock-free techniques and briefly reviews work related to that presented in this paper. Our general approach to supporting lock-free operations on complex data structures is explained in Section 3. Section 4 describes our CAS-based lock-free algorithms for red-black trees (focusing on the most challenging operation – deletion). Finally, Section 5 concludes the paper and discusses some directions for future work.

2 Background and Related Work

The most common form of synchronization provided between concurrent processes[1] in shared memory multiprocessors is based on the use of locks. While simple to use, ubiquitous, and easily amenable to performance analysis, the use of locks also has negative side-effects. These include the potential for deadlock, unavoidable overhead at each synchronization point (typically a system call) and possible scheduling anomalies such as priority inversion (where high-priority processes are delayed waiting for a lock held by a low-priority process) and convoying (where a delayed process holding a lock effectively blocks all other processes waiting for the lock).

2.1 Synchronization Primitives

Most modern computer architectures provide primitives (using one or more machine instructions) for both lock-based (pessimistic) and lock-free (optimistic) synchronization. The most common of these are Test-And-Set (TAS) for lock-based synchronization and Compare-And-Swap (CAS) for lock-free synchronization, though a number of variants also exist.

Most research on lock-free synchronization using real machine primitives has focused on CAS, as does this paper. Pseudocode for the *atomic* CAS instruction is shown in Figure 1(a). A shared variable (`shVble`) is compared to an earlier saved copy (`savedValue`) and, if they are the same, the shared variable is updated (with `newValue`) and `True` is returned. Otherwise, `False` is returned. Figure 1(b) shows how CAS is used to effect the lock-free modification of shared data. The updating process makes a copy (`savedValue`) of the shared variable (`shVble`) and computes a new value (`newValue`) using it. CAS is then used to attempt to update the shared variable. If the update fails, the process is repeated (via the `do-while` loop). Note that there is no appreciable overhead in using CAS when concurrent processes are accessing different data. When

[1]We refer to processes in the paper, but threads or other units of concurrency are equally applicable.

```
CAS(shVble,savedValue,newValue)
 Begin Atomic
 if (shVble==savedValue) {
    shVble=newValue;
    return True;
 } else {
    return False;
 }
 End Atomic
```

(a)

```
do {
    savedValue=shVble;
    newValue=computeNewValue(savedValue);
}while (!CAS(shVble,savedValue,newValue));
```

(b)

Figure 1: The specification (a) and use (b) of CAS.

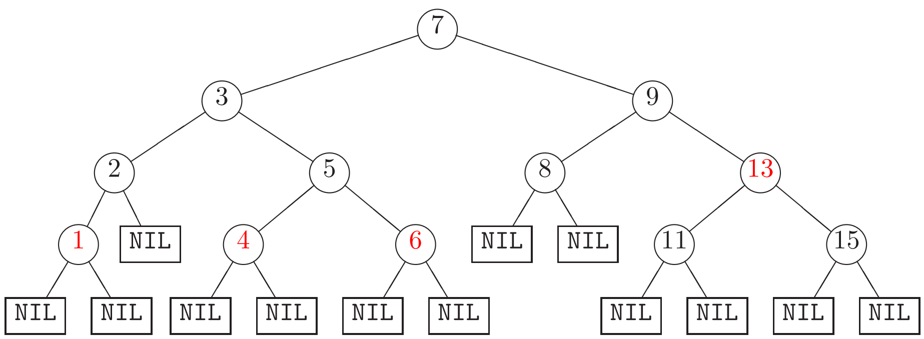


Figure 2: An example red-black tree.

two or more processes attempt to concurrently update a single shared datum, however, only one process succeeds. The CAS operations of all other concurrent processes fail and those processes must "rollback" and recompute their modifications to the shared data (using the new value) before retrying their CAS operations.

2.2 Sequential Red-Black Trees

A red-black tree is a binary search tree (BST) with five associated red-black properties:

1. Every node is either red or black.
2. The root node is black.
3. External nodes are black.
4. A red node's children are both black.
5. All paths from a node to its leaf descendants contain the same number of black nodes.

An ordinary BST with n internal nodes can have height as large as n, so that searches, insertions and deletions in an ordinary BST can take $O(n)$ time. The red-black properties ensure that a red-black tree has height at most $O(\log n)$, so red-black tree search and update operations cost at most $O(\log n)$ time.

An example of a red-black tree is shown in Figure 2, which exhibits both the BST and red-black properties. First, each node is either red or black. Second, the root, whose key is 7, is black. Third, the external nodes (the NIL nodes) are all black. Fourth, the red nodes 1, 4, 6, and 13 each have two black children. Fifth, the paths from each node to its leaf descendants (external node descendants) have the same number of black

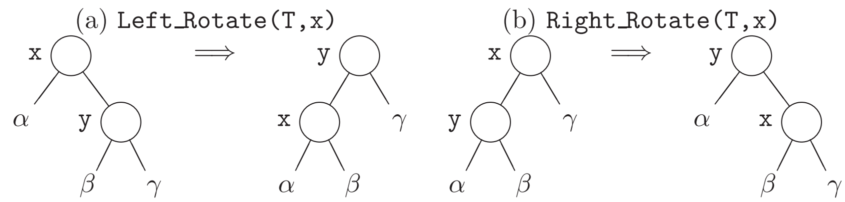


Figure 3: Rotation in a red-black tree.

```
Left_Rotate(T,x) {
  y = right[x];
  right[x] = left[y];
  p[left[y]] = x;
  p[y] = p[x];
  if (p[x] == p[root]) root[T] = y;
  else if (x == left[p[x]])left[p[x]] = y;
  else right[p[x]] = y;
  left[y] = x;
  p[x] = y;
}
```

Figure 4: Pseudocode for a left rotation.

nodes. For example, the path from the root down to NIL including 7, 9, 8, NIL has 4 black nodes, and so does the path 7, 3, 5, 4, NIL, etc.

There are three operations defined on red-black trees: search, insert and delete. All these operations must preserve the red-black properties. Since searching does not modify the tree, the red-black properties are, naturally, preserved. However, both inserts and deletes may cause changes to the tree's structure, which, if not carefully managed, will violate the red-black properties. Rotation operations are used to help restore balance and the red-black properties after insertions and deletions.

We now briefly review sequential rotations, insertions and deletions in red-black trees. We use the algorithms from Cormen et al. [3] for this purpose and later modify them to be lock-free in Section 4. Cormen et al. use a single sentinel node `nil[T]` to represent all the NIL external nodes in a red-black tree as well as the root's parent. The colour of the sentinel node is black; its other fields contain arbitrary values.

2.2.1 Rotations in Red-Black Trees

Rotation operations preserve the BST property while restructuring red-black trees to ensure balance. There are two types of rotations: left and right (shown in Figure 3, where `T` refers to the red-black tree and `x` is the root of the subtree to be rotated). In a left rotation, node `x`'s right child, `y`, becomes the new parent of `x`. `x`'s previous parent becomes `y`'s parent, and `y`'s previous left child becomes `x`'s new right child. Figure 4 shows pseudocode for `Left_Rotate(T,x)`. `Right_Rotate` is symmetric to `Left_Rotate(T,x)`.

2.2.2 Sequential Insertion in Red-Black Trees

Insertion in a red-black tree occurs in two stages: the insertion of the new node followed by "fixing up" the red-black properties. The `RB_Insert` routine (Figure 5) simply walks down the tree from the root, selecting the left or right link depending on the key of the node to be inserted, until it finds the internal node parent under which the new node should be inserted. Once the new node has been linked into the tree, the routine

```
RB_Insert(T,x) {
  z = p[root]; // dummy root parent assumed
  y = root[T];
  while(y!=nil[T]) { // Find insert point z
    z = y;
    if (key[x] < key[y]) y = left[y];
    else y = right[y]
  } /* end while */
  /* Place new node x as child of z */
  p[x] = z;
  if (z == p[root]) { root[T] = x; left[p[root]]=x; }
  else if (key[x] < key[z]) left[z] = x;
  else right[z] = x;
  left[x] = nil[T];
  right[x] = nil[T];
  colour[x] = red;
  RB_Insert_Fixup(T,x);
}
```

Figure 5: Pseudocode for insertion.

`RB_Insert_Fixup` (Figure 6) restores any red-black properties that may have been violated by the insertion. `RB_Insert_Fixup` restores the red-black properties by re-colouring nodes and performing zero, one or two rotations. Beginning at the insertion point, the process proceeds upwards towards the root of the tree, re-colouring nodes as long as two consecutive red nodes exist (violating red-black rule 4). Following this re-colouring, the routine may perform one or two rotations to re-balance the tree. Collectively, the re-colouring and rotation(s) ensure that all five red-black properties are re-established.

2.2.3 Sequential Deletion from Red-Black Trees

Deletion in a red-black tree also occurs in two stages: the deletion of the target node, followed by "fixing up" the red-black properties. Deletion, however, is somewhat more complex than insertion because the deletion of a node may require additional restructuring of the tree. The `RB_Delete` routine (Figure 7) identifies the node to be deleted and unlinks it from the tree (possibly shuffling key values in the process). If the colour of the removed node was black (causing a violation of property 5), then the `RB_Delete_Fixup` routine (Figure 8) is invoked to restore the red-black properties. `RB_Delete_Fixup`, like `RB_Insert_Fixup`, walks towards the root of the tree performing re-colouring and rotate operations as required.

2.3 Review of Lock-Free Techniques and Algorithms

We divide this review into discussions of universal primitives, lock-free algorithms for specific data structures, Software Transactional Memory (STM), and Ma's algorithm [4] (upon which we build).

2.3.1 Universal Primitives

A synchronization primitive is said to be universal if it can be used to construct any wait-free or lock-free data structure.[2] Herlihy [5, 6] proves that certain universal primitives do exist. To show the universality of a synchronization primitive, Herlihy [6] uses the concept of consensus number (the maximum number of processes that can solve a consensus problem using a certain synchronization primitive). Herlihy specifies a

[2] Wait-free techniques guarantee that every process will complete its operation in a finite number of steps, while lock-free techniques only guarantee that some process will complete its operation in a finite number of steps.

```
RB_Insert_Fixup(T,x) {
  while (colour[p[x]]==red) {
    if (p[x]==left[p[p[x]]]) {
      y = right[p[p[x]]];
      if (colour[y]==red) { // Case 1
        colour[p[x]] = black;
        colour[y] = black;
        colour[p[p[x]]] = red;
        x = p[p[x]];
      } else {
        if (x==right[p[x]]) { // Case 2
          x = p[x];
          Left_Rotate(T,x);
        } /* end if */
        colour[p[x]] = black; // Case 3
        colour[p[p[x]]] = red;
        Right_Rotate(T,p[p[x]]);
      } /* end else */
    } else { // p[x] = right[p[p[x]]]
      /* Symmetric to above. */
    } /* end if */
  } /* end while */
  colour[root[T]] = black;
}
```

Figure 6: Pseudocode for insertion fixup.

```
RB_Delete(T,z) {
  if (left[z]==nil[T] || right[z]==nil[T])
    y = z;
  else
    y = SUCCESSOR(z); // key-order successor
  if (left[y]!=nil[T]) x = left[y];
  else x = right[y];
  p[x] = p[y];
  if (p[y]==p[root])
    root[T] = x;
  else {
    if (y==left[p[y]]) left[p[y]] = x;
    else right[p[y]] = x;
  } // end else
  if (y!=z)
    key[z] = key[y];
  if (colour[y]==black)
    RB_Delete_Fixup(T,x);
  return y;
}
```

Figure 7: Pseudocode for deletion.

```
RB_Delete_Fixup(T,x) {
  while(x!=root[T] && colour[x]==black) {
    if (x==left[p[x]]) {
      w = right[p[x]];
      if (colour[w]==red) { // Case 1
        colour[w] = black;
        colour[p[x]] = red;
        Left_Rotate(T,p[x]);
        w = right[p[x]];
      }
      if (colour[left[w]]==black &&
          colour[right[w]]==black) { // Case 2
        colour[w] = red;
        x = p[x];
      } else {
        if (colour[right[w]]==black) { // Case 3
          colour[left[w]] = black;
          colour[w] = red;
          Right_Rotate(T,w);
          w = right[p[x]];
        }  // end if
        colour[w] = colour[p[x]]; // Case 4
        colour[p[x]] = black;
        colour[right[w]] = black;
        Left_Rotate(T,p[x]);
        x = root[T];
      } /* end else */
    } else { // p[x] = right[p[p[x]]]
      /* Symmetric to above */
    } /* end if */
  } /* end while */
  colour[x] = black;
}
```

Figure 8: Pseudocode for deletion fixup.

necessary and sufficient condition for the universality of a synchronization primitive as follows: "An object is universal in a system of n processes iff it has a consensus number greater than or equal to n." This condition implies that if a synchronization primitive's consensus number is infinity, it can solve a consensus problem in a system for an unbounded number of processes. A primitive with infinite consensus number is therefore universal. Herlihy shows that there are several synchronization primitives with infinite consensus number, namely, memory-to-memory move and swap, augmented queue, CAS, and fetch-and-cons. Plotkin [7] has also shown the universality of another primitive, known as "Sticky Bits". While any universal primitive can be used to construct wait-free and lock-free data structures, due to hardware constraints, only CAS and its extension DCAS and a variant, LL/SC (Load-Linked and Store-Conditional), have been implemented.

2.3.2 Lock-Free Algorithms for Specific Data Structures

To find a general method (also called a universal method) to implement a lock-free data structure, Herlihy [8] devised the first technique to convert any sequential data structure into a wait-free concurrent data structure. Other general methods (e.g., [9, 10, 11, 12]) followed. General methods, however, are commonly accepted as not being efficient when compared to lock-based algorithms. For instance, LaMarca [13] shows that the performance of Herlihy's and Barnes' general methods are worse than a spin-lock-based solution.

In contrast to general methods, some lock-free data structures implemented with data-structure specific algorithms have shown better performance than corresponding lock-based algorithms. Michael and Scott [1] performed a series of tests on several lock-free data structures and their results show that the lock-free data structures perform better than their lock-based counterparts. The lock-free data structures considered include link-based queues by Michael and Scott [14] and Prakash et al. [15], a link-based stack by Treiber [16], and optimized versions of a stack and a skew heap resulting from applying Herlihy's method. Michael and Scott also considered an LL/SC implementation of counters, as well as quicksort and traveling salesman applications using some of the tested lock-free data structures. All the results showed better performance with the lock-free data structures than with their lock-based counterparts.

Included among the many lock-free algorithms for specific data structures are some that address dictionary[3] structures. These are of particular interest since such structures provide the same basic functionality as red-black trees and thus are possible alternatives to their use. Among the more recent work on such structures are [17, 18, 19]. We chose to consider red-black trees since they are a good representative of a larger class of complex, multi-link data structures for which we are generally interested in trying to develop lock-free algorithms.

2.3.3 Software Transactional Memory

For complex, multi-link data structures (such as red-black trees), the construction of efficient lock-free algorithms using simple primitives is challenging. For example, to complete a single logical operation (e.g., deletion of a node), such an algorithm may have to make changes to many different parts of the data structure. To simplify the development of such algorithms, some researchers have chosen to develop Software Transactional Memories (STMs), where a sequence of operations performed in the memory can be treated as a transaction and be either committed or aborted as needed.

Both Herlihy et al. [21] and Harris and Fraser [22] have implemented concurrent operations for red-black trees using STMs. While the implementation details of Herlihy et al.'s STM and Harris and Fraser's STM are different, both provide similar sets of application programming interfaces for transactional operations (e.g., to start, commit, and abort transactions). We choose to describe Harris and Fraser's implementation, which builds on their earlier work with software multi-word CAS (MCAS) [20].

The idea behind Harris and Fraser's implementation is that each process involved in updating the tree will optimistically perform all of the operations as part of one transaction. The nodes that are modified will store their old data, new data, and version numbers in the STM. After a process completes its work, the operations recorded in the STM will either be committed or aborted. They will be aborted if the version number of any of the recorded nodes has been changed by another process.

[3]Dictionary structures support keyed insertion and retrieval of data as well as the deletion of data with specific keys.

The STM approach simplifies the implementation of concurrent red-black trees. However, at commit time, all the nodes that have been recorded for updates during a transaction have to have their versions checked using a potentially large series of CAS operations. The nodes involved may include those located anywhere from a leaf to the root and may cover a large area of the tree. Hence, the likelihood of re-execution due to contention is high. Also, the cost of re-execution will also be high since all the tentative updates to a transaction are lost when the transaction is aborted. Finally, there is the potential for "cascade" effects where abortion of one process may indirectly result in unnecessary aborts of others (who accessed data touched by the aborted process).

The algorithm we describe in this paper reduces the working area of a process in the tree to just a few nearby nodes. Therefore, more processes may run concurrently in a red-black tree with less re-execution cost due to contention. The price paid is that processes must sometimes wait for other processes to vacate their working area (in a fashion similar to spin locks). The cost of such waiting is expected to be significantly less than the cost of transaction aborts.

2.3.4 Ma's Insertion Algorithm

Ma [4] presents a lock-free insertion algorithm for red-black trees. Ma's algorithm is based on Cormen et al.'s sequential insertion algorithm [3] with the addition of a "local area" concept and the use of lock-free primitives to control concurrency. We now describe Ma's local area concept, which we adapt and extend in this paper.

Ma adds an extra `flag` field to each node. When the flag of a node is set by a process, that process gains control of the node. The local area consists of the set of nodes that a process must have full control of to ensure successful completion of an operation (in Ma's case, an insertion). For insertion, the local area consists of the current node `x`, `x`'s parent, `x`'s grandparent, and `x`'s uncle (refer to Figure 9 (a)). With the local area concept, if several processes in a localized region have overlapping areas, only the one process that obtains all the flags in its local area will be able to proceed. Other processes will have to re-attempt to gain control of their local areas (effectively waiting for the successful process, but without blocking).

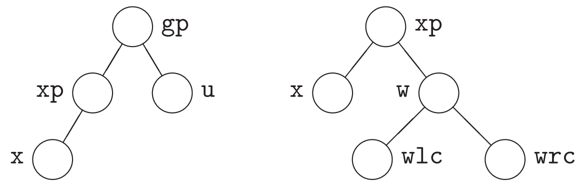
As one insertion process completes its processing in one part of the tree, it may either finish entirely (and release all its flags) or move up the tree. In the latter case, the process first obtains the flags of nodes in its new local area, moves up, and then releases the flags that it set for the nodes in its old local area. Releasing flags allows other processes that may have been waiting to advance.

Ma's algorithm is entirely based on CAS-type primitives. Unfortunately, as originally presented, her approach requires the use of CAS, DCAS and TCAS (Triple CAS — with three sets of arguments). While CAS support is commonly available and DCAS has been implemented in some machine architectures, TCAS is unavailable.

We have since re-examined Ma's algorithm and realized that a child node can only be moved away from its parent by a rotation. In Ma's original algorithm, a process performing a rotation that separates a node q from one of its children must have control (i.e., hold the flag) of q. Thus, if a process has set the flag of a node q, no other process can move one of the children of q away from q. Ma uses TCAS only during rotations to simultaneously change pointers between a parent-child node pair, while ensuring the child has not been moved from its parent by another process. Because the parent's flag has been set by the process performing the rotation, our above arguments guarantee that the child will not be moved away from its parent by any other process. Thus, we can safely change the child's parent pointer and then the parent's child pointer without using any CAS, since no other process will be trying to change those pointers. Therefore Ma's use of TCAS is not required.

We also examined Ma's use of DCAS and concluded that it was also unnecessary. Ma uses DCAS when she has previously set the flag of a node q and wants to set the flag of node r, where r is either the parent or child of q. The DCAS simultaneously sets the flag of r and checks that r remains the parent or child of q. We can, instead, set the flag of r using CAS. After this is done, since we now have control of both q and r, we can check whether q and r remain parent and child (or child and parent) and retry if necessary.

Applying these two optimizations, we have designed Ma's insertion algorithm for red-black trees so that it requires only CAS. Further, this algorithm is compatible with the lock-free deletion approach presented



(a) Insertion Local Area (b) Deletion Local Area

Figure 9: Local areas for insertion and deletion.

later in this paper.

3 Using CAS for Complex Data Structures

CAS is relatively simple to apply in creating lock-free operations for singly-linked data structures (e.g., linked lists, queues, etc.). CAS can be easily used to effect local changes to such data structures by "swinging" a single pointer. Unfortunately, operations on more complicated data structures can be problematic due to the need to make complex changes to the structures in an atomic fashion. In general, complex changes could require n-CAS type primitives, where n could be relatively large. Such synchronization primitives do not currently exist and are unlikely to be implemented in hardware due to the need to concurrently manipulate many memory locations (which is impractical given current memory systems design).

Rather than attempt to use CAS (or variants) to directly manipulate the underlying data structure, instead we use CAS to ensure that concurrent processes always maintain a safe distance from one another. This distancing allows the processes to operate on distinct parts of the data structure concurrently without fear of interference. By carefully designing the lock-free operations, efficient concurrent updates may be supported on a wide range of surprisingly complex data structures, including red-black trees. Further, the likelihood of parallel processes having to wait for significant periods of time is low. In general, the expected wait time is proportional to the level of contention, which is proportional to the number of concurrent processes. For large data structures with many search operations and fewer updates (a common characteristic of many practical problems), contention will be low.

In addition to applying the local area and flag concept introduced by Ma, we add a series of "intention markers", which are set by processes and used to keep processes away from regions where another process may have an unexpected effect (i.e., where a process may have some intention of action). By enforcing a safe distance between processes, we can perform safe concurrent operations using only CAS. In essence, we trade a small amount of potential concurrency (by preventing processes from operating closer to one another) for safety. Using our scheme, there are, of course, none of the negative scheduling side-effects associated with competitive solutions that use fine-granularity (node level) locking.

While the algorithms presented in this paper are specific to red-black trees, the concepts illustrated for red-black trees are generally applicable to other complex, multi-link data structures. We ultimately envision the creation of a library of lock-free algorithms for important data structures using this technique. Such a library will allow simple and efficient parallel program development for shared memory machines, where the programmer is isolated from the heroic programming efforts needed to develop lock-free structures using hardware supported synchronization primitives.

4 Lock-Free Algorithms for Red-Black Trees Using CAS

We selected red-black trees as our first implementation target for four reasons. First, they are a complex, multi-link data structure. Second, we have personal experience with them. Third, they are directly applicable to a wide-range of online searching algorithms. Finally, we chose red-black trees rather than other balanced

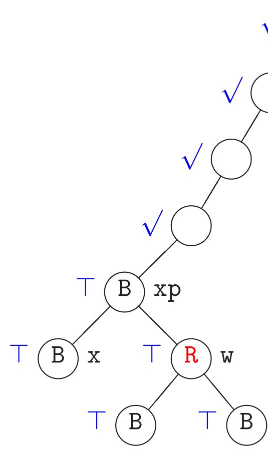


Figure 10: Intention markers for a deletion process.

BSTs (e.g., AVL trees) because operations on red-black trees have more localized effects than those of other balanced BSTs. (This localization yields a simpler problem, though one which is still challenging and relevant.)

In the description of various aspects of our algorithms, we will refer to specific "cases" occurring in either the insertion or deletion fixup code as described in [3]. For reference, these cases are identified in blue in the comments in Figures 6 and 8, respectively. We begin by describing the key concepts, design choices and requirements discovered in the creation of our algorithms. In particular, we focus on the need for special rules to construct a correct lock-free deletion algorithm. Such rules, while specific to red-black trees, provide useful lessons learned about using the technique that may also be applicable to the design of lock-free algorithms for other complex data structures. Before concluding the paper we also present and discuss our parallel lock-free versions of the algorithms from [3].

4.1 The Local Areas for Insertion and Deletion

As defined by Ma [4], the local area is a set of nearby nodes that a process may access and modify while performing an operation on a red-black tree and over which it must therefore have full control. The local area is realized by the addition of a flag to each node. When a process successfully sets a flag (using CAS), it gains full control of the associated node.

Figure 9 (a) shows the nodes that make up an insertion process's local area. The node `x` first points to the newly inserted node and then to its ancestor nodes as the process (performing insertion Case 1) moves up towards the root. (Also, `xp` is `x`'s parent, `gp` is `x`'s grandparent, and `u` is `x`'s uncle.) The local area for a deletion, shown in Figure 9 (b), is different. For deletion, `x` initially points to the node that replaces the deleted node and then (via deletion Case 2) to ancestor nodes as the process moves up the tree. (Also, `w` is the sibling of `x`, `wlc` is `w`'s left child, and `wrc` is `w`'s right child.)

4.2 Intention Markers

4.2.1 Definition and Example

Intention markers ensure a safe distance between any two processes so that each process is guaranteed to be able to act. An intention marker is an integer field in each node that is set by a process to indicate that the process intends to move its local area up to that node. To set a marker, a process places its process ID in the marker field of the node. Figure 10 shows the required intention markers for a deletion process. The √

marks represent the intention markers[4]. A ⊤ mark next to a node indicates that the process also has the flag of that node. A process can place its intention marker on a node only if the node has no other process's marker or flag, but a process can set a flag on a node even if another process's marker is on the node.

Consider Figure 11, which shows a sequence of cases that a deletion process $\mathtt{P}_1$ might go through in the presence of a nearby process $\mathtt{P}_2$.[5] In Figure 11 (a), $\mathtt{P}_1$ (shown in blue) is in deletion Case 2, $\mathtt{P}_2$ (shown in black) is also in deletion Case 2 ($\mathtt{P}_2$'s local area is located below its intention markers and is not shown in the picture). Each of the two processes will colour its `w` red and move its `x` up one level. To move `x` up, a process must first place its new highest intention marker at the top of the four intention markers. In Figure 11 (a), $\mathtt{P}_1$ can set the additional marker at the top because no other process's marker or flag is on the node, but $\mathtt{P}_2$ cannot because node `a` already has $\mathtt{P}_1$'s marker.

In Figure 11 (b), $\mathtt{P}_1$ has performed deletion Case 2. In particular, $\mathtt{P}_1$ has set the new intention marker at the top, moved `x` up, obtained the flags of nodes in its new local area, and released the flags of nodes in its old local area. $\mathtt{P}_1$ could place its flags on nodes `c` and `f` even though $\mathtt{P}_2$'s markers are on these nodes (flags are less constrained than intention markers). Having its new local area, $\mathtt{P}_1$ can determine that it is now in deletion Case 1.

Figure 11 (d) shows the result after P_1 performs deletion Case 1 and is now in deletion Case 3. P_1 has acquired the flags of nodes `j` and `k` for its new local area. The second highest flag, on node `c`, is kept for the rotation on P_1`.xp` (in deletion Case 4) that follows deletion Case 3. Notice that P_2 has been "pulled" one level up inside P_1's local area. In Figure 11 (b), P_2 highest marker is on P_1`.w`, and, in Figure 11 (d), P_2's highest marker is now on P_1`.xp`. The rotations of deletion Case 3 and Case 4 that follow Case 1 each pull P_2 one level up. This unexpected side-effect of rotations significantly complicates our basic technique.

Figure 11 (e) shows the result after $\mathtt{P}_1$ performs deletion Case 3 and Figure 11 (f) shows the result after $\mathtt{P}_1$ performs deletion Case 4. When P_1 exits, it normally releases the flags and intention markers that it holds. P_1 does not release them if it needs to apply the Move-Up rule. This rule is discussed in Section 4.3.

4.2.2 The Necessity of Intention Markers

In this section we discuss the necessity of intention markers as illustrated in Figure 12. In the figure, $\mathtt{P}_1$ (shown in blue) and $\mathtt{P}_2$ (shown in black) are both in deletion Case 2, and neither process has any intention markers. (The nodes shown with square boxes represent external leaf nodes that are coloured black and contain arbitrary values in their other fields. These external nodes have been omitted in other parts of the figure and from other figures in this paper for simplicity.) $\mathtt{P}_1$ and $\mathtt{P}_2$ have each coloured their `w` red and are ready to move up. However, before `x` is moved up, the process's new local area must be obtained. The node that will be the new $\mathtt{P}_1$`.w` when $\mathtt{P}_1$`.x` is moved up is already held by $\mathtt{P}_2$. Also, the node that will be the new $\mathtt{P}_2$`.w` when $\mathtt{P}_2$`.x` is moved up is already held by $\mathtt{P}_1$. Hence, neither process can proceed.

Intention markers ensure sufficient distance between processes so that this problem does not occur. For instance, if $\mathtt{P}_1$ had an intention marker on $\mathtt{P}_1$`.x`'s grandparent in Figure 12, then $\mathtt{P}_2$ would not have been able to put its marker there and hence would not have been able to be where it is in the picture (and thus cause the problem).

However, one intention marker is not sufficient to provide a safe distance. Four intention markers with two rules that we present next are both necessary and sufficient to provide the required distance between processes. (The reader is referred to Kim [23] for discussion of these and other rules created for an optimized version of our deletion algorithm.)

4.3 Overview of the Algorithm

Our algorithm is based on the sequential red-black tree algorithms in Cormen et al. [3] and the local-area-based lock-free red-black tree insertion algorithm by Ma [4]. Specifically, we have incorporated the concept of intention markers into Ma's insertion algorithm and developed a new, compatible deletion algorithm.

[4]For convenience, markers are shown in the colour of their associated process rather than using a process ID.
[5]Note that higher level "marked" nodes which are not involved in the rotations are omitted from Figure 11.

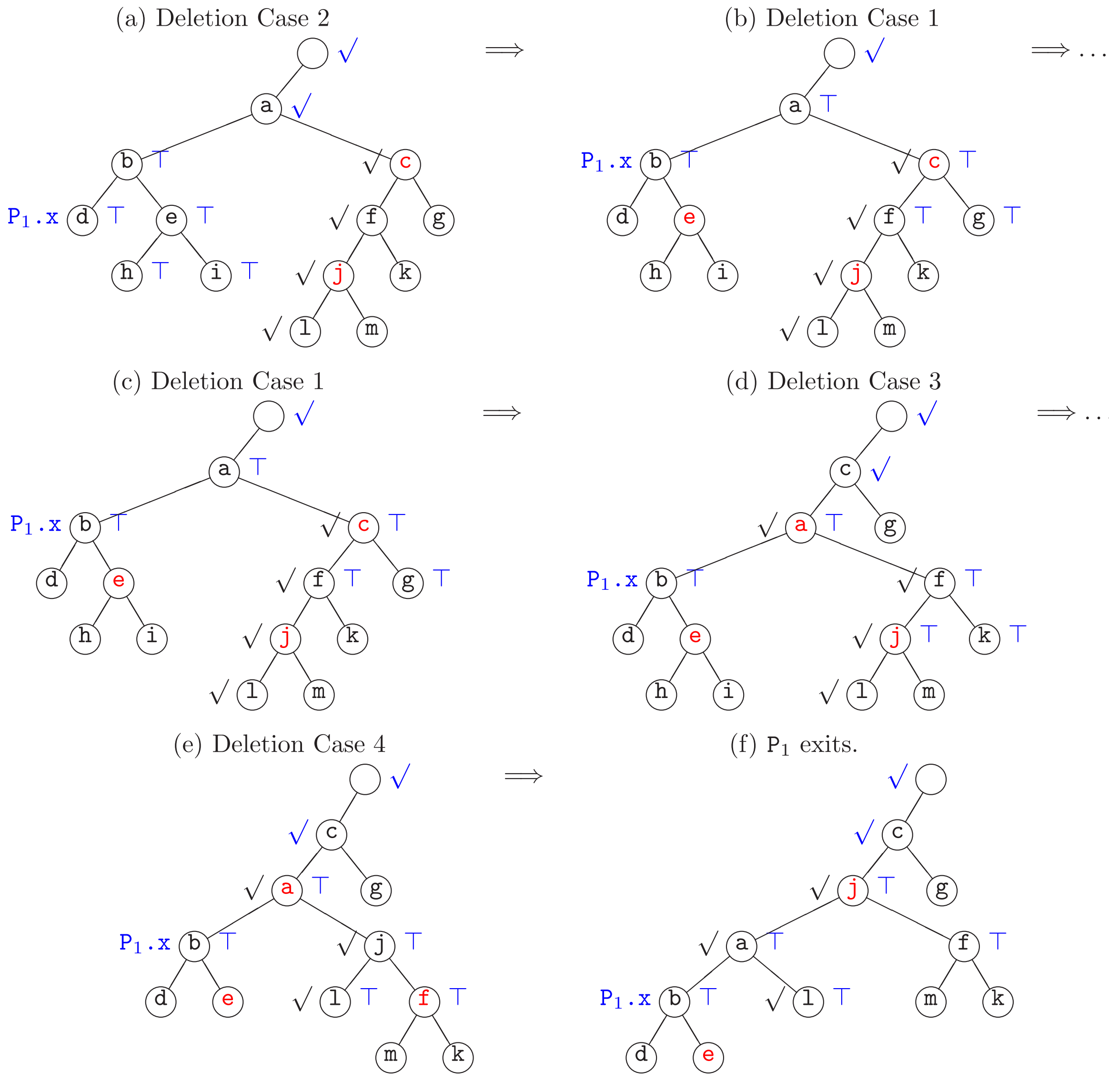


Figure 11: Example of intention markers with active processes.

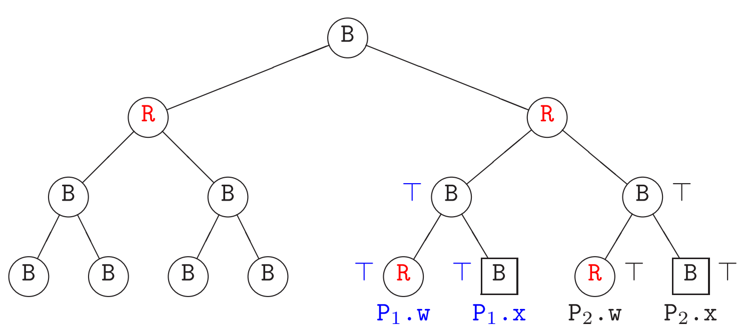
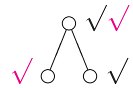


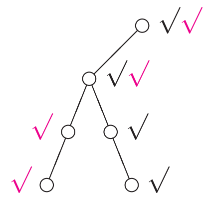
Figure 12: The necessity of intention markers.

However, as suggested earlier, there are certain special cases that arise in our algorithms. These are now discussed.

Assuming the use of four intention markers, we considered all the possible actions that a process could perform. In so doing, we found two situations where two intention markers (one from each of two different processes) could be incorrectly placed on a single node. This happened as a result of rotations during insertions and deletions. We call this the "double marker problem" and illustrate it in the following:



If a double marker were to be allowed, then we would have to increase the number of intention markers per process to five to ensure the same distance between processes. Moreover, if we allow the double marker problem to exist, then two consecutive double markers as illustrated below can occur:



If two consecutive double markers were allowed to exist, we would have to increase the number of intention markers to six to keep processes a safe distance apart. Then the creation of three consecutive double markers would be possible, continuing the cycle. Thus, if we do not prevent double markers, no number of intention markers can ensure a safe distance between processes. To prevent the double marker problem, we have introduced two rules (the "Spacing rule" and the "Move-Up rule").

The goal of the Spacing rule is to ensure there is always at least one "space" between any two processes. While this rule reduces the number of situations where double markers are created, it cannot, alone, eliminate the problem. This explains the need for the Move-Up rule. We first motivate and then state the Spacing rule. Then we show why the Spacing rule, alone, is insufficient to prevent double markers. This explanation provides a foundation for discussing the Move-Up rule which, together with the Spacing rule, prevents the double marker problem and ultimately makes the use of four intention markers sufficient.

Consider Figure 13. $\mathtt{P}_1$ is a deleting process and double markers result on node `w`. $\mathtt{P}_2$ is shown in black, $\mathtt{P}_3$ is shown in magenta. In the figure and the rest of the pictures drawn in this section, we assume that processes hold the flags of their local areas and omit the $\top$ symbols to avoid cluttering the figures.

In Figure 13, we observe that $\mathtt{P}_3$'s markers are on `lc` and `b` and that $\mathtt{P}_2$'s markers are on `w` and `rc`. After Case 3's right rotation on `w`, $\mathtt{P}_3$'s marker that was on `lc` is moved to `w` and creates a double marker problem on `w`. The motivation of the Spacing rule is to not allow $\mathtt{P}_3$ to come up the tree to a point where it can hold a marker on `lc` so the double markers cannot be created on `w`. The Spacing rule requires processes to leave a single space between themselves and another process when they are placing markers. Thus, in Figure 13 (a), $\mathtt{P}_3$ could not have put its marker on node `lc`.

For the definition of the Spacing rule, consider Figure 14. In Figure 14, blue $\mathtt{P}_1$ and black $\mathtt{P}_2$ are both doing deletions. Both processes have done Case 2 and are trying to move up. For both processes, `t` denotes the node on which a new marker is to be placed.

The Spacing Rule: A process can put a marker on node `t` only if the following three nodes have no flags or markers:

1. `t`,
2. the parent of `t`, and
3. the sibling of `t`.

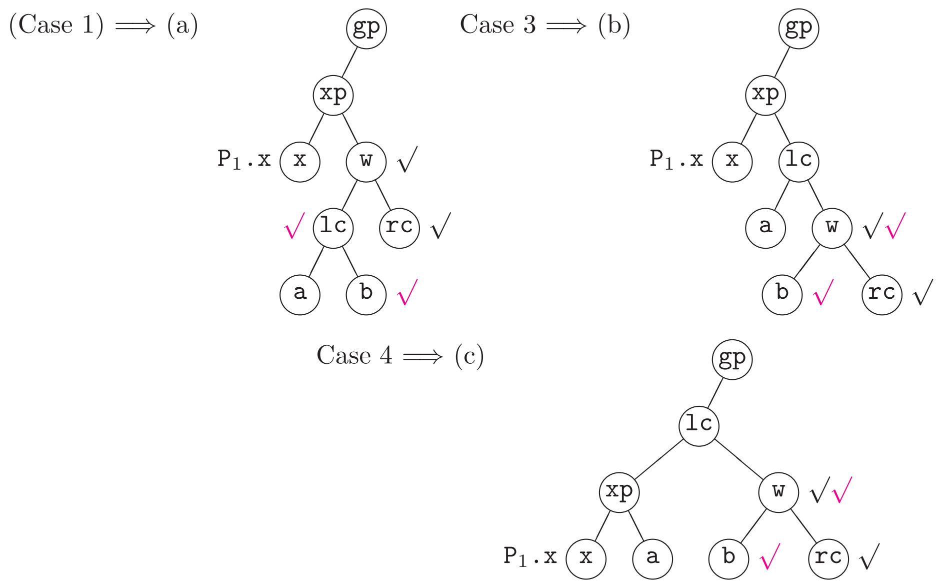


Figure 13: A situation where the double marker problem occurs.

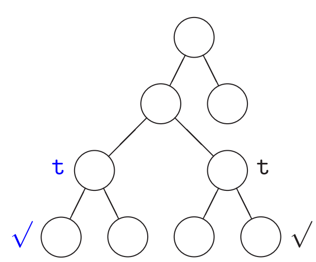


Figure 14: Need for the Spacing rule.

To do this check, a process obtains the flag of each of the above three nodes, using a sequence of CAS operations, in the order listed. After obtaining each flag, the process checks to see if any marker is on the node. If there is a marker, the process releases the flags obtained so far and restarts the task of obtaining the flags (starting from `t`). Similarly, if a CAS fails, then the process releases the flags that it has obtained so far and restarts the task. In Figure 14 then, either P_1 or P_2 will succeed in placing a marker on its `t`, but not both. The process that successfully places a marker will proceed first. The other will back off and have to retry placing its marker.

The Spacing rule guarantees a single space between processes when they are placing markers. Yet, the Spacing rule, by itself, is insufficient to produce a correct algorithm due to the possible effects of rotations. For instance, consider the situation shown in Figure 15. In Figure 15 (a), blue P_1, black P_2, magenta P_3, and cyan P_4 are all deletion processes that have placed their markers according to the Spacing rule. In part (b) of the figure, P_1 has moved up (by repetitions of Case 2) and is about to perform a rotation. P_1 performs deletion Cases 1, 3, and 4, and the results are shown in parts (c), (d), and (e), respectively. In part (f), P_2 moves one level up. In part (g), P_4 moves up and is about to perform deletion Case 1. The result after P_4 performs deletion Case 1 is shown in part (h), and corresponds to Figure 13 (a) (Situation 2), in a symmetric way, where a double marker can be produced. In Figure 15 (h), the processes are still too close together, and this is why the double marker problem can occur. To address this problem, the Move-Up rule was created and is now described.

Considering all the possible cases that a process could go through, with the Spacing rule applied, we found three situations where two processes became too close to one another. These situations are shown in Figures 16, 17 and 18. In all of the three situations, process P_1 performs deletion Case 3 and Case 4. Further, nodes `b` and `rc` each have markers from different processes. To deal with these situations, we introduced the Move-Up rule.

The purpose of the Move-Up rule is for the process, P_1, doing the rotation to allow one of the two processes that are brought too close to one other in the local area of P_1 to move up (thereby eliminating the problem). This rule is intended to ensure that at least one of the two processes will be kept far enough away from any other process that will perform a rotation so that no double markers can be created. The implementation of this rule requires a mechanism for inter-process communication (between P_1 and whichever process is selected to move up). Such a mechanism is assumed but no specific approach is prescribed.

The Move-Up Rule: After P_1 performs deletion Case 4, if P_1 finds itself in one of the situations shown in Figures 16, 17 and 18, P_1 allows one of the two processes that are too close — P_2 or P_3 in the pictures — to move up to P_1's `gp`. P_1 chooses the process that is located highest in the tree to move up. In Situations 1 and 2, P_2 will be allowed to move up. In Situation 3, either P_2 or P_3 can be chosen, and we arbitrarily chose P_2 in our algorithm. Note that a process that is allowed to move up by P_1 is not constrained by the Spacing rule to maintain its distance from nodes marked by P_1. It is, however, still constrained to maintain its distance from nodes marked by all other processes.

4.4 The Lock-Free Parallel Algorithms

We now revisit the `RB_Insert`, `RB_Insert_Fixup`, `RB_Delete` and `RB_Delete_Fixup` algorithms presented earlier in this paper. Newly added code is shown in red in the parallel versions of these algorithms: `Par_RB_Insert` (Figure 19), `Par_RB_Insert_Fixup` (Figure 25), `Par_RB_Delete` (Figure 27) and `Par_RB_Delete_Fixup` (Figure 29). The rotation routines are unchanged but the effects of the rotations create the key challenges in developing the parallel version of the algorithms.

Note that, in the following discussions of rotation specific aspects of the algorithms, we adopt the practice of discussing only one of the two possible (symmetric) rotations.

We assume that each node in the tree contains a boolean field `flag` and an integer `marker` which implement the flags and markers as described earlier in the paper. Further, if `marker` is 0, then the node is assumed to be unmarked. Additionally, we assume that the initial (empty) red-black tree is constructed

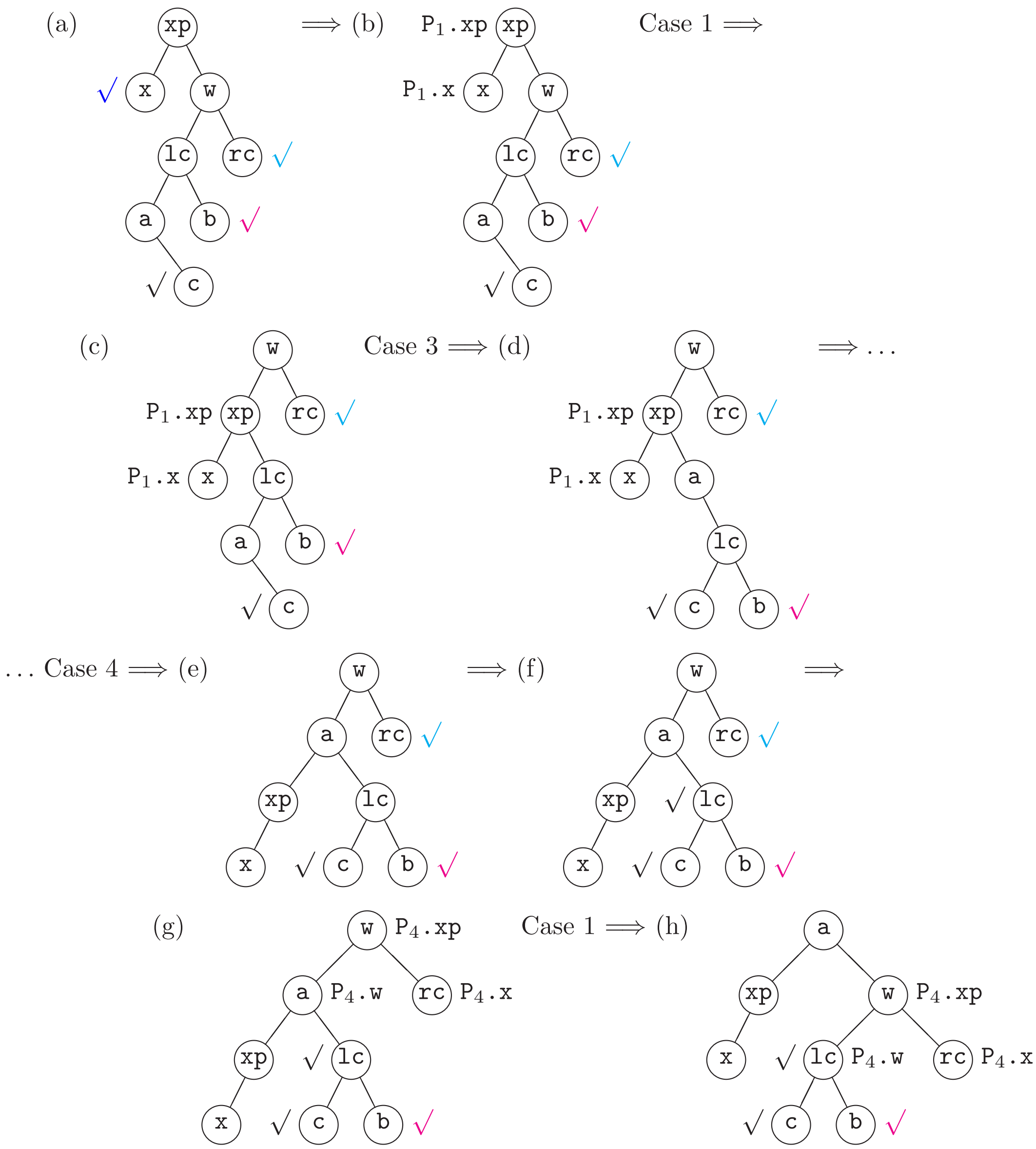


Figure 15: A situation where the markers of two processes come too close to one another.

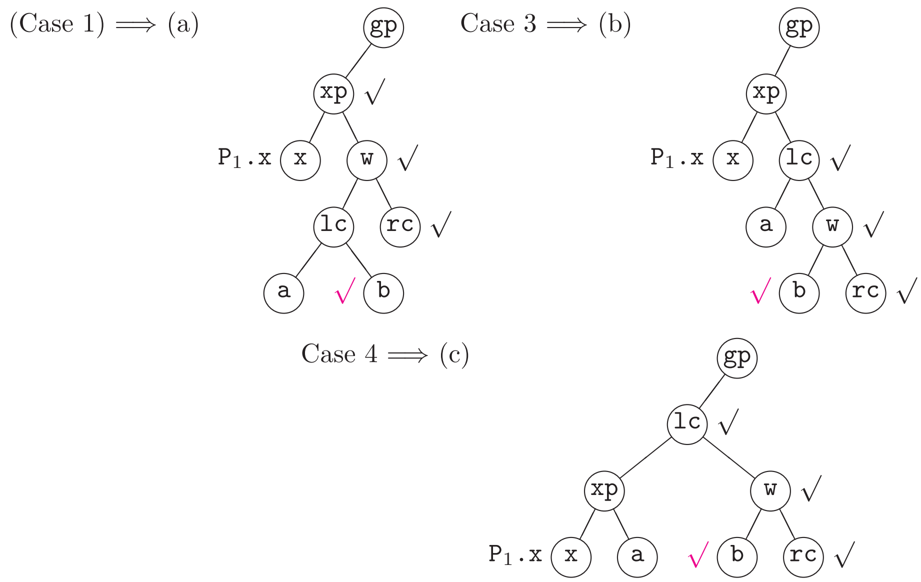


Figure 16: First situation requiring the Move-Up rule.

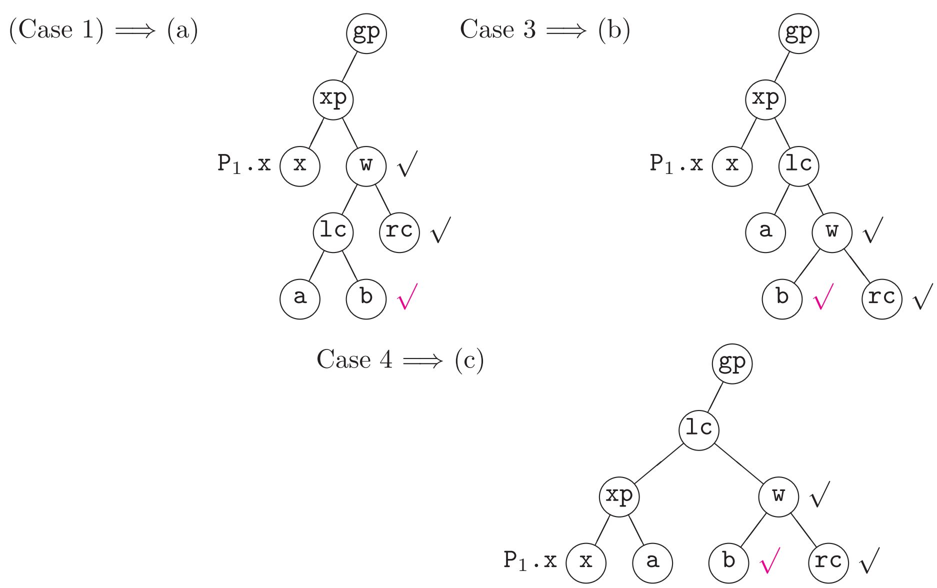


Figure 17: Second situation requiring the Move-Up rule.

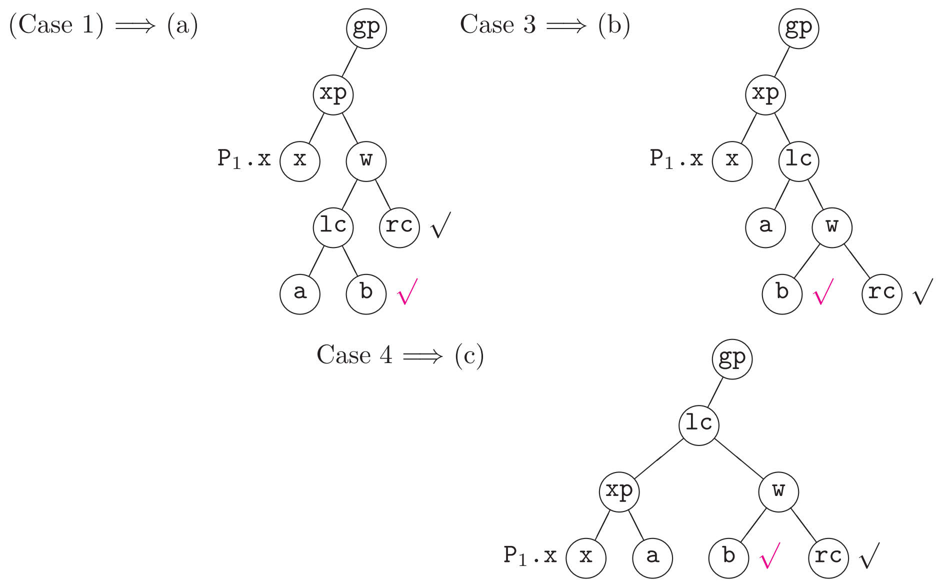


Figure 18: Third situation requiring the Move-Up rule.

before any operations are invoked and that it includes six dummy ancestor nodes and a dummy sibling to the NIL root (all coloured black) on which the necessary flags and markers can be placed during operations at or near the root of the tree. Note also that, unlike Cormen et al, we do not use a single sentinel node, `nil[T]`, to represent all the NIL external nodes. This eliminates potential contention when dealing with NIL nodes and thereby improves the performance of our algorithms (albeit at the expense of additional memory). Additionally, this eliminates the need for special case handling in many situations and simplifies the code for presentation.

As described earlier, using flags and markers, no synchronization (beyond that needed to handle the markers and flags themselves) is required during insertion processing. The routine `Par_RB_Insert` has been modified to acquire flags as it walks down the tree to the insertion point. If it ever fails to do so, the process has encountered a process (inserter or deleter) coming up the tree and must restart its search to avoid interfering with it. (This favours more "mature" operations and helps prevent starvation.) Once the insertion point is found, the routine `SetupLocalAreaForInsert` (Figure 20) is invoked to finish acquiring the remaining flags in the local area and to set markers on the four nodes above the insertion point. If this routine fails we also start a new search for the possibly changed insertion point.

The routine `SetupLocalAreaForInsert` calls the routine `GetFlagsAndMarkersAbove` (Figure 21), which uses the routine `GetFlagsForMarkers`(Figure 22). `GetFlagsAndMarkersAbove` in turn calls the routine `SpacingRuleIsSatisfied` (Figure 24), which is obviously the one responsible for enforcing the spacing rule, which, except for the effects of rotations, keeps all processes at least one "space" away from one another. These routines, along with `MoveInserterUp` (Figure 26) and `MoveDeleterUp` (Figure 30), are the primary routines for managing flags and markers.

`GetFlagsAndMarkersAbove` is responsible for acquiring additional markers as a process moves up the tree. It first acquires flags on the nodes corresponding to each of the four held intention markers. This ensures that it is safe to then attempt to add one (for deletion) or two (for insertion) additional markers above. If any flag/marker acquisitions fail, the process backs off and tries again. For presentation purposes, the routine `GetFlagsForMarkers` has been created which is used to get the flags on the existing marked nodes. The rest of the processing is done in `GetFlagsAndMarkersAbove` itself. Both routines check (using the `IsIn` routine on the `moveUpStruct`) whether or not a node whose flag is to be acquired has been inherited from another process via the Move-Up rule. The `moveUpStruct` is set in the routine `ApplyMoveUpRule`, which is

```
Par_RB_Insert(T,x) {
  restart: z = p[root];
  while(!CAS(flag[root[T]],FALSE,TRUE));
  y = root[T];
  while(y is not a NIL node) { // Find insert point z
    z = y;
    if (key[x] < key[y]) y = left[y];
    else y = right[y];
    if (!CAS(flag[y],FALSE,TRUE)) {
      flag[z]=FALSE; // release held flag
      goto restart;
    }
    if (y is not a NIL node)
      flag[z]=FALSE; // release old y's flag
  } /* end while */
  flag[x]=TRUE;
  if (!SetupLocalAreaForInsert(z)) {
    flag[z]=FALSE; // release held flag
    goto restart;
  }
  /* Place new node x as child of z */
  p[x] = z;
  if (z == p[root]) { root[T] = x; ; left[p[root]]=x; }
  else if (key[x] < key[z]) left[z] = x;
  else right[z] = x;
  left[x] = new NIL node;
  right[x] = new NIL node;
  colour[x] = red;
  RB_Insert_Fixup(T,x);
}
```

Figure 19: Pseudocode for parallel insertion.

```
SetupLocalAreaForInsert(z) {
   // try to get flags for rest of local area
   zp=p[z]; // take a copy of our parent pointer
   if (!CAS(flag[zp], FALSE, TRUE))
     return (FALSE);
   if (zp!=p[z]) {  // parent has changed - abort
     flag[zp]=FALSE;
     return (FALSE);
   }
   if (z==left[p[z]]) // uncle is the right child
     uncle=right[p[z]];
   else  // uncle is the left child
     uncle=left[p[z]];
   if (!CAS(flag[uncle], FALSE, TRUE)) {
     flag[p[z]]=FALSE;
     return (FALSE);
   }
   // Now try to get the four intention markers above p[z].
   // The second argument is only useful for deletes so we pass z
   // which is not an ancestor of p[z] and will have no effect.
   if (!GetFlagsAndMarkersAbove(p[z],z)) {
      flag[p[z]]=flag[uncle]=FALSE;
      return (FALSE);
    }
    return(TRUE);
 }
```

Figure 20: Pseudocode to setup the local area for an insertion.

```
GetFlagsAndMarkersAbove(start,numAdditional) {
   // Check for a moveUpStruct provided by another process (due to Move-Up rule
   // processing) and set 'PIDtoIgnore' to the PID provided in that structure.
   // Use the 'IsIn' function to determine if a node is in the moveUpStruct.
   // Start by getting flags on the four nodes we have markers on
   if (!GetFlagsForMarkers(start,moveUpStruct, pos1, pos2, pos3, pos4))
       return(FALSE);
   // Now get additional marker(s) above
   firstnew=p[pos4];
   if (!IsIn(firstnew,moveUpStruct) && (!CAS(flag[firstnew],FALSE,TRUE))) {
       ReleaseFlags(moveUpStruct, FAILURE, pos4, pos3, pos2, pos1);
       return (FALSE);
   }
   if ((firstnew!=p[pos4]) &&
       (!SpacingRuleIsSatisfied(firstnew,start,PIDtoIgnore,moveUpStruct))) {
       ReleaseFlags(moveUpStruct, FAILURE, firstnew, pos4, pos3, pos2, pos1);
       return (FALSE);
   }
   if (numAdditional==2) { // insertion so need another marker
       secondnew=p[firstnew];
       if ((!IsIn(secondnew,moveUpStruct) && !CAS(flag[secondnew],FALSE,TRUE))) {
           ReleaseFlags(moveUpStruct, FAILURE, firstnew, pos4, pos3, pos2, pos1);
             return (FALSE);
       }
       if ((secondnew!=p[firstnew]) &&
           (!SpacingRuleIsSatisfied(secondnew,start,PIDtoIgnore,moveUpStruct))) {
           ReleaseFlags(moveUpStruct, FAILURE, secondnew, firstnew, pos4, pos3,
                        pos2, pos1);
           return (FALSE);
   }  }
   marker[firstnew]=myPID;
   if (numAdditional==2) marker[secondnew]=myPID;
   // release the four topmost flags acquired to extend markers.
   // This leaves flags on nodes now in the new local area.
   if (numAdditional==2) ReleaseFlags(moveUpStruct, SUCCESS, secondnew);
   ReleaseFlags(moveUpStruct, SUCCESS, firstnew, pos4, pos3);
   if (numAdditional==1) ReleaseFlags(moveUpStruct, SUCCESS, pos2);
   return (TRUE)
}
```

Figure 21: Pseudocode to add intention markers as needed.

```
GetFlagsForMarkers(start, moveUpStruct, pos1, pos2, pos3, pos4) {
    pos1=p[start];
    if ((!IsIn(pos1,moveUpStruct) && (!CAS(flag[pos1],FALSE,TRUE)))
        return(FALSE);
    if (pos1!=p[start]) { // verify that parent is unchanged
        ReleaseFlags(moveUpStruct, FAILURE, pos1)
        return (FALSE);
    }
    pos2=p[pos1];
    if ((!IsIn(pos2,moveUpStruct) && !CAS(flag[pos2],FALSE,TRUE))) {
        flag[pos1]=FALSE;
        return(FALSE);
    }
    if (pos2!=p[pos1]) { // verify that parent is unchanged
        ReleaseFlags(moveUpStruct, FAILURE, pos2, pos1)
        return (FALSE);
    }
    pos3=p[pos2];
    if (!IsIn(pos3,moveUpStruct) && (!CAS(flag[pos1],FALSE,TRUE))) {
        flag[pos2]=flag[pos1]=FALSE;
        return(FALSE);
    }
    if (pos3!=p[pos2]) { // verify that parent is unchanged
        ReleaseFlags(moveUpStruct, FAILURE, pos3, pos2, pos1)
        return (FALSE);
    }
    pos4=p[pos3];
    if (!IsIn(pos4,moveUpStruct) && (!CAS(flag[pos1],FALSE,TRUE))) {
        flag[pos3]=flag[pos2]=flag[pos1]=FALSE;
        return(FALSE);
    }
    if (pos4!=p[pos3]) { // verify that parent is unchanged
        ReleaseFlags(moveUpStruct, FAILURE, pos4, pos3, pos2, pos1)
        return (FALSE);
    }
    return (TRUE)
}
```

Figure 22: Pseudocode to get flags for existing intention markers.

```
ReleaseFlags(moveUpStruct, success, nodesToRelease) {
    // Release flags identified in nodesToRelease
    foreach nd in nodesToRelease do
      if (success) { // release flag after successfully moving up
         if (!IsIn(nd,moveUpStruct)) flag[nd]=FALSE
         else { // nd is in the inherited local area
             if (IsGoalNode(nd, moveUpStruct))
                 // release unneeded flags in moveUpStruct
                 // and discard moveUpStruct
         }
      } else { // release flag after failing to move up
         if (!IsIn(nd,moveUpStruct)) flag[nd]=FALSE;
      }
}
```

Figure 23: Pseudocode to add intention markers as needed.

shown in Figure 31. If a node is in the `moveUpStruct`, the flag need not be acquired since it is already held.

The routine `ReleaseFlags` is called by `GetFlagsAndMarkersAbove` to release flags either due to rollback (`!success`) or completion of marker acquisition. It accepts a list, `nodesToRelease`, of nodes whose flags are to be released and releases them unless they are in (`IsIn`) the `moveUpStruct`. Each `moveUpStruct` also contains a "goal node" which identifies the node which, when reached by a process moving up, allows it to release the flags in the moveUpStruct. This handling is also done by `ReleaseFlags`.[6] Additionally, it is possible for a process which is doing a move up, to, itself, perform a deletion case 4 rotation and have to ask another process to move up. This results in the need to be able to pass a sequence of `moveUpStruct`s. This is discussed further after the routine `ApplyMoveUpRule` is presented.

The routine `SpacingRuleIsSatisfied` is used before a process sets a marker on a new node and checks to ensure that no marker is already set on that node, its parent or its sibling. This ensures that a distance of at least one space is maintained between processes (except, possibly, as a result of rotations). The routine handles two special cases. First, it checks that it is not attempting to acquire a flag on 'z', the node which, during a deletion, contained the value being removed. (This is because the flag is already held.) For insertion, calls to `SpacingRuleIsSatisfied` pass a node which cannot be in the local area so the tests on 'z' will have no effect. The second special case considered is related to the Move-Up rule. If the process has been selected to "move up" then the parameter `PIDtoIgnore` will contain the process ID of the "nearby" process that this one is allowed to move up past.

There is only a single change to `Par_RB_Insert_Fixup` which is to call `MoveInserterUp` (Figure 26) to manage the flags and markers as the process moves up the tree.

The routine `MoveInserterUp` (Figure 26) handles the process of moving an inserter up two levels (insertion, Case 2). It uses `GetFlagsAndMarkersAbove`, in part, to do this. New flags and markers are simply acquired and then the old ones are released.

Note that, before calling `Par_Delete`, we assume we have safely located the deletion point using a traversal technique like the one used in `Par_Insert` (Figure 19) and that the flag is thus held on node z at the beginning of `Par_Delete`. Similarly, we assume that the routine `FindSuccessor` called by `Par_Delete` also behaves in this way. Once the node to be deleted is found, the routine `SetupLocalAreaForDelete` (Figure 28) is invoked to finish acquiring flags in the local area and to set markers on the four nodes above the deletion point (again, using the routine `GetFlagsAndMarkersAbove`). If `SetupLocalAreaForDelete` fails we return `FALSE` and the caller must retry the deletion.

[6] Both `IsIn` and `IsGoalNode` simply return `FALSE` when no moveUpStruct is available.

```
SpacingRuleIsSatisfied(t,z,PIDtoIgnore) {
  // We hold flags on both t and z.
  // check that t has no marker set
  if (t!=z)
    if (marker[t]!=0)
      return (FALSE);
  // check that t's parent has no flag or marker
  tp = p[t];
  if (tp!=z) {
    if ((!IsIn(tp,moveUpStruct)) && (!CAS(flag[tp],FALSE,TRUE)))
      return (FALSE);
    if (tp!=p[t]) { // verify that parent is unchanged
      flag[tp]=FALSE;
      return (FALSE);
    }
    if (marker[tp]!=0) {
      flag[tp]=FALSE;
      return (FALSE);
    }
  }
  // check that t's sibling has no flag or marker
  if (t==left[tp]) ts=right[tp]
  else ts=left[tp];
  if ((!IsIn(ts,moveUpStruct)) && (!CAS(flag[ts],FALSE,TRUE))) {
    if (tp!=z) ReleaseFlags(moveUpStruct,FAILURE,tp);
    return (FALSE);
  }
  if ((marker[ts]!=0) && (marker[ts]!=PIDtoIgnore)) {
    ReleaseFlags(moveUpStruct,FAILURE,ts);
    if (tp!=z) ReleaseFlags(moveUpStruct,FAILURE,tp);
    return (FALSE);
  }
  if (tp!=z) ReleaseFlags(moveUpStruct,FAILURE,tp);
  ReleaseFlags(moveUpStruct,FAILURE,ts);
  return (TRUE);
}
```

Figure 24: Pseudocode to verify that the spacing rule is satisfied.

```
PAR_RB_Insert_Fixup(T,x) {
  while (colour[p[x]]==red) {
    if (p[x]==left[p[p[x]]]) {
      y = right[p[p[x]]];
      if (colour[y]==red) { // Case 1
        colour[p[x]] = black;
        colour[y] = black;
        colour[p[p[x]]] = red;
        x=MoveInserterUp(x);
      } else {
        if (x==right[p[x]]) { // Case 2
          x = p[x];
          Left_Rotate(T,x);
        } /* end if */
        colour[p[x]] = black; // Case 3
        colour[p[p[x]]] = red;
        Right_Rotate(T,p[p[x]]);
      } /* end else */
    } else { // p[x] = right[p[p[x]]]
      /* Symmetric to above. */
    } /* end if */
   } /* end while */
   colour[root[T]] = black;
 }
```

Figure 25: Pseudocode for parallel insertion fixup.

```
MoveInserterUp(oldx) { // Move up two levels
    // Check for a moveUpStruct from another process (due to Move-Up rule)
    // Get direct pointers
    oldp=p[oldx];
    oldgp=p[oldp];
    if (oldp==left[oldgp]) olduncle=right[oldgp]
    else olduncle=left[oldgp];
    // Extend intention markers (getting flags to set them)
    // From oldgp to top and two more. Also convert markers on
    // oldggp and oldgggp to flags.
    while (!GetFlagsAndMarkersAbove(oldgp,2)) ;
    // Get flags on rest of new local area (uncle)
    newx=oldgp;
    newp=p[newx];
    newgp=p[newp];
    if (newp==left[newgp]) newuncle=right[newgp]
    else newuncle=left[newgp];
    if (!IsIn(newuncle,moveUpStruct))
        while (!CAS(flag[newuncle],FALSE,TRUE)) ;
    // release flags on old local area (oldx, oldxp, olduncle)
    ReleaseFlags(moveUpStruct,SUCCESS, oldx, oldxp, olduncle);
    return(newx);
}
```

Figure 26: Pseudocode to move an inserter up the tree.

```
Par_RB_Delete(T,z) {
  // we now hold the flag of z
  if (left[z] is a NIL node || right[z] is a NIL node)
    y = z;
  else
    y = FindSuccessor(z); // key-order successor
  // we now hold the flag of y AND of z
  if (!SetupLocalAreaForDelete(y,z)) { // release flags
    flag[y]=FALSE;
    if (y!=z)
      flag[z]=FALSE;
    return (FALSE); // Deletion failed - try again
  }
  if (left[y] is not a NIL node) x = left[y];
  else x = right[y];
  // unlink y from the tree
  p[x] = p[y];
  if (p[y]==p[root])
    root[T] = x;
  else {
    if (y==left[p[y]]) left[p[y]] = x;
    else right[p[y]] = x;
  } // end else
  if (y!=z) {
    key[z] = key[y];
    flag[z]=FALSE;
  }
  if (colour[y]==black)
    RB_Delete_Fixup(T,x);
  else
    // Release flags and marker held in local area
  return (TRUE);
}
```

Figure 27: Pseudocode for parallel deletion.

```
SetupLocalAreaForDelete(y,z) {
  if (left[y] is not a NIL node) x = left[y];
  else x = right[y];
  // Try to get flags for the rest of the local area
  if (!CAS(flag[x],FALSE,TRUE)) return (FALSE);
  yp=p[y]; // keep a copy of our parent pointer
  if ((yp!=z) && (!CAS(flag[yp],FALSE,TRUE))) {
    flag[x]=FALSE; return (FALSE);
  }
  if (yp!=p[y]) {  // verify that parent is unchanged
    flag[x]=FALSE; if (yp!=z) flag[yp]=FALSE;
    return (FALSE);
  }
  if (y==left[p[y]]) w=right[p[y]];
  else w=left[p[y]];
  if (!CAS(flag[w], FALSE, TRUE)) {
    flag[x]=FALSE; if (yp!=z) flag[yp]=FALSE;
    return (FALSE);
  }
  if (w is not a NIL node) {
    wlc=left[w]; wrc=right[w];
    if (!CAS(flag[wlc], FALSE, TRUE)) {
      flag[x]=flag[w]=FALSE;
      if (yp!=z) flag[yp]=FALSE;
      return (FALSE);
    }
    if (!CAS(flag[wrc], FALSE, TRUE)) {
      flag[x]=flag[w]=flag[wlc]=FALSE;
      if (yp!=z) flag[yp]=FALSE;
      return (FALSE);
    }
  }
  if (!GetFlagsAndMarkersAbove(yp,z)) {
    flag[x]=flag[w]=flag[wlc]=flag[wrc]=FALSE;
    if (yp!=z) flag[yp]=FALSE;
    return (FALSE);
  }
  return(TRUE);
}
```

Figure 28: Pseudocode to setup the local area for a deletion.

The routine SetupLocalAreaForDelete provides the same processing for deletions that SetupLocalAreaForInsert does for insertions. Naturally, the local area created is that for deletion, not insertion.

The routine Par_RB_Delete_Fixup (Figure 29) may perform three different rotations, each of which modifies the executing process's local area and may affect markers set by other processes on nodes moved as a side-effect of the rotations. Much of the corrective action required is handled by the routines FixUpCase1 and FixUpCase3, described later. A process doing deletion Case 4 (the third and final rotation) is about to exit, so the handling of this case is simpler. In Case 2, the process moves up the tree, so like the insertion case, there is a routine, MoveDeleterUp (Figure 30), used to manage flags and markers as this happens. During Case 4, a call is made to ApplyMoveUpRule (Figure 31) which checks to see if the Move-Up rule needs to be used and which returns a flag, didMoveUp, accordingly. Finally, just before exiting, Par_RB_Delete_Fixup releases the flags it holds unless the Move-Up rule was applied, in which case the flags have been passed to the process selected to move up and, thus, need not be released.

The routine MoveDeleterUp (Figure 30) is similar to MoveInserterUp and is responsible for the process of moving a deleter up one level (deletion, Case 2). Like MoveInserterUp, it also uses GetFlagsAndMarkersAbove to do some of its work.

The routine ApplyMoveUpRule (Figure 31) checks for the three situations where the Move-Up rule is required (Figures 16, 17, and 18). If any of the situations apply, the routine selects a process to pass its flags to. It then creates a moveUpStruct containing the addresses of all the nodes in its local area and the process ID of the other of the two process that have been brought too close together by the deletion Case 4 rotate just performed. The moveUpStruct is then made available to the process selected to move up which will use it to "inherit" the flags held by the executing process. ApplyMoveUpRule then returns TRUE to indicate that a move up occurred and hence inform Par_RB_Delete_Fixup that it should not release the flags in its local area.

Recall that it is also possible for a process which is doing a move up, to, itself, ask another process to move up. Further, this may happen multiple times and leads to the creation of a sequence of moveUpStructs. In essence, if a process has a chain of one or more moveUpStructs then, after performing deletion case 4, it will create a new moveUpStruct describing its own local area, etc. and append it to the end of the existing chain which is made available, in its entirety, to the process selected to move up. This means that a process might hold a potentially large number of flags (though they would all be released quickly) and also that ReleaseFlags (Figure 23) must be able to process a list of moveUpStructs, each with their own goal node. Fortunately, these can be handled, individually and in order.[7]

The routine FixUpCase1 called from the routine Par_RB_Delete_Fixup (Figure 29) is responsible for dealing with the effects of a deletion case 1 rotation. There are two aspects to this problem; adjusting the local area of the process that has done the rotation and moving any relocated markers from other processes to where they belong after the rotation. After such a rotation, the former node w will have been rotated up to replace the former xp and the sub-tree rooted at wlc will have been re-attached under the former xp. This means that w becomes the new grandparent of x which is outside the local area as is x's uncle, the former wrc. Hence, flags must be released on these nodes. Before releasing the flag on w, however, its marker must be set. This is because w is now one of the four nodes above the local area on which we must hold intention markers. Correspondingly, the highest held intention marker (now, the fifth marker) must also be released. Additionally, the former wlc is now the new w but since we already have a flag on it, no additional processing is required. We also need to acquire flags on the two children of this node which will become the new wlc and wrc. This will always be possible since any processes beneath the new w must be at least 1 space away. This means that they cannot already hold the required flags. They may have markers on the nodes but this does not preclude the acquisition of a flag.

In addition to the processing just described, one or more markers in the local area that are held by other processes may have been relocated by the rotation just done and these must be moved to their correct positions. This will always be possible since we hold the necessary flags in the local area. The processing required to implement FixUpCase1 is tedious but not particularly insightful and therefore is omitted.

Finally, the routine FixUpCase3 which is called from Par_RB_Delete_Fixup (Figure 29), like FixUpCase1,

[7]The process of communicating and combining the moveUpStructs is subtle but do-able. We do not discuss it in this paper.

```
Par_RB_Delete_Fixup(T,x) {
  done=FALSE; didMoveUp=FALSE;
  while(x!=root[T] && colour[x]==black && !done) {
    if (x==left[p[x]]) {
      w = right[p[x]];
      if (colour[w]==red) { // Case 1
        colour[w] = black;
        colour[p[x]] = red;
        Left_Rotate(T,p[x]);
        w = right[p[x]];
        FixUpCase1(x);
      }
      if (colour[left[w]]==black &&
          colour[right[w]]==black) { // Case 2
        colour[w] = red;
        x=MoveDeleterUp(x);
      } else {
        if (colour[right[w]]==black) { // Case 3
          colour[left[w]] = black;
          colour[w] = red;
          Right_Rotate(T,w);
          FixUpCase3(x);
          w = right[p[x]];
        }  // end if
        colour[w] = colour[p[x]]; // Case 4
        colour[p[x]] = black;
        colour[right[w]] = black;
        Left_Rotate(T,p[x]);
        didMoveUp=ApplyMoveUpRule(x,w);
        done=TRUE;
      } /* end else */
    } else { // p[x] = right[p[p[x]]]
      /* Symmetric to above */
    } /* end if */
  } /* end while */
  if (!didMoveUp) {
    colour[x] = black;
    // Release local area flags and fix relocated markers
  }
}
```

Figure 29: Pseudocode for parallel deletion fixup.

```
MoveDeleterUp(oldx) { // Move up one level
    // Check for a moveUpStruct from another process (due to Move-Up rule)
    // Get direct pointers
    oldp=p[oldx];
    if (oldx==left[oldp]) oldw=right[oldp]
    else oldw=left[oldp];
    oldwlc=left[oldw]; oldwrc=right[oldw];
    // Extend intention markers (getting flags to set them)
    // From oldgp to top and one more. Also convert marker on
    // oldgp to a flag.
    while (!GetFlagsAndMarkersAbove(oldgp,1)) ;
    // Get flags on rest of new local area (w, wlc, wrc)
    newx=oldp;
    newp=p[newx];
    if (newx==left[newp]) neww=right[newp]
    else neww=left[newp];
    if (!IsIn(neww,moveUpStruct))
      while (!CAS(flag[neww],FALSE,TRUE)) ;
    newwlc=left[neww]; newwrc=right[neww];
    if (!IsIn(newwlc,moveUpStruct))
      while (!CAS(flag[newwlc],FALSE,TRUE)) ;
    if (!IsIn(newwrc,moveUpStruct))
      while (!CAS(flag[newwrc],FALSE,TRUE)) ;
    // release flags on old local area (oldx, oldwlc, oldwrc, oldw)
    ReleaseFlags(moveUpStruct,SUCCESS, oldx, oldw, oldwlc, oldwrc);
    return(newx);
}
```

Figure 30: Pseudocode to move a deleter up in the tree.

```
ApplyMoveUpRule(x,w) {
  // Check in our local area to see if two processes beneath
  // use have been brought too close together by our rotations.
  // The three cases correspond to Figures 17, 18 and 19.
  if (((marker[w]==marker[p[w]])&&(marker[w]==marker[right[w]])&&
       (marker[w]!=0)&&(marker[left[w]]!=0))
         // Situation from Figure 17
  || ((marker[w]==marker[right[w]])&&(marker[w]!=0)&&(marker[left[w]]!=0))
         // Situation from Figure 18
  || ((marker[w]==0)&&(marker[left[w]]!=0)&&(marker[right[w]]!=0)))
         // Situation from Figure 19
    {
      // Build structure listing the nodes we hold flags on
      // (moveUpStruct) and specifying the PID of the other
      // "too-close" process (marker[left[w]) and the goal (GP).
      // Make structure available to process id: marker[right[w]].
      return (TRUE)
    }
      return (FALSE)
}
```

Figure 31: Pseudocode to handle the Move-Up rule.

must adjust the calling process's local area and correct any misplaced markers occurring as a result, in this case, of a deletion case 3 rotation. After such a rotation, the former `wlc` becomes the new `w` and the former `w` becomes the new `wrc`. The former `wrc` is no longer in the local area and hence must have its flag released. A child of `wlc` now enters the local area as the new `wlc` and must have a flag acquired for it. No changes are required to the intention markers held by the rotating process but corrections will still be needed to those held by other processes that were affected by the rotation.

5 Conclusions and Future Work

In this paper we have presented lock-free algorithms for red-black trees. By using CAS to carefully ensure that concurrent processes do not get close enough to one another to interfere, we avoid the necessity of using less implementable synchronization primitives (e.g., TCAS and higher CAS extensions). Further, by remaining lock-free, we also avoid the negative side-effects and unavoidable overhead of lock-based concurrency control. Our technique allows significant concurrency without the need for costly large-scale rollbacks as may be required by other approaches. We expect that the overhead introduced (periodic, small scale rollbacks) will be minimal except under high contention scenarios. Overall, our synchronization operations do not affect the asymptotic running time of search and update operations which still require at most $O(\log n)$ time.

Directions for future research include certain optimizations to increase concurrency[8], doing an implementation of our algorithms to allow for a performance assessment against alternative (lock and STM based) strategies. We also plan to implement other complex data structures to identify common usage patterns for our approach and, ultimately, to develop a library of highly efficient CAS-based lock-free data structures that can be easily employed by programmers of shared memory multiprocessors to improve the efficiency

[8]Some optimizations have already been developed but they were omitted from the paper because they result in a more complicated algorithm.

and reliability (with respect to deadlock) of their code.

Acknowledgements

This research was supported, in part, by the Natural Sciences and Engineering Research Council of Canada under grant numbers RGPIN 155240-2002 and OGP-0194227.

References

[1] Maged M. Michael, Michael L. Scott. Relative performance of preemption-safe locking and non-blocking synchronization on multiprogrammed shared memory multiprocessors. *Proceedings of the* 11^{th} *International Parallel Processing Symposium*, April 1997. IEEE Computer Society Press: Geneva, Switzerland, 1997; 267–273.

[2] Michael Greenwald, David Cheriton. The synergy between nonblocking synchronization and operating system structure. *Proceedings of the* 2^{nd} *Symposium on Operating System Design and Implementation*, October, 1996. USENIX Association: Seattle, Washington, 1996; 123–136.

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms* (2ed). MIT Press: Cambridge, MA, 2001.

[4] Jianwen Ma. *Lock-Free Insertions on Red-Black Trees*. MSc thesis, University of Manitoba, October, 2003.

[5] Maurice Herlihy. Impossibility and universality results for wait-free synchronization. *Proceedings of the* 7^{th} *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, August 1988. ACM Press: Toronto, Canada, 1988; 276–290.

[6] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems* 1991; **11**(1):124–149.

[7] Serge A. Plotkin. Sticky bits and universality of consensus. *Proceedings of the* 8^{th} *ACM Symposium on Principles of Distributed Computing*, August 1989. ACM Press: Edmonton, Canada, 1989; 159–175.

[8] Maurice Herlihy. A methodology for implementing highly concurrent data structures. *Proceedings of the* 2^{nd} *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, March 1990. ACM Press: Seattle, Washington, 1990; 197–206.

[9] Greg Barnes. A method for implementing lock-free data structures. *Proceedings of the* 5^{th} *ACM Symposium on Parallel Algorithms and Architectures*, August 1993. ACM Press: Velen, Germany, 1993; 261–270.

[10] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems* 1993; **15**(5):745–770.

[11] Sundeep Prakash, Yann-Hang Lee, Theodore Johnson. *Nonblocking algorithms for concurrent data structures*. University of Florida, Technical Report 91-002, 1991.

[12] John Turek, Dennis Shasha, Sundeep Prakash. Locking without blocking: Making lock based concurrent data structure algorithms nonblocking. *Proceedings of the* 11^{th} *ACM Symposium on Principles of Database Systems*, July 1992. ACM Press: San Diego, CA, 1992; 212–222.

[13] Anthony LaMarca. A performance evaluation of lock-free synchronization protocols. *Proceedings of the* 13^{th} *annual ACM Symposium on Principles of Distributed Computing*, August 1994. ACM Press: Los Angeles, CA, 1994; 130–140.

[14] Maged M. Michael, Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. *Proceedings of the* 15^{th} *ACM Symposium on Principles of Distributed Computing*, May 1996. ACM Press: Philadelphia, PA, 1996; 267–275.

[15] Sundeep Prakash, Yann-Hang Lee, Theodore Johnson. A nonblocking algorithm for shared queues using compare-and-swap. *IEEE Transactions on Computers* 1994; **43**(5):548–559.

[16] R. K. Treiber. *Systems programming: Coping with parallelism*. IBM Almaden Research Center, Research Report RJ5118, 1986.

[17] Maged M. Michael. High Performance Dynamic Lock-Free Hash Tables and List-Based Sets. *Proceedings of the* 14^{th} *Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA 2002)*, August 2002. ACM Press: Winnipeg, Canada, 2002; 73–82.

[18] Mikhail Fomitchev, Eric Ruppert. Lock-free linked lists and skip lists. *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing (PODC '04)*, July 2004. ACM Press: St. Johns, Canada, 2004; 50–59.

[19] Hakan Sundell, Philippas Tsigas. Scalable and lock-free concurrent dictionaries. *Proceedings of the 2004 ACM symposium on Applied computing (SAC '04)*, March 2004. ACM Press: Nicosia, Cyprus, 2004; 1438–1445.

[20] Timothy L. Harris, Keir Fraser, Ian A. Pratt. A Practical Multi-word Compare-and-Swap Operation. *Proceedings of the 16th International Conference on Distributed Computing (DISC 2002)*, October, 2002. Springer Verlag: Toulouse, France, 2002; 265–279.

[21] Maurice Herlihy, Mark Moir, Victor Luchangco, William N. Scherer. Software transactional memory for dynamic-sized data structures. *Proceedings of the* 22^{nd} *ACM Symposium on Principles of Distributed Computing*, July 2003. ACM Press: Boston, MA, 2003; 92–101.

[22] Tim Harris, Keir Fraser. Language support for lightweight transactions. *Proceedings of the* 18^{th} *Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2003. ACM Press: Anaheim, CA, 2003; 388–402.

[23] Jong Ho Kim. *High-Concurrency Lock-Free Algorithms for Red-Black Trees*. MSc thesis, University of Manitoba, October, 2005.