Lock-Free Linked Lists Using Compare-and-Swap

John D. Valois
Rensselaer Polytechnic Institute
valoisj@cs.rpi.edu

Abstract

Lock-free data structures implement concurrent objects without the use of mutual exclusion. This approach can avoid performance problems due to unpredictable delays while processes are within critical sections. Although universal methods are known that give lock-free data structures for any abstract data type, the overhead of these methods makes them inefficient when compared to conventional techniques using mutual exclusion, such as spin locks.

We give lock-free data structures and algorithms for implementing a shared singly-linked list, allowing concurrent traversal, insertion, and deletion by any number of processes. We also show how the basic data structure can be used as a building block for other lock-free data structures.

Our algorithms use the single word Compare-and-Swap synchronization primitive to implement the linked list directly, avoiding the overhead of universal methods, and are thus a practical alternative to using spin locks.

1 Introduction

A *concurrent object* is an abstract data type that permits concurrent operations that appear to be atomic. We can implement a concurrent object as a data structure in shared memory and a set of algorithms that manipulate the data structure using atomic *synchronization primitives*, such as READ, WRITE, FETCH&ADD, and COMPARE&SWAP. Care is required to synchronize concurrent processes so that the data structure is not corrupted and so that operations return the correct results. The conventional way to do this is with *mutual exclusion*, guaranteeing exclusive access to a process manipulating the data structure.

Mutual exclusion is well understood; in particular, a number of efficient *spin locking* techniques have been developed [3, 8, 20]. However, the delay of a process while in a critical section (for example, due to a page fault, multitasking preemption, memory access latency, etc.) forms a bottleneck which can cause performance problems such as convoying and priority inversion.

Lock-free data structures implement concurrent objects without the use of mutual exclusion. Such data structures may be able to guarantee that some process will complete its operation in a finite amount of time, even if other processes halt; in this case the data structure is *non-blocking*. If the data structure can guarantee that every (non-faulty) process will complete its operation in a finite amount of time, then it is *wait-free*.

Although several *universal* methods are known for wait-free implementation of any arbitrary concurrent object, they involve considerable overhead, making them impractical, especially compared to spin locks. It is sometimes possible to devise lock-free data structures that implement a particular concurrent object directly, without the use of universal methods. Such techniques can offer the benefits of lock-free synchronization without sacrificing efficiency.

We present algorithms and data structures that directly implement a non-blocking singly-linked list. To our knowledge, these are the first such algorithms to allow processes to arbitrarily traverse the linked list structure, inserting and deleting nodes at any point in the list, using only the commonly available COMPARE&SWAP primitive[1], and providing performance competitive with spin locks.

A linked list is also useful as a building block for other concurrent objects. We show how the lock-free linked list can be used to build several non-blocking implementations of a concurrent *dictionary* object.

The rest of this paper is organized as follows: Section 2 reviews related work, and describes the requirements of the linked list data structure as well as some of the problems encountered in implementing one in a lock-free manner. Section 3 describes the data structure and basic algorithms for list traversal, insertion, and deletion. Section 4 shows how to extend these techniques to implement higher-level abstract data types such as a dictionary. Section 5 discusses the management and allocation of memory, garbage collection, and the ABA problem. Section 6 concludes with some directions for further work.

2 Related Work

Researchers have considered the benefits of avoiding mutual exclusion since at least the early 1970's [6]. Lamport [17] gave the first lock-free algorithm for the problem of a singe-writer/multi-reader shared variable. Herlihy [10] proved that for non-blocking implementation of most interesting data types (linked lists among them), a synchronization primitive that is *universal*, in conjunction with READ and WRITE, is both necessary and sufficient. A universal

[1] We also use Test&Set and Fetch&Add; however, these are easily implemented with Compare&Swap.

PODC 95 Ottawa Ontario CA © 1995 ACM 0-89791-710-3/95/08..$3.50

```
COMPARE&SWAP(a : address, old, new : word)
  returns boolean

    BEGIN ATOMIC
1     if a^ ≠ old
2       a^ ← new
3       return TRUE
4     else
5       return FALSE
    END ATOMIC
```

Figure 1: The COMPARE&SWAP synchronization primitive.

primitive is one that can solve the *consensus problem* [7] for any number of processes; COMPARE&SWAP is a universal primitive.

The first universal method was given by Herlihy [13]; many others followed [1, 4, 11, 22, 26]. However, it has become increasingly apparent that universal methods suffer from several sources of inefficiency, such as wasted parallelism, excessive copying, and generally high overhead.

In addition to the universal methods, algorithms have also been developed for lock-free objects that are implemented directly. Most of this work has focused on the FIFO queue data type (cf. [27] for many references), but algorithms have also been developed for sets [18], union-find [2], scheduling [16], and garbage collection [12]. There has also been a large body of work on implementing more primitive types of objects, such as atomic registers and counters. We note that many of these papers present data structures that are based on the linked list; however, none of them permit modifications to the interior of the list.

Massalin and Pu [19] coined the term *lock-free* and implemented a multiprocessor operating system kernel using lock-free data structures. However, their algorithms require a two word version of the COMPARE&SWAP synchronization primitive that is not widely available.

2.1 Requirements

The abstract concept of a list is a collection of items which have a linear order; i.e., each item in the list has a *position*. A singly-linked list data structure consists of a collection of *cells*, each representing an item in the list. These cells contains a number of fields, in particular a field *next*, which contains a pointer to the cell occupying the next position in the list. Other fields may contain memory management information, data dependent on the application using the list, etc. A special *root pointer* points to the first cell in the list.

We use the following notation in our algorithms: if p is a pointer, then $p^\wedge$ represents the contents of the memory location pointed to. If p points at a structure in memory, for example a cell, then $p^\wedge.field$ refers to a field within the structure.

We use COMPARE&SWAP as our main synchronization primitive. The COMPARE&SWAP primitive takes as arguments the pointer, and old and new values. As shown in Figure 1, it atomically checks the value of the pointer, and if it is equal to the old value, updates the pointer to the new value. In either case, it returns an indication of whether it succeeded or not.

The COMPARE&SWAP primitive is often used to *swing* pointers; to atomically change them from one value to another. We will also make use of the primitives TEST&SET and FETCH&ADD. Both atomically read and modify the value of a shared memory location, returning the original value. The TEST&SET primitive sets the new value to TRUE, while FETCH&ADD adds and arbitrary value to it.

The COMPARE&SWAP primitive is widely available, being found on many common architectures. Newer architectures include the LOAD-LOCKED and STORE-CONDITIONAL primitives, which can implement COMPARE&SWAP. It can also be implemented on uniprocessors using the technique of *atomic restartable sequences* [5]. Finally, we note that there is growing support for providing COMPARE&SWAP in distributed memory machines as well [9, 21].

In order to make concrete the abstract notion of position, it is convenient to introduce the idea of a *cursor*. A cursor is associated with an item in the list; the cursor is said to be *visiting* that item. A cursor may also be visiting a distinguished position at the end of the list which is not associated with any item.

All access to the list is accomplished via a cursor. When a new cursor is created, it is visiting the first item in the list (or the special end position if the list is empty). An existing cursor can *traverse* the list by moving from its current position to the next one in the list.

New items can be added to the list by inserting them at the position immediately preceding that visited by a given cursor. An item being visited by a cursor can be removed from the list by deleting it.

We require our lock-free objects to be non-blocking, but not necessarily wait-free. The non-blocking requirement ensures that the delay of one process cannot affect any other; the wait-free property, while desirable, imposes too much overhead upon the implementation. Furthermore, starvation at high levels of contention is more efficiently handled by techniques such as exponential backoff (for example, see [15]).

We also require our objects to be *linearizable* [14]; this implies that operations appear to happen atomically at some point during their execution. Proofs that our data structures are linearizable are beyond the scope of this paper, but are straightforward.

2.2 Problems

The use of COMPARE&SWAP to swing pointers is susceptible to the *ABA problem*, discussed in depth in Section 5. Our solution relies on the careful memory management, and in particular on the use of two operations, SAFEREAD and RELEASE. They will be fully described in Section 5; however, for the time being SAFEREAD can be treated as a normal READ and RELEASE can be treated as a no-op. In addition to these two operations, we will also discuss the management of free cells in Section 5.

At first glance, it may not seem too difficult to implement a lock-free linked list. Traversing this data structure is simple, since it does not involve changes to the list structure. Insertion of new cells is straightforward using COMPARE&SWAP; given a pointer q to a new cell, and pointers p and s to cells in the list such that $p^\wedge.next = s$, we initialize $q^\wedge.next = s$ and then swing the *next* field of p to q. If the operation succeeds, then the new cell has been linked into the list; otherwise, a concurrent operation has

changed the list structure, and we must retry the operation after re-reading the pointers.

However, when we consider deleting cells from the list we run into difficulties. First, note that when we delete a cell from the list, other processes may have cursors visiting that cell; we would like these processes to be able to continue using their cursors to traverse the list, as well as to access the contents of the deleted cell. This can be accomplished by simply keeping the contents of the deleted cell intact; but this will complicate the reuse of cells that have been deleted from the list. We call this *cell persistence.*

Two more serious difficulties are the following. When we delete a cell from the list, we swing the *next* pointer in the preceding cell to point at the following cell. Suppose that another process concurrently inserts a cell at the position immediately following a cell being deleted. It is possible that we might end up with the situation in Figure 2; the cell containing B has been deleted successfully, but the cell containing C has not been inserted into the list correctly.

Another problem occurs if another process concurrently deletes an adjacent cell; this can result in one of the deletions being undone, as shown in Figure 3. These problems stem from the fact that we cannot observe the state of the *next* fields in two different cells simultaneously, and overcoming them would seem to require the use of a synchronization primitive capable of operating on two words of memory simultaneously. However, we shall show in the next section that this is not the case.

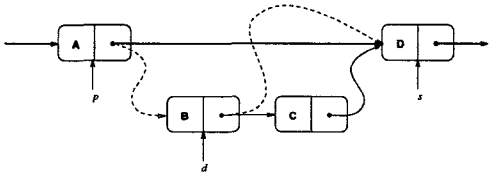

Figure 2: Deletion of B concurrent with insertion of C.

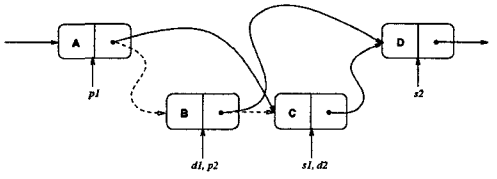

Figure 3: Concurrent deletion of B and C; second is undone.

3 Auxiliary Nodes and Basic Operations

In order to overcome the problems described in the last section, we add *auxiliary nodes* to the data structure. An auxiliary node is a cell that contains only a *next* field. We require that every normal cell in the list have an auxiliary node as its predecessor and as its successor. We permit "chains" of auxiliary nodes in the list (i.e., we do not require that every auxiliary node have a normal cell as its predecessor and successor), although such chains are undesirable for performance reasons.

The list also contains two dummy cells as the first and last normal cells in the list. These two cells are pointed at by the root pointers *First* and *Last*. These dummy cells need not, respectively, be preceded and followed by auxiliary nodes. Thus, an empty list data structure consists of these two dummy cells separated by an auxiliary node (Figure 4).

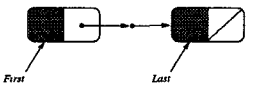

Figure 4: An empty linked list, with two dummy nodes and an auxiliary node.

A cursor is implemented as three pointers into the data structure: *target* is a pointer to the cell at the position the cursor is visiting. If the cursor is visiting the end-of-list position, then *target* will be equal to *Last*.

The pointer *pre_aux* points to an auxiliary node in the data structure. For a cursor c, if $c\hat{}.pre\_aux = c\hat{}.target$, then the cursor is *valid*; otherwise it is *invalid*.

The pointer *pre_cell* points to a regular cell in the data structure. This pointer is used only by the TRYDELETE operation described below.

An invalid cursor indicates that the structure of the list in the vicinity of the cursor has changed (due to a concurrent insertion or deletion by another process) since the pointers in the cursor were last read. The UPDATE algorithm, given in Figure 5, examines the state of the list and updates the pointers in the cursor so that it becomes valid.

Since the list structure contains auxiliary nodes (perhaps more than one in a row), the UPDATE algorithm must skip over them. If two adjacent auxiliary nodes are found in the list, the UPDATE algorithm will remove one of them.

Traversal of the list data structure is accomplished using the FIRST and NEXT operations, which use the UPDATE operation. Algorithms are given in Figures 6 and 7. The NEXT operation returns FALSE if the cursor is already at the end of the list and cannot be advanced.

Adding new cells into the list requires the insertion of both the cell and a new auxiliary node. This insertion is restricted to occur in the following way: The new auxiliary node will follow the new cell in the list, and insertion can only occur between an auxiliary node and a normal cell, as shown in Figure 8.

Figure 9 gives an algorithm, which takes as arguments a cursor and pointers to a new cell and auxiliary node. The algorithm will try to insert the new cell and auxiliary node

```
UPDATE(c : cursor)

1   if c^.pre_aux^.next = c^.target
2     return
3   p ← c^.pre_aux
4   n ← SAFEREAD(p^.next)
5   RELEASE(c^.target)
6   while n ≠ Last and n^ is not a normal cell
7     COMPARE&SWAP(c^.pre_cell^.next, p, n)
8     RELEASE(p)
9     p ← n
10    n ← SAFEREAD(p^.next)
11  c^.pre_aux ← p
12  c^.target ← n
```

Figure 5: The cursor UPDATE algorithm.

```
FIRST(c : cursor)

1    c^.pre_cell ← SAFEREAD(First)
2    c^.pre_aux ← SAFEREAD(First^.next)
3    c^.target ← NULL
4    UPDATE(c)
```

Figure 6: The FIRST algorithm.

```
NEXT(c : cursor)
  returns boolean

1    if c^.target = Last
2      return FALSE
3    RELEASE(c^.pre_cell)
4    c^.pre_cell ← SAFEREAD(c^.target)
5    RELEASE(c^.pre_aux)
6    c^.pre_aux ← SAFEREAD(c^.target^.next)
7    UPDATE(c)
8    return TRUE
```

Figure 7: The NEXT algorithm.

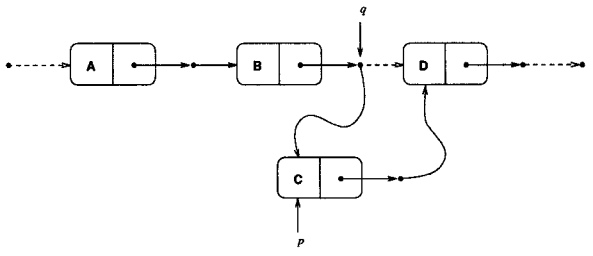

Figure 8: Inserting a new cell and auxiliary node.

```
TRYINSERT(c : cursor, q : cell^, a : aux. node^)
  returns boolean

1    WRITE(q^.next, a)
2    WRITE(a^.next, c^.target)
3    r ← CSW(c^.pre_aux, c^.target, q)
4    return r
```

Figure 9: The TRYINSERT algorithm.

at the position specified by the cursor, returning the value TRUE if successful.

If the cursor becomes invalid, then the operation returns without inserting the new cell and returns the value FALSE. This allows a higher-level operation to detect that a change to the structure of the list occured and to take it into account before attempting to insert the new cell again. For example, in the next section we show how the items in the list can be kept sorted using this technique.

Given a valid cursor, the cell that it is visiting can also be deleted from the list. As with the insertion of new cells, if the list structure changes (i.e., the cursor becomes invalid) then the operation fails and must be tried again. Figure 10 gives the TRYDELETE algorithm.

The deletion of the cell from the list leaves an "extra" auxiliary node; concurrent processes deleting adjacent cells can result in longer chains. Most of the TRYDELETE algorithm is concerned with removing the extra auxiliary nodes from the list. Normally, removing the extra auxiliary node that results from the deletion of a cell from the list is accomplished by simply swinging the pointer in the cell pointed at by the *pre_cell* pointer in the cursor.

However, this does not always work; in particular, this cell may have itself been deleted from the list, in which case swinging its *next* pointer will not remove the extra auxiliary node. In order to overcome this problem, we add a *back_link* field to the normal cells in the list. When a cell is deleted from the list, the *pre_cell* field of the cursor is copied into the cell's *back_link* field. The TRYDELETE algorithm can then use these pointers to traverse back to a cell that has not been deleted from the list.

With just two processes, it is possible to create a chain of auxiliary nodes (with no intervening normal cells) of any length. However, any such chain can exist in the list only as long as some process is executing the TRYDELETE algorithm. If all deletions have been completed, then the list will contain no extra auxiliary nodes.

To see this, assume that there is a chain of two or more auxiliary nodes in the list. Let x be the normal cell that was deleted from between the first two auxiliary nodes in the chain. Note that this implies that the normal cell that immediately preceded x in the list has not been deleted.

By assumption, the operation that deleted x has completed. Consider the loop at lines 17–21 of the TRYDELETE algorithm. The only way for the process to exit this loop, and hence to complete the operation, is for another deletion operation to have extended the chain of auxiliary nodes by deleting the normal cell y immediately following the chain, since the cell x is at the front of the chain.

Furthermore, the deletion of y must have occured after the operation deleting x had set its *back_link* pointer at line 6; otherwise the auxiliary node following y would

```
TRYDELETE(c : cursor)
  returns boolean

 1    d ← c^.target
 2    n ← c^.target^.next
 3    r ← CSW(c^.pre_aux^.next, d, n)
 4    if r ≠ TRUE
 5      return FALSE
 6    WRITE(d^.back_link, c^.pre_cell)
 7    p ← c^.pre_cell
 8    while p^.back_link ≠ NULL
 9      q ← SAFEREAD(p^.back_link)
10      RELEASE(p)
11      p ← q
12    s ← SAFEREAD(p^.next)
13    while n^.next^ is not a normal cell
14      q ← SAFEREAD(n^.next)
15      RELEASE(n)
16      n ← q
      repeat
17      r ← CSW(p^.next, s, n)
18      if r = FALSE
19        RELEASE(s)
20        s ← SAFEREAD(p^.next)
21    until r = TRUE
          or p^.back_link ≠ NULL
          or n^.next is not a normal cell
22    RELEASE(p)
23    RELEASE(s)
24    RELEASE(n)
25    return TRUE
```

Figure 10: The TRYDELETE algorithm.

```
FINDFROM(k : key, c : cursor)
  returns boolean

1    while c^.target ≠ Last
2      if c^.target^.key = k
3        return TRUE
4      else if c^.target^.key > k
5        return FALSE
6      else
7        NEXT(c)
8    return FALSE
```

Figure 11: The FINDFROM algorithm.

have been included in the chain found in lines 13–16. Thus, the chain of *back_link* pointers followed by the process that deleted y will lead to the same normal cell that preceded x.

Now, the only way for the operation that deleted y to have completed is for the same reason as above; i.e., another TRYDELETE operation must extend the chain of auxiliary nodes by deleting a cell z. Since the length of the list must be finite, there must be a last such deletion which, but by the argument above, cannot have completed. Thus this operation must still be in progress, contradicting the assumption that there were no TRYDELETE operations in progress.

4 Dictionaries

A linked list is useful as a building block for other data structures. We now show how the ideas in the last section can be applied to the problem of implementing various lock-free data structures for the dictionary abstract data type.

A *dictionary* contains a collection of items which are distinguished by distinct *keys*, and provides the operations FIND, INSERT, and DELETE. Using the data structures and algorithms presented in Section 3, we can implement a non-blocking dictionary using four data structures: a sorted list, a hash table, a skip list, and a binary search tree.

4.1 List Structures

We will assume that each cell has a field *key* which contains the unique key for the item stored in the cell. We will ensure that the keys of items stored in the dictionary are unique be keeping the items in the list sorted by their key values. Figure 11 gives an algorithm that searches the list, starting from a given cursor position, for a cell containing a given key. It returns a boolean value indicating whether or not an item with the requested key was found. The dictionary FIND operation is implemented by using this operation, starting from the first position in the list.

The dictionary INSERT operation is performed by the algorithm in Figure 12. It is necessary to first ensure that an item with the same key is not already in the dictionary. If one is not, then the FINDFROM algorithm will leave the cursor positioned in the correct place to insert the new cell.

If the insertion of the new cell fails due to changes to the list structure by concurrent operations, it is necessary to check again that the key value will be unique, after updating the value of the cursor. Note that the cursor UPDATE algorithm ensures that if another cell is inserted with the

```
INSERT(k : key)

1       FIRST(c)
2       q ← new cell
3       a ← new aux. node
4       initialize other fileds of q
    loop:
5       r ← FINDFROM(k, c)
6       if r = TRUE
7         return
8       r ← TRYINSERT(c, q, a)
9       if r = TRUE
10        return
11      UPDATE(c)
12      goto loop
```

Figure 12: The INSERT algorithm.

```
DELETE(k : key)

1     FIRST(c)
  loop:
2     r ← FINDFROM(k, c)
3     if r = FALSE
4       return
5     r ← TRYDELETE(c)
6     if r = TRUE
7       return
8     UPDATE(c)
9     goto loop
```

Figure 13: The DELETE algorithm.

same key, the cursor will be positioned in such a way that the FINDFROM algorithm will find it.

The dictionary DELETE operation is accomplished in a similar way; the FINDFROM algorithm is used to locate the position of the cell containing the given key (if it is in the list), and the TRYDELETE algorithm is then used to delete the cell. If the TRYDELETE algorithm fails, we update the cursor and continue the search for the key.

We can compare the performance of this non-blocking concurrent dictionary implementation to a similar sequential implementation using a sorted linked list. A non-constant factor slowdown can come from two sources: work done traversing extra auxiliary nodes in the list structure, and repetitive calls to TRYINSERT and TRYDELETE. For a single operation, it is impossible to place bounds on this extra work.

However, we can bound the amortized work by considering a sequence of dictionary operations performed by a number of processes. With p concurrent processes, each successfully completed operation can cause $p-1$ concurrent processes to have to retry a TRYINSERT or TRYDELETE operation. In addition, in the worst case each operation may have to traverse an extra auxiliary node left by every previous operation. Thus, the total work done by the concurrent non-blocking implementation for a sequence of n operations by p processes is $O(n^2)$, within a constant factor of optimal.

A straightforward extension of this implementation uses a hash table. In this case, if we assume that the hash function evenly distributes the operations across the lists, then we would expect the extra work done to be $O(1)$.

We can implement a lock-free skip list [24] as a collection of k sorted singly-linked lists[2], such that higher level lists contain a subset of the cells in lower level lists. As in [23], insertions and deletions are performed one level at a time, insertions starting with the bottom level and working up, and deletions starting at the top and working down.

Although the structure of the skip list reduces the amount of work done traversing the list, a large amount of extra work may be incurred due to processes attempting to modify the same portion of the list. In the worst case this extra work may be $O(p \log n)$.

4.2 Binary Search Trees

Binary search trees can also be implemented by adapting the techniques of Section 3. Each cell in the tree has a left and right auxiliary node between itself and its subtrees (these auxiliary nodes are present even if the subtree is empty). Thus, searching for a cell with a given key in the binary search tree is almost identical to the algorithm for the standard sequential binary search tree.

Since the insertion of new cells occurs only at the leaves of the tree, adding new cells to the tree is fairly straightforward, involving simply swinging the pointer in the auxiliary node at the leaf. The remainder of this section will deal with the deletion of cells from the tree.

To delete cells with at most one child, we must first insure that the cell will not gain a second child during deletion. To do this we first merge the subtrees by swinging the auxiliary node pointer preceding the empty child to point at the auxiliary node preceding the child to be deleted. Thus we effectively "short circuit" any processes traversing the tree from proceeding down that branch of the tree, shunting them to the other branch instead. We can then splice out the cell to be deleted and remove extra auxiliary nodes using techniques similar to those in Section 3.

If a cell has two children, we must move one of the subtrees first. Note that we cannot move any cell closer to the root, since this could result in concurrent processes being unable to find its key while traversing the tree, resulting in non-linearizable behavior. Instead, we can move one of the subtrees of the cell being deleted down in the tree; e.g., making its left subtree the left child of its in-order successor.

Figure 14 illustrates how this could be done; first we find the in-order successor (node G) of the node to be deleted (node F). We then swing the auxiliary node preceding its (empty) left child to point at the left subtree of the cell to be deleted. We can then remove the cell and extra auxiliary nodes with three more steps, as indicated in the figure.

The effect of this deletion method on the performance of the binary search tree is unknown. If we consider only FIND and INSERT dictionary operations, then the amount of extra work done by a sequence of operations is expected to be $O(n \log n)$, since the tree has expected height $O(\log n)$ and any cell that is inserted can only have been retried once for every cell on the path back to the root.

[2] k is a parameter generally chosen to be $\Theta(\log N)$, where N is the number of items expected to be in the skip list.

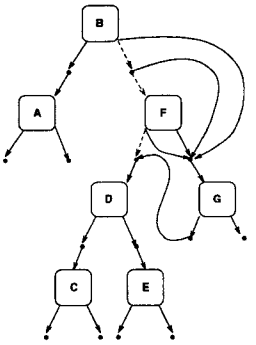

Figure 14: Deletion of cell with two children.

5 Memory Management

We have thus far assumed that new cells could be allocated whenever necessary, and that deleted cells could be left intact for cursors to continue traversing them. In addition, we claimed in Section 2.1 that our solution to the ABA problem relied on careful memory management. In this section we address these issues.

5.1 The ABA Problem

We have used the COMPARE&SWAP primitive in our algorithms to atomically swing pointers from their current value to a new one. However, using COMPARE&SWAP in this manner is susceptible to the following problem known as the ABA problem. When swinging a pointer, we do not want the pointer to change if its value has changed from when it was read. This problem occurs when the pointer has changed, but then subsequently changes back to its original value. In this case, the COMPARE&SWAP primitive will successfully change the value of the pointer, possibly corrupting the data structure.

There are several ways to avoid this problem. One commonly used approach makes use of a double-word version of the COMPARE&SWAP operation. The idea is to attach a *tag* value to each pointer; every time the pointer is changed, the tag is incremented (the double-word COMPARE&SWAP is used to change both the pointer and tag values simultaneously). Thus, even if the pointer changes back to a previous value, the tag value will most likely be different and the COMPARE&SWAP operation will fail. Unfortunately, this double-word version of COMPARE&SWAP is not available on most architectures.

Another approach is to use a stronger primitive. For example, on on architectures such as the DEC Alpha, the LOAD-LOCKED operation can be used to read a pointer, and the STORE-CONDITIONAL operation can be used to swing it. Unlike COMPARE&SWAP, the STORE-CONDITIONAL primitive will change the pointer only if it has not changed, and it is not susceptible to the ABA problem.

Although the LOAD-LOCKED and STORE-CONDITIONAL primitives are found on a fair number of newer architectures, this technique suffers from the fact that these primitives are implemented with certain restrictions; for example, it is generally not possible to read from memory between a LOAD-LOCKED and a STORE-CONDITIONAL (cf. [25]). This restriction makes it impossible to implement our algorithms using these primitives.

The approach we take in this paper makes use of the observation that in the normal operation of the algorithms given in the previous sections, a pointer is never changed back to a previous value. The only way for a pointer to take on a previous value is for cells to be reused after they have been deleted from the data structure. If we prohibit this reuse, then we may use the COMPARE&SWAP primitive without worrying about the ABA problem.

In most applications, it is probably not realistic to assume that cells will not be reused. However, we make the further observation that the ABA problem can only occur if a cell is reused while another process has a pointer to it. Thus, we can safely reuse cells, avoiding the ABA problem, as long as we can guarantee that no other processes have pointers to the cell.

We accomplish this through the use of reference counts; each cell has a field *refct* and another field *claim* (described below). These reference counts are manipulated through the SAFEREAD and RELEASE operations used in the algorithms. Note that the problem of cell persistence is also solved by the use of these reference counts, as cells that can no longer be accessed from the list or through cursors are available for reuse.

The SAFEREAD operation atomically reads a pointer and increments the reference count in the cell being pointed at. The RELEASE operation decrements the reference count and reclaims the cell for reuse, if there are not other pointers to the cell. Figures 15 and 16 give algorithms for these operations.

```
SafeRead(p : pointer)
  returns pointer

1  loop:  q ← Read(p)
2         if q = NULL then
3           return NULL
4         Increment(q^.refct)
5         if q = Read(p) then
6           return q
          else
7           Release(q)
8         goto loop
```

Figure 15: Algorithm for the SAFEREAD operation.

```
Release(p : pointer)

1   c ← Fetch&Add(p^.refct, −1)
2   if c > 1 then
3     return
4   c ← Test&Set(p^.claim)
5   if c = 1 then
6     return
    else
7     Reclaim(p)
```

Figure 16: Algorithm for the RELEASE operation.

Note that care must be taken in the RELEASE algorithm, as it is possible for more than one processes to concurrently

see the reference count go to zero in the same cell. The *claim* field in the cell is used to ensure that only one process will actually try to reclaim the cell for reuse.

5.2 Managing Free Cells

In addition to the SAFEREAD and RELEASE operations, we need to be able to allocate and reclaim cells. One way of solving this problem is with another concurrent object, which acts as a set containing free cells that may be allocated to processes. This object provides two operations: ALLOC removes a free cell from the set and returns it to be used by a process, and RECLAIM returns a cell no longer being used to the set of free cells.

For brevity, we describe only a very simple implementation of this object, in which free cells must all be of the same size. are kept on a simple list. Much more elaborate schemes are possible; in particular, in [28] we show how to extend these ideas to implement a lock-free buddy system which provides management of variable-sized cells.

We keep cells which are not in use on a free list. New cells are allocated by removing them from the front of the list, and cells are reclaimed by putting them back on the front (i.e., the list acts as a stack). Figures 17 and 18 give algorithms for the ALLOC and RECLAIM operations.

```
ALLOC()
  returns pointer

      repeat
1       q ← SAFEREAD(Freelist)
2       if q = NULL
3         return NULL
4       r ← CSW(Freelist, q, q^.next)
5       if r = FALSE
6         RELEASE(q)
7     until r = TRUE
8     WRITE(q^.claim, 0)
9     return q
```

Figure 17: Algorithm for the ALLOC operation.

The variable *Freelist* is a pointer to the first free cell on the list. Note that the ALLOC algorithm must make use of the SAFEREAD and RELEASE operations in order to avoid the ABA problem.

```
RECLAIM(p : pointer)

      repeat
1       q ← Freelist
2       WRITE(p^.next, q)
3       r ← CSW(Freelist, q, p)
4     until r = TRUE
```

Figure 18: Algorithm for the RECLAIM operation.

6 Conclusion

We have presented algorithms using COMPARE&SWAP for manipulating a singly-linked list with concurrent processes without the use of mutual exclusion. This includes traversing the list using cursors, insertion and deletion of nodes at any point in the list, and memory management. We have shown how these techniques can be used as building blocks for other types of concurrent objects, such as the dictionary abstract data type. All of our algorithms have the property that they are non-blocking.

We chose COMPARE&SWAP as our synchronization primitive for several reasons. Not only is it is universal, in the sense that it is powerful enough to implement a non-blocking linked list, but it is also commonly available on a number of architectures.

We expect the performance of our algorithms to be competitive with similar data structures that use spin locks. The most time consuming operation is most likely performing a SAFEREAD on each cell as we traverse the list; it would be useful to have this operation implemented in hardware.

Preliminary performance analysis of these algorithms can be found in [28]; however, more work remains to be done in order to quantitatively determine the performance tradeoffs between algorithms such as these and more traditional methods using mutual exclusion. We are currently examining the performance of these algorithms and data structures experimentally.

References

[1] J. Alemany and E. W. Felten. Performance issues in non-blocking synchronization on shared-memory multiprocessors. In *Proceedings of the Eleventh Symposium on Principles of Distributed Computing*, pages 125–134, 1992.

[2] R. J. Anderson and H. Woll. Wait-free parallel algorithms for the union-find problem. In *Proceedings of the Twenty-Third ACM Symposium on Theory of Computing*, pages 370–380, 1991.

[3] T. Anderson. *Operating System Support for High Performance Multiprocessing.* PhD thesis, University of Washington, Department of Computer Science and Engineering, Seattle, WA, 1991. University of Washington Department of Computer Science and Engineering Technical Report 91-08-10.

[4] G. Barnes. A method for implementing lock-free shared data structures. In *Proceedings of the Fifth Symposium on Parallel Algorithms and Architectures*, pages 261–270, 1993.

[5] B. N. Bershad, D. D. Redell, and J. R. Ellis. Fast mutual exclusion for uniprocessors. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 223–233, 1992.

[6] W. B. Easton. Process synchronization without long-term interlock. In *Proceedings of the Third Symposium on Operating Systems Principles*, pages 95–100, 1972.

[7] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, pages 374–382, 1985.

[8] G. Graunke and S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. *Computer*, 23:60–69, June 1990.

[9] D. B. Gustavson. The Scalable Coherent Interface and related standards projects. *IEEE Micro*, pages 10–22, February 1922.

[10] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1):124–149, January 1991.

[11] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15:745–770, November 1993.

[12] M. Herlihy and J. Moss. Lock-free garbage collection for multiprocessors. In *Proceedings of the Third Symposium on Parallel Algorithms and Architectures*, pages 229–236, July 1991.

[13] M. P. Herlihy. Impossibility and universality results for wait-free synchronization. In *Proceedings of the Seventh Symposium on Principles of Distributed Computing*, pages 276–290, 1988.

[14] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[15] Q. Huang and W. E. Weihl. An evaluation of concurrent priority queue algorithms. In *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, pages 518–525, 1991.

[16] S. Hummel and E. Schonberg. Low-overhead scheduling of nested parallelism. *IBM Journal Of Research And Development*, 35:743–765, Sep 1991.

[17] L. Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, 1977.

[18] V. Lanin and D. Shasha. Concurrent set manipulation without locking. In *Proceedings of the Seventh Symposium on Principles of Database Systems*, pages 211–220, March 1988.

[19] H. Massalin and C. Pu. A lock-free multiprocessor OS kernel. Technical Report CUCS–005–91, Columbia University, 1991.

[20] J. Mellor-Crummey and M. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[21] M. M. Michael and M. L. Scott. Implementation of general-purpose atomic primitives for distributed shared-memory multiprocessors. In *First International Symposium on High Performance Computer Architecture*, January 1995. Also Univ. of Rochester Computer Science Dept. TR 528.

[22] S. A. Plotkin. Sticky bits and universality of consensus. In *Proceedings of the Eighth Symposium on Principles of Distributed Computing*, pages 159–175, 1989.

[23] W. Pugh. Concurrent maintenance of skip lists. Technical Report CS-TR-2222.1, Institute for Advanced Computer Studies, Department of Computer Science, University of Maryland, College Park, 1989.

[24] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, June 1990.

[25] R. L. Sites, editor. *Alpha Architecture Reference Manual.* Digital Press, Burlington, MA, 1992.

[26] J. Turek, D. Shasha, and S. Prakash. Locking without blocking: Making lock based concurrent data structure algorithms nonblocking. In *Proceedings of the Eleventh Symposium on Principles of Database Systems*, 1992.

[27] J. D. Valois. Implementing lock-free queues. In *Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems*, pages 64–69, Las Vegas, NV, October 1994. Available as RPI Dept. of Comp. Sci. Tech. Report #94–17.

[28] J. D. Valois. *Lock-Free Data Structures.* PhD thesis, Rensselaer Polytechnic Institute, Department of Computer Science, 1995.