

Lock Free Programming Practice



@睡眼惺忪的小叶先森 编

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, multi-tasking 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 primitive¹, 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 single-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

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

```

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

  BEGIN ATOMIC
  1  if  $a^{\wedge} \neq old$ 
  2     $a^{\wedge} \leftarrow new$ 
  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 an 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.

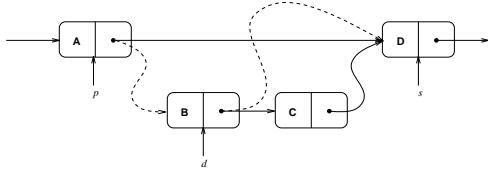


Figure 2: Deletion of B concurrent with insertion of C.

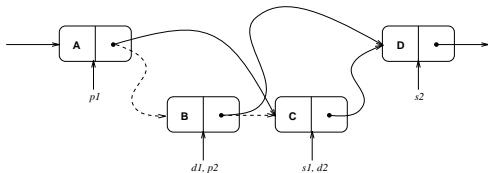


Figure 3: Concurrent deletion of B and C; second is undone.

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).

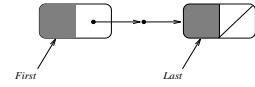


Figure 4: An empty linked list, with two dummy nodes and an auxiliary node.

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 $c^.pre_aux = c^.target$, 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    $c^{\wedge}.\text{pre\_cell} \leftarrow \text{SAFEREAD}(First)$ 
2    $c^{\wedge}.\text{pre\_aux} \leftarrow \text{SAFEREAD}(First^{\wedge}.\text{next})$ 
3    $c^{\wedge}.\text{target} \leftarrow \text{NULL}$ 
4   UPDATE( $c$ )

```

Figure 6: The FIRST algorithm.

```

NEXT( $c$  : cursor)
returns boolean

1   if  $c^{\wedge}.\text{target} = \text{Last}$ 
2       return FALSE
3   RELEASE( $c^{\wedge}.\text{pre\_cell}$ )
4    $c^{\wedge}.\text{pre\_cell} \leftarrow \text{SAFEREAD}(c^{\wedge}.\text{target})$ 
5   RELEASE( $c^{\wedge}.\text{pre\_aux}$ )
6    $c^{\wedge}.\text{pre\_aux} \leftarrow \text{SAFEREAD}(c^{\wedge}.\text{target}^{\wedge}.\text{next})$ 
7   UPDATE( $c$ )
8   return TRUE

```

Figure 7: The NEXT algorithm.

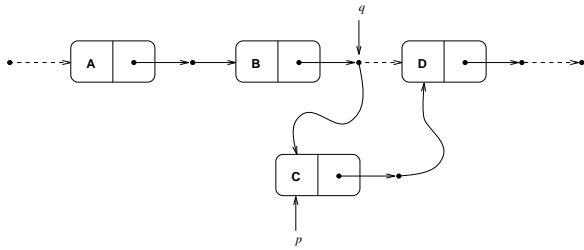


Figure 8: Inserting a new cell and auxiliary node.

```

TRYINSERT( $c$  : cursor,  $q$  : cell $^{\wedge}$ ,  $a$  : aux. node $^{\wedge}$ )
returns boolean

```

```

1   WRITE( $q^{\wedge}.\text{next}, a$ )
2   WRITE( $a^{\wedge}.\text{next}, c^{\wedge}.\text{target}$ )
3    $r \leftarrow \text{CSW}(c^{\wedge}.\text{pre\_aux}, c^{\wedge}.\text{target}, q)$ 
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 occurred 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 occurred 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    $d \leftarrow c^\wedge.\text{target}$ 
2    $n \leftarrow c^\wedge.\text{target}^\wedge.\text{next}$ 
3    $r \leftarrow \text{CSW}(c^\wedge.\text{pre\_aux}^\wedge.\text{next}, d, n)$ 
4   if  $r \neq \text{TRUE}$ 
5     return FALSE
6    $\text{WRITE}(d^\wedge.\text{back\_link}, c^\wedge.\text{pre\_cell})$ 
7    $p \leftarrow c^\wedge.\text{pre\_cell}$ 
8   while  $p^\wedge.\text{back\_link} \neq \text{NULL}$ 
9      $q \leftarrow \text{SAFEREAD}(p^\wedge.\text{back\_link})$ 
10     $\text{RELEASE}(p)$ 
11     $p \leftarrow q$ 
12     $s \leftarrow \text{SAFEREAD}(p^\wedge.\text{next})$ 
13    while  $n^\wedge.\text{next}^\wedge$  is not a normal cell
14       $q \leftarrow \text{SAFEREAD}(n^\wedge.\text{next})$ 
15       $\text{RELEASE}(n)$ 
16       $n \leftarrow q$ 
17    repeat
18       $r \leftarrow \text{CSW}(p^\wedge.\text{next}, s, n)$ 
19      if  $r = \text{FALSE}$ 
20         $\text{RELEASE}(s)$ 
21       $s \leftarrow \text{SAFEREAD}(p^\wedge.\text{next})$ 
22    until  $r = \text{TRUE}$ 
23    or  $p^\wedge.\text{back\_link} \neq \text{NULL}$ 
24    or  $n^\wedge.\text{next}$  is not a normal cell
25   $\text{RELEASE}(p)$ 
26   $\text{RELEASE}(s)$ 
27   $\text{RELEASE}(n)$ 
return TRUE

```

Figure 10: The TRYDELETE algorithm.

```

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

1   while  $c^\wedge.\text{target} \neq \text{Last}$ 
2     if  $c^\wedge.\text{target}^\wedge.\text{key} = k$ 
3       return TRUE
4     else if  $c^\wedge.\text{target}^\wedge.\text{key} > k$ 
5       return FALSE
6     else
7        $\text{NEXT}(c)$ 
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 by 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       $q \leftarrow \text{new cell}$ 
3       $a \leftarrow \text{new aux. node}$ 
4      initialize other fields of  $q$ 
loop:
5       $r \leftarrow \text{FINDFROM}(k, c)$ 
6      if  $r = \text{TRUE}$ 
7          return
8       $r \leftarrow \text{TRYINSERT}(c, q, a)$ 
9      if  $r = \text{TRUE}$ 
10         return
11     UPDATE( $c$ )
12     goto loop

```

Figure 12: The INSERT algorithm.

```

DELETE( $k$  : key)

1      FIRST( $c$ )
loop:
2       $r \leftarrow \text{FINDFROM}(k, c)$ 
3      if  $r = \text{FALSE}$ 
4          return
5       $r \leftarrow \text{TRYDELETE}(c)$ 
6      if  $r = \text{TRUE}$ 
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 lists², 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.

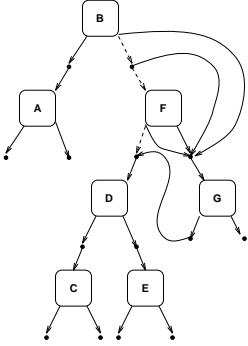


Figure 14: Deletion of cell with two children.

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 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:  $q \leftarrow \text{READ}(p)$ 
    2     if  $q = \text{NULL}$  then
    3         return NULL
    4     INCREMENT( $q^.refct$ )
    5     if  $q = \text{READ}(p)$  then
    6         return  $q$ 
    7     else
    8         RELEASE( $q$ )
    goto loop

```

Figure 15: Algorithm for the SAFEREAD operation.

```

RELEASE( $p$  : pointer)
    1  $c \leftarrow \text{FETCH\&ADD}(p^.refct, -1)$ 
    2 if  $c > 1$  then
    3     return
    4  $c \leftarrow \text{TEST\&SET}(p^.claim)$ 
    5 if  $c = 1$  then
    6     return
    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 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 trade-offs 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 1992.
- [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.

Correction of a Memory Management Method for Lock-Free Data Structures *

Maged M. Michael Michael L. Scott

Department of Computer Science

University of Rochester

Rochester, NY 14627-0226

{michael, scott}@cs.rochester.edu

December 1995

Abstract

Memory reuse in link-based lock-free data structures requires special care. Many lock-free algorithms require deleted nodes not to be reused until no active pointers point to them. Also, most lock-free algorithms use the `compare_and_swap` atomic primitive, which can suffer from the “ABA problem” [1] associated with memory reuse. Valois [3] proposed a memory management method for link-based data structures that addresses these problems. The method associates a reference count with each node of reusable memory. A node is reused only when no processes or data structures point to it. The method solves the ABA problem for acyclic link-based data structures, and allows lock-free algorithms more flexibility as nodes are not required to be freed immediately after a delete operation (e.g. dequeue, pop, delete min, etc.). However, there are race conditions that may corrupt data structure that use this method. In this report we correct these race conditions and present a corrected version of Valois’s method.

Keywords: concurrency, lock-free, non-blocking, memory management, `compare_and_swap`.

*This work was supported in part by NSF grants nos. CDA-94-01142 and CCR-93-19445, and by ONR research grant no. N00014-92-J-1801 (in conjunction with the DARPA Research in Information Science and Technology—High Performance Computing, Software Science and Technology program, ARPA Order no. 8930).

1 Introduction

On shared memory multiprocessor systems, processes communicate by concurrently updating shared data structures. To ensure the consistency of these data structures, processes have to synchronize their access to them. Mutual exclusion locks are the most widely used technique for ensuring the consistency of concurrent data structures. However mutual exclusion locks suffer from significant performance degradation on multiprogrammed and asynchronous systems, as a slow process can delay faster processes [5].

Motivated by these problems, many lock-free methodologies and algorithms have been developed. Most of these algorithm use the atomic primitive `compare_and_swap` and have to deal with the “ABA problem” [1], which occurs if a process reads a value A in a shared location, computes a new value, and then attempts a `compare_and_swap` operation. The `compare_and_swap` may succeed when it should not, if between the read and the `compare_and_swap` some other process(es) change the A to a B and then back to an A again. The most common solution is to associate a modification counter with a pointer, to always access the counter with the pointer in any read-modify-`compare_and_swap` sequence, and to increment it in each successful `compare_and_swap`. This solution does not guarantee that the ABA problem will not occur, but makes it extremely unlikely. To implement this solution, one must either employ a double-word `compare_and_swap`, or else use array indices instead of pointers, so that they may share a single word with a counter.

Valois [3], in his Ph.D. thesis on lock-free data structures, proposes an alternative solution to the ABA problem, which *guarantees* that this problem will not occur, without the need for modification counters or the double-word `compare_and_swap`. Valois’s solution relies on associating a reference count with each node. A node is only reused if no private process pointers or shared pointers point to it. Like most reference count mechanisms, the method is usable only with acyclic structures, as it is vulnerable to memory leakage with circular structures. Valois presents algorithms for non-blocking queues [2] and linked lists [4] that do not allow immediate memory reuse of deleted nodes. They need to be used with the associated memory management method.

We discovered race conditions in the memory management method and its application to lock-free algorithms. The races may cause active nodes to be incorrectly reused, thereby corrupting the lock-free data structure. In the remainder of this report we present a corrected version of Valois’s memory management method for lock-free data structures.

2 Memory Management Method

In this section we present an overview of Valois’s memory management method for lock-free data structures, the race conditions that we discovered, and a corrected version of the method.

Valois’s memory management method basically relies on a reference count associated with each reusable memory node, to determine whether it is safe or not to reuse the node. A node can be reused only if there are no pointers that point to it in the data structure or in private process variables.

The method uses four basic routines: NEW, RECLAIM, SAFEREAD, and RELEASE. NEW allocates a node from a free list and initializes its *refct* and *claim* fields. RECLAIM frees a deleted node when it is ready to be reused. SAFEREAD reads a pointer to a node and increments the node’s reference count. RELEASE decrements the reference count of a node and determines whether it can be freed safely or not. Figure 1 presents (semantically equivalent) simplified pseudocode for these routines based on the algorithms in Valois’s dissertation.

Valois [3] present a lock-free algorithm for concurrent queues that uses his memory mangement method. Figure 2 presents pseudocode for the DEQUEUE operation (Valois provided the authors a modified version in private correspondence, as the version in the dissertation contains typographical errors).

The algorithms contain two race conditions. One is in the RELEASE operation, and the other is in the DEQUEUE operation. They are shown in figures 3 and 4, respectively.

The first race condition arises from the timing window between decrementing *refct* of the released node, and the test-and-set operation on the *claim* bit of the same node. A node can be reclaimed twice as shown in figure 3. The solution is to perform the decrement and test-and-set operations together atomically.

The second race condition arises from allowing a shared pointer to point to a node *before* incrementing its *refct* field. Thus, an active node in the data structure can be freed incorrectly. The solution is to increment the *refct* field of a node *before* any operation that might result in that a shared pointer points to that node. If the operation fails and the shared pointer does not point to the node in question, then the node has to be released.

Figure 5 presents a corrected version of the DEQUEUE operation. Figure 6 presents a corrected version of Valois's memory management method. The RECLAIM operation is the same as in figure 1.

3 Conclusions

In this report we present corrections to Valois's memory management method for link-based lock-free data structures. However, the memory management mechanism remains impractical: no finite memory can guarantee to satisfy the memory requirements of the method all the time. Problems occur if a process reads a pointer to a node (incrementing the node's reference counter) and is then delayed. While it is not running, other processes can insert and delete an arbitrary number of additional nodes. Because of the pointer held by the delayed process, neither the node referenced by that pointer nor any of its successors can be freed. It is therefore possible to run out of memory even if the number of items in the data structure is bounded by a constant. In experiments with a queue of maximum length 12 items, we ran out of memory several times during runs of ten million enqueues and dequeues, using a free list initialized with 64,000 nodes. We hope that this report will help other researchers develop practical memory management methods based on the ideas in Valois's method.

References

- [1] *System/370 Principles of Operation*. IBM Corporation, 1983.
- [2] J. D. Valois. Implementing Lock-Free Queues. In *Seventh International Conference on Parallel and Distributed Computing Systems*, Las Vegas, NV, October 1994.
- [3] J. D. Valois. Lock-Free Data Structures. Ph. D. dissertation, Rensselaer Polytechnic Institute, May 1995.
- [4] J. D. Valois. Lock-free Linked Lists using Compare-and-swap. In *Proceedings of the Fourteenth ACM Symposium on Principles of Distributed Computing*, Ottawa, Ontario, Canada, August 1995.
- [5] J. Zahorjan, E. D. Lazowska, and D. L. Eager. The Effect of Scheduling Discipline on Spin Overhead in Shared Memory Parallel Systems. *IEEE Transactions on Parallel and Distributed Systems*, 2(2):180–198, April 1991.

node_type:

data	application dependent
links	zero or more link fields point to other nodes
refct	reference count
claim	one bit set if node is free

NEW()

- 1 **loop**
- 2 $p \leftarrow \text{SAFEREAD}(\&\text{Freelist})$
- 3 **if** $p = \text{NULL}$
- 4 **error** out of memory
- 5 **if** $\text{CAS}(\&\text{Freelist}, p, p^.next) = \text{TRUE}$
- 6 $p^.claim \leftarrow 0$
- 7 **return** p
- 8 **else**
- 9 $\text{RELEASE}(p)$

RECLAIM(p)

- 1 **repeat**
- 2 $q \leftarrow \text{Freelist}$
- 3 $p^.next \leftarrow q$
- 4 **until** $\text{CAS}(\&\text{Freelist}, q, p) = \text{TRUE}$

SAFEREAD(p)

- 1 **loop**
- 2 $q \leftarrow p^$
- 3 **if** $q = \text{NULL}$
- 4 **return** NULL
- 5 $\text{INCREMENT}(\&q^.refct)$
- 6 **if** $q = p^$
- 7 **return** q
- 8 **else**
- 9 $\text{RELEASE}(q)$

RELEASE(p)

- 1 **if** $p = \text{NULL}$
- 2 **return**
- 3 **if** $\text{FETCHANDADD}(\&p^.refct, -1) > 1$
- 4 **return**
- 5 **if** $\text{TESTANDSET}(\&p^.claim) = 1$
- 6 **return**
- 7 **for** all link fields q in $p^$
- 8 $\text{RELEASE}(q)$
- 9 **RECLAIM**(p)

Figure 1: The basic data structures and operations of Valois's memory management method.

```

DEQUEUE()
1 repeat
2     p  $\leftarrow$  SAFERead(&Head)
3     if p^.next = NULL
4         RELEASE(p)
5         return empty queue
6     r  $\leftarrow$  CAS(&Head, p, p^.next)
7     if r = FALSE
8         RELEASE(p)
9 until r = TRUE
10 INCREMENT(&p^.next^.refct)
11 v  $\leftarrow$  p^.next^.value
12 RELEASE(p)
13 RELEASE(p)
14 return v

```

Figure 2: Dequeue operation with race condition.

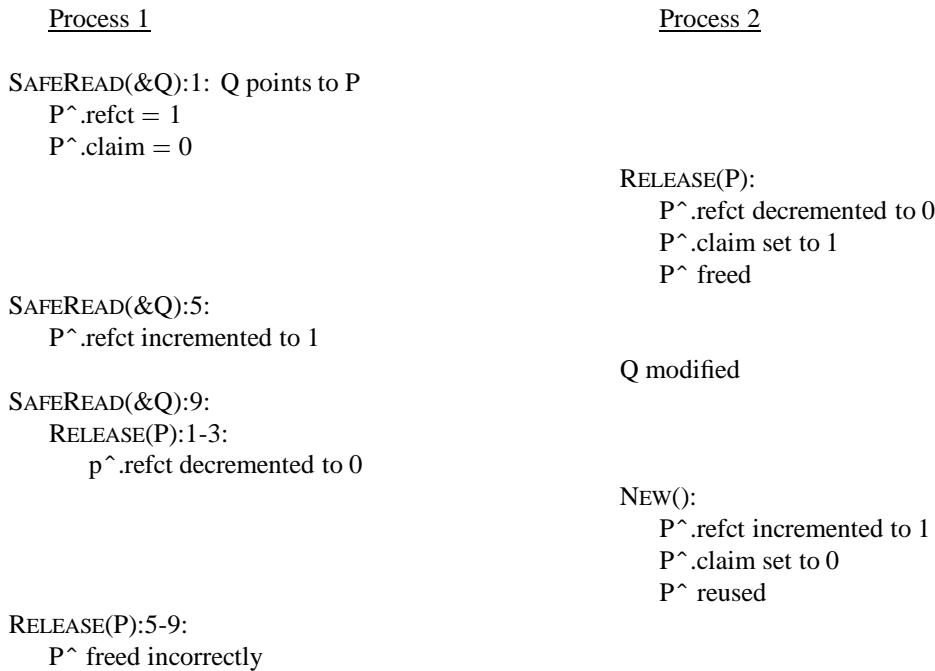


Figure 3: Race condition in RELEASE.

Process 1

DEQUEUE:6:
 CAS succeeds Head points to $P^$
 $P^.\text{refct} = 2$

Process 2

DEQUEUE:6:
 CAS succeeds Head points to $P^.\text{next}^$
 DEQUEUE:12-13:
 $P^.\text{refct}$ decremented to 0
 $P^.\text{claim}$ set to 1
 $P^$ freed incorrectly

Figure 4: Race condition in DEQUEUE.

```

DEQUEUE()
1 repeat
2   p  $\leftarrow$  SAFERREAD(&Head)
3   next  $\leftarrow$   $p^.\text{next}$ 
4   if next = NULL
5     RELEASE(p)
6   return empty queue
7   ATOMICADD(&next $^.\text{refct\_claim}$ , 2)
8   r  $\leftarrow$  CAS(&Head, p, next)
9   if r = FALSE
10    RELEASE(next)
11    RELEASE(p)
12 until r = TRUE
13 v  $\leftarrow$   $p^.\text{next}^.\text{value}$ 
14 RELEASE(p)
15 return v

```

Figure 5: Corrected dequeue operation.

node_type:

data	application dependent
links	zero or more link fields point to other nodes
refct_claim	combined reference count and claim bit

```
DECREMENTANDTESTANDSET(ptr)
1 repeat
2   old ← ptr^
3   new ← old - 2
4   if new = 0
5     new ← 1
6   until CAS(ptr, old, new) = TRUE
7   return (old - new) AND 1
```

```
CLEARLOWESTBIT(ptr)
1 repeat
2   old ← ptr^
3   new ← old - 1
4   until CAS(ptr, old, new) = TRUE
```

```
NEW()
1 loop
2   p ← SAFEREAD(&Freelist)
3   if p = NULL
4     error out of memory
5   if CAS(&Freelist, p, p^.next) = TRUE
6     CLEARLOWESTBIT(&p^.refct_claim)
7     return p
8   else
9     RELEASE(p)
```

```
SAFEREAD(p)
1 loop
2   q ← p^
3   if q = NULL
4     return NULL
5   ATOMICADD(&q^.refct_claim, 2)
6   if q = p^
7     return q
8   else
9     RELEASE(q)
```

```
RELEASE(p)
1 if p = NULL
2   return
3 if DECREMENTANDTESTANDSET(&p^.refct_claim) = 0
4   return
5 for all link fields q in p^
6   RELEASE(q)
7 RECLAIM(p)
```

Figure 6: Corrected basic operations for Valois's memory management method.

A Pragmatic Implementation of Non-Blocking Linked-Lists

Timothy L. Harris

University of Cambridge Computer Laboratory,
Cambridge, UK, tim.harris@cl.cam.ac.uk

Abstract. We present a new non-blocking implementation of concurrent linked-lists supporting linearizable insertion and deletion operations. The new algorithm provides substantial benefits over previous schemes: it is conceptually simpler and our prototype operates substantially faster.

1 Introduction

It is becoming evident that non-blocking algorithms can deliver significant benefits to parallel systems [MP91,LaM94,GC96,ABP98,Gre99]. Such algorithms use low-level atomic primitives such as compare-and-swap – through careful design and by eschewing the use of locks it is possible to build systems which scale to highly-parallel environments and which are resilient to scheduling decisions.

Linked-lists are one of the most basic data structures used in program design, and so a simple and effective non-blocking linked-list implementation could serve as the basis for many data structures. This paper presents a novel implementation of linked-lists which is non-blocking, linearizable and which is based on the compare-and-swap (CAS) operation found on contemporary processors.

Section 5 sketches a proof of correctness, describes the use of model-checking to perform exhaustive verification within a limited application domain and also describes empirical tests performed on execution traces from an actual implementation.

In Sect. 6 we compare the performance of the new algorithm against that of a lock-based implementation and against an existing non-blocking algorithm. Compared with these other thread-safe algorithms, ours provides the best performance on each of three simulated workloads and for every level of concurrency.

2 Overview

In this section we present an overview of our algorithm and the difficulty in implementing non-blocking linked-lists. As a running example consider an ordered list containing the integers 10 and 30 along with sentinel *head* and *tail* nodes:



Such a data structure may comprise cells containing two fields: a **key** field used to store the element and a **next** field to contain a reference to the next cell in the list.

Insertion is straightforward: a new list cell is created (*below, left*) and then introduced using single CAS operation on the **next** field of the proposed predecessor (*below, right*).



In this case the atomicity of the CAS ensures the the nodes either side of the insertion have remained adjacent. This simple guarantee is insufficient for deletions within the list. Suppose that we wish to remove the value 10. An obvious way of excising this node would be to perform a CAS that swings the reference from the head so that the node containing 30 becomes the first in the list:



Although this CAS ensures that the node 10 was still at the start of the list it cannot ensure that no additional nodes were introduced between the 10 node and the 30 node. If this deletion took place concurrently with the previous insertion then that new node would be lost:



The single CAS could neither detect nor prevent changes between 10 and 30 once the deletion procedure had selected 30. Our proposed solution – and indeed the crux of the algorithms presented here – is to use two separate CAS operations in place of that single one. The first of these is used to *mark* the **next** field of the deleted node in some way (*below, left*), whereas the second is used to excise the node (*below, right*):



We say that a node is *logically deleted* after the first stage and that it is *physically deleted* after the second. A marked field may still be traversed but takes a numerically distinct value from its previous unmarked state; the structure of the list is retained while signalling concurrent insertions to avoid introducing new nodes immediately after those that are logically deleted. In our example the concurrent insertion of 20 would observe 10 to be logically deleted and would attempt to physically delete it before re-trying the insertion.

3 Related Work

Generalized non-blocking implementations based on CAS were presented by Herlihy [Her91,Her93]. However, linked-lists based on this general scheme are highly centralized and suffer poor performance because they essentially use CAS to change a shared global pointer from one version of the structure to the next.

Valois was the first to present an effective CAS-based non-blocking implementation of linked-lists [Val95]. Although highly distributed, his implementation is very involved. The list is held with *auxilliary cells* between adjacent pairs of ordinary cells. Auxilliary exist to provide an extra level of indirection so that a cell may be removed by joining together the auxilliary cells adjacent to it. Valois' algorithm exposes a more general and lower level interface than we do here; he provides explicit *ursors* to identify cells in the list and operations to insert or delete nodes at those points.

The originally-published algorithm contained a number of errors relating to how reference-counted storage was managed. One has been reported previously and others were identified when implementing Valois' algorithm for comparison in this paper [MS95,Val01].

To overcome the complexity of building linearizable lock-free linked-lists using CAS, Greenwald suggested a stronger double-compare-and-swap (DCAS) primitive that atomically updates two storage locations after confirming that they both contain required values [Gre99]. DCAS is not available on today's multi-processor architectures. However, it does admit a simple linearizable linked-list algorithm: insertions proceed as described in Sect. 2 and deletions by atomic updates to the `next` field of the cell being removed as well as that of its predecessor. Greenwald's work was an extension of earlier non-linearizable DCAS-based linked-list algorithms due to Massalin and Pu [MP91].

4 Algorithms

In this section we present our new algorithm in pseudo-code modeled on C++ and designed for execution on a conventional shared-memory multi-processor system supporting *read*, *write* and atomic *compare-and-swap* operations. We assume that the operations defined here are the only means of accessing linked list objects. Each processor executes a sequence of these operations, defining a *history* of invocations/responses and inducing a *real-time* order between them. We say that an operation *A precedes B* if the response to *A* occurs before the invocation of *B* and that operations are *concurrent* if they have no real-time ordering.

A *sequential* history is one in which each invocation is followed immediately by its corresponding response. Our basic correctness requirement is linearizability which requires that (*a*) the responses received in every concurrent history are equivalent to those of some legal sequential history of the same requests and (*b*) the ordering of operations within the sequential history is consistent with the real-time order [HW90]. Linearizability means that operations appear

```

class List<KeyType> {
    Node<KeyType> *head;
    Node<KeyType> *tail;
    List() {
        head = new Node<KeyType> ();
        tail = new Node<KeyType> ();
        head.next = tail;
    }
}

class Node<KeyType> {
    KeyType key;
    Node *next;
    Node (KeyType key) {
        this.key = key;
    }
}

```

Fig. 1. An instance of the List class contains two fields which identify the head and the tail. Instances of Node contain two fields identifying the key and successor of the node.

```

public boolean List::insert (KeyType key) {
    Node *new_node = new Node(key);
    Node *right_node, *left_node;

    do {
        right_node = search (key, &left_node);
        if ((right_node != tail) && (right_node.key == key)) /*T1*/
            return false;
        new_node.next = right_node;
        if (CAS (&(left_node.next), right_node, new_node)) /*C2*/
            return true;
    } while (true); /*B3*/
}

```

Fig. 2. The List::insert method attempts to insert a new node with the supplied key.

to take effect atomically at some point between their invocation and response. Our implementation is additionally *non-blocking*, meaning that some operation will complete in a finite number of steps, even if other operations halt.

We write CAS(addr, o, n) for a CAS operation that atomically compares the contents of `addr` against the old value `o` and – if they match – writes `n` to that location. CAS returns a boolean indicating whether this update took place. Our design was guided by the assumption that a CAS operation is slower to execute than a write which in turn is slower than a read.

4.1 Implementing Sets

Initially we will consider a set object supporting three operations: Insert(k), Delete(k), Find(k). Each parameter k is drawn from a set of totally-ordered keys. The result of an Insert, a Delete or a Find is a boolean indicating success or failure. The set is represented by an instance of `List` which contains a singly-linked list of instances of `Node`. As sketched in Sect. 2 these are held in ascending order with sentinel head and tail nodes.

```

public boolean List::delete (KeyType search_key) {
    Node *right_node, *right_node_next, *left_node;

    do {
        right_node = search (search_key, &left_node);
        if ((right_node == tail) || (right_node.key != search_key)) /*T1*/
            return false;
        right_node_next = right_node.next;
        if (!is_marked_reference(right_node_next))
            if (CAS (&(right_node.next), /*C3*/
                     right_node_next, get_marked_reference (right_node_next)))
                break;
    } while (true); /*B4*/
    if (!CAS (&(left_node.next), right_node, right_node_next)) /*C4*/
        right_node = search (right_node.key, &left_node);
    return true;
}

```

Fig. 3. The `List::delete` method attempts to remove a node containing the supplied key.

```

public boolean List::find (KeyType search_key) {
    Node *right_node, *left_node;

    right_node = search (search_key, &left_node);
    if ((right_node == tail) ||
        (right_node.key != search_key))
        return false;
    else
        return true;
}

```

Fig. 4. The `List::find` method tests whether the list contains a node with the supplied key.

The reference contained in the next field of a node may be in one of two states: marked or unmarked. A node is marked if and only if its next field is marked. Marked references are distinct from normal references but still allow the referred-to node to be determined – for example they may be indicated by an otherwise-unused low-order bit in each reference. Intuitively a marked node is one which should be ignored because some process is deleting it. The function `is_marked_reference(r)` returns `true` if and only if `r` is a marked reference. Similarly `get_marked_reference(r)` and `get_unmarked_reference(r)` convert between marked and unmarked references.

The concurrent implementation comprises four methods (Fig. 2-5). The first three, `List::insert`, `List::delete` and `List::find` implement the Insert, Delete and Find operations respectively. The fourth, `List::search`, is used during each of these operations. It takes a search key and returns references to two nodes called the *left node* and *right node* for that key. The method ensures that these nodes satisfy a number of conditions. Firstly, the key of the left node must be less than the search key and the key of the right node must be greater than

```

private Node *List::search (KeyType search_key, Node **left_node) {
    Node *left_node_next, *right_node;

    search_again:
    do {
        Node *t = head;
        Node *t_next = head.next;

        /* 1: Find left_node and right_node */
        do {
            if (!is_marked_reference(t_next)) {
                (*left_node) = t;
                left_node_next = t_next;
            }
            t = get_unmarked_reference(t_next);
            if (t == tail) break;
            t_next = t.next;
        } while (is_marked_reference(t_next) || (t.key<search_key)); /*B1*/
        right_node = t;

        /* 2: Check nodes are adjacent */
        if (left_node_next == right_node)
            if ((right_node != tail) && is_marked_reference(right_node.next))
                goto search_again; /*G1*/
            else
                return right_node; /*R1*/

        /* 3: Remove one or more marked nodes */
        if (CAS (&(left_node.next), left_node_next, right_node)) /*C1*/
            if ((right_node != tail) && is_marked_reference(right_node.next))
                goto search_again; /*G2*/
            else
                return right_node; /*R2*/
        } while (true); /*B2*/
    }
}

```

Fig. 5. The `List::search` operation finds the left and right nodes for a particular search key.

or equal to the search key. Secondly, both nodes must be unmarked. Finally, the right node must be the immediate successor of the left node. This last condition requires the search operation to remove marked nodes from the list so that the left and right nodes are adjacent. As we will show the `List::search` method is implemented so that these conditions are satisfied concurrently at some point between the method's invocation and its completion.

`List::search` is divided into three sections. The first section iterates along the list to find the first unmarked node with a key greater than or equal to the search key. This is the right node. The left node preliminarily refers to the previous unmarked node that was found. The second stage examines these nodes. If `left_node` is the immediate predecessor of `right_node` then `List::search` returns. Otherwise, the third stage uses a CAS operation to remove marked nodes between `left_node` and `right_node`.

`List::insert` uses `List::search` to locate the pair of nodes between which the new node is to be inserted. The update itself takes place with a single CAS operation (C2) which swings the reference in `left_node.next` from `right_node` to the new node.

`List::delete` uses `List::search` to locate the node to delete and then uses a two-stage process to perform the deletion. Firstly, the node is logically deleted by marking the reference contained in `right_node.next` (C3). Secondly, the node is physically deleted. This may be performed directly (C4) or within a separate invocation of search.

The `List::find` method is shown in Fig. 4. It invokes `List::search` and examines the resulting right node.

5 Correctness

In this section we describe three approaches taken to checking the correctness of the algorithms presented here. Section 5.1 outlines a proof of linearizability and progress, Sect. 5.2 describes the exhaustive testing of some cases through model checking and Sect. 5.3 describes a method we used for examining traces from particular program runs.

5.1 Proof Sketch

We will take a fairly direct approach to outlining the linearizability of the operations by identifying particular instants during their execution at which the complete operation appears to occur atomically.

Conditions Maintained by Search. Our argument relies on the conditions identified in Sect. 4.1 which the implementation of `List::search` guarantees hold at some point during its invocation. For the ordering constraints, note that when right node is initialized the preceding loop ensured that `search_key` \leq `right_node.key`. Similarly `left_node.key < search_key` because otherwise the loop would have terminated earlier.

For the adjacency condition and the mark state of the left node we must separately consider each return path. If `List::search` returns at R1 then the test guarding the return statement ensures that right node was the immediate successor of the left node when the `next` field of that node was read into the local variable `t_next`. The same value of `t_next` is found to be unmarked before initializing `left_node`. If `List::search` returns at R2 then C1 establishes the required conditions.

For the mark state of the right node, observe that both return paths confirm that the right node is unmarked after the point at which the first three conditions must be true. Nodes never become unmarked and so we may deduce that the right node was unmarked at that earlier point.

Linearization points. Let $\text{op}_{i,m}$ be the m^{th} operation performed by processor i and let $d_{i,m}$ be the final real-time at which the `List::search` post-conditions are satisfied during its execution. These $d_{i,m}$ identify the times at which the outcome of the operations become inevitable and we shall take the ordering between them to define the linearized order of $\text{Find}(k)$ operations or unsuccessful updates. For a successful find at $d_{i,m}$ the right node was unmarked and contained the search key. For an unsuccessful insertion it exhibits a node with a matching key. For an unsuccessful deletion or find it exhibits the left and right nodes which, respectively, have keys strictly less-than and strictly greater-than the search key.

Furthermore let $u_{i,m}$ be the real-time at which the update C2 inserts a node or C3 logically deletes a node. We shall take $u_{i,m}$ as the linearization points for such successful updates. In the case of a successful insertion the CAS at $u_{i,m}$ ensures that the left node is still unmarked and that the right node is still its successor. For a successful deletion the CAS at $u_{i,m}$ serves two purposes. Firstly, it ensures that the right node is still unmarked immediately before the update (that is, it has not been logically deleted by a preceding successful deletion). Secondly, the update itself marks the right node and therefore makes the deletion visible to other processors.

Progress. We will show that the concurrent implementation is non-blocking. We will show that each successful insertion causes exactly one update, that each successful deletion causes at most two updates and that unsuccessful operations do not cause any updates.

The CAS instructions C1 and C4 each succeed only by unlinking marked nodes from the list. Therefore the number of times that these CAS instructions succeed is bounded above by the number of nodes that have been marked. Exactly one node is marked during each successful deletion (C3) and therefore at most one update may be performed by C1 or C4 for each successful deletion. The remaining CAS instructions (C2 and C3) occur respectively exactly once on the return paths from successful insertions and deletions.

Since there are no recursive or mutually-recursive method definitions consider each backward branch in turn:

- Each time B1 is taken the local variable `t` is advanced once node down the list. The list is always contains the unmarked tail node and the nodes visited have successively strictly larger keys.
- Each time B2 is taken the CAS at C1 has failed and therefore the value of `left_node.next` \neq `left_node.next`. The value of the field must have been modified since it was read during the loop ending at B1. Modifications are only made by successful CAS instructions and each operation causes at most two successful CAS instructions.
- Each time B3 or B4 is taken the CAS at C2 or C4 has failed. As before, the value held in that location must have been modified since it was read in `List::search` and at most two such updates may occur for each operation.

- Each time G_1 or G_2 is taken then a node which was previously unmarked has been marked by another processor. As before, at most two updates may occur for each operation.

5.2 Model Checking

The dSPIN model checker was used to exhaustively verify the operations for certain problem domains. dSPIN is an extension of the SPIN model checker with adds support for pointers, storage management, function calls and local scopes [Hol97,IS99]. This made it more suitable than SPIN for a natural representation of these algorithms.

The modeled state contains two representations of the set: one comprises a linked list of cells whereas the other is summarized as a bit vector. The linked list is updated as proposed here using atomic `d_step` instructions to implement CAS. The bit vector is checked or updated using further `d_steps` at the proposed linearization points.

The model was parameterized according to the number of concurrent threads, the number of operations that each would attempt and the range of key values that could be used. The two largest configurations we could practicably test were with four threads, each performing one operation with three potential keys and with two threads each performing two operations with four potential keys.

5.3 Practical Testing

The linearizability of the operations has also been tested pragmatically. Although such tests cannot provide the assurances of formal methods they are nonetheless important because they avoid the need to make simplifying assumptions for tractability. In particular, the use of relaxed memory models means that the operations supported by a conventional shared memory machine are not linearizable; a direct implementation is likely to fail without further memory barrier instructions.

It is not generally possible to record actual timestamp values for arbitrary operations within an running process. Instead, we surrounded the code executed at each linearization point with further instructions to record coherent per-processor cycle counts.

The resulting intervals were recorded to an in-memory log which was then replayed sequentially in timestamp order. The results thus obtained were compared with those from the concurrent execution. The replay program contains simple heuristics to deal with overlapping intervals. If these cannot determine a consistent linearized order then the replay program reports unresolved inconsistencies for manual inspection and re-ordering.

6 Results

The algorithm described in Sect. 4 has been implemented in a combination of C and SPARC V9 assembly language. We evaluated its performance on an E450

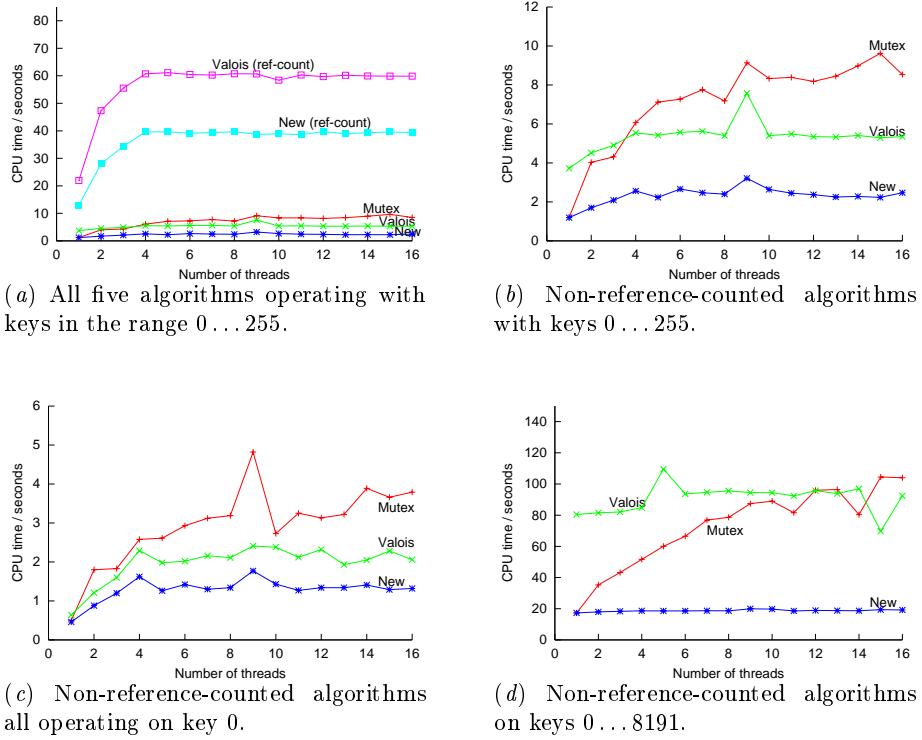


Fig. 6. CPU time (user+system) accounted to the benchmark application for each algorithm on a variety of workloads. In each case the x-axis shows the number of concurrent threads.

server running Solaris 8 and fitted with four 400MHz SPARC V9 processors and 4GB physical memory. It is worth emphasising that the code in Fig. 1-4 is intended merely as pseudo-code and does not reflect an optimised (or even necessarily correct) implementation. Processors may require additional memory barriers – for example between initializing the fields of a new node and introducing it into the list, or between the CAS that logically deletes a node and the CAS that physically deletes it.

The test application compared our implementation against Valois' lock-free algorithm and against a straightforward one in which the list is protected by a mutual exclusion lock¹. Both lock-free algorithms were evaluated with and without reference-counting. All list cells were allocated ahead of time so that the performance of particular memory allocation functions was not included in the

¹ This comparison against a lock-based algorithm is somewhat unfair: the simplified programming model there makes it straightforward to implement a more efficient data structure such as a tree or skip list.

results. The code to manipulate reference-counts is based on Valois' as modified by Michael and Scott with the exception that reference counts are recursively decremented when a cell is freed.

We generated a workload of insertion and deletion operations by randomly choosing keys uniformly distributed within a particular range, selecting equiprobably between insertions and deletions. We used per-thread linear congruential random number generators with the same parameters as the `lrand48` function from the Solaris 8 `libc` library. Seeds were chosen to give non-overlapping series.

The test harness was parameterized on the algorithm to use, the number of concurrent threads to operate and the range of keys that might be inserted or deleted. In each case every thread performed 1 000 000 operations. Figure 6 shows the CPU accounted to the process as a whole for each of the algorithms tested on a variety of workloads.

It is immediately apparent that our algorithm performs notably better for every experiment using more than one thread. In the case of single-threaded execution it outperforms Valois' algorithm in these tests and its performance equals that of the lock-based implementation. The relative performance compared with Valois' algorithm is not surprising: we avoid the need to create, traverse and excise auxiliary nodes.

In addition to the workloads presented in those graphs we also tested configurations with larger ranges of keys, or where the list was initially 'primed' with a long sequence of nodes that would never be deleted. In each case this increased the total number of nodes in the list and thereby added to the cost of retrying operations when CAS instructions fail. One fear was that the lock-free algorithms would start to perform poorly because of the potential for multiple retries. We studied workloads up to lists of 65 536 elements and were unable to find any configuration for which the algorithms based on mutual-exclusion give the best performance. We suspect that although each retry becomes more costly, the likelihood of retries decreases as the rate of conflicting updates falls.

Figure 6a shows the performance of reference-counted implementations. The CPU requirements of Valois' algorithm are degraded by a factor of 5 in the single-threaded case, rising to over 11 for sixteen threads. Similarly, the CPU time required by our algorithms is degraded by a factor of 10 rising to over 15. In each case this is a consequence of need to manipulate reference-counts (using CAS operations) at each stage during a list's traversal. Valois reports that he had originally intended to assume the use of a tracing garbage collector [Val01].

The performance of reference-count manipulation is hampered because the SPARC processor does not provide atomic fetch-and-add. However, measurements taken on a dual-processor Intel x86 machine (with that facility [PPr96]) suggest that the degradation is low when compared with the overall costs seen here. When the reference counts lie on separate lines in the L1 data cache then updates implemented through CAS are 10% slower than those using fetch-and-add. This rises to a factor of 2 degradation when the two processors attempt to update the same address.

Of course, our results are optimistic in that they do not consider the cost of performing GC. However, as Jones and Lins write, if the size of the active data structure is fixed then the cost of copying collectors may be reduced arbitrarily at the expense of the total heap size [JL96]. More practically, they report that overall costs of around 10-20% are typical in modern well-implemented systems.

We examined a further approach to storage reclamation based on the deferred freeing of nodes. In this scheme each node contains an additional field through which it can be linked onto a to-be-freed list when it is excised from the main list. Each thread takes a snapshot of a global timer as its *current time* before starting each operation. Entries are removed from a to-be-freed list when the time of their excision precedes the minimum current time of any thread: at that point no thread can still have a reference to the node held in any of its local variables.

Our implementation allocates a pair of to-be-freed lists for each thread. These are termed the *old list* and the *new list* and are held along with a separate per-thread timer snapshot that is more recent than the excision time of any element of the old list. When the minimum current time exceeds the snapshot then the entire contents of the old list are freed and the elements of the new list are moved to the old list.

This deferred freeing scheme introduces two principal overheads when compared with the used of garbage collection. Firstly, a CAS operation is needed to place nodes on a to-be-freed list – in our implementation this increased the CPU requirements by 15% compared with the results from Fig. 6a operating with 16 threads. The second overhead is the cost of removing elements from the to-be-freed lists and establishing when it is safe to do so. This was a further 1% when performed every 1000 operations and 5% every 100, rising to 52% if performed after every operation.

Figure 7 presents a further analysis of the run-time performance of the three non-reference-counted algorithms, showing the distribution of execution-times for four different kinds of operation. These results were gathered when 8 concurrent threads performing insertions and deletions of keys in the range 0...255. In the case of successful operations the lock-based implementation is able to achieve lower execution times than either lock-free scheme. However, it is also occasionally prone to much longer execution times which explain the higher mean execution time suggested by Fig. 6.

The situation is somewhat different for unsuccessful operations in that both lock-free algorithms obtain some execution times which are lower than those of the lock-based implementation – recall that unsuccessful operations may occur without requiring any CAS operations or other updates to the data structure.

7 Delete Greater-than-or-equal

Now consider the problem of implementing a further operation of the form `DeleteGE(k)` which returns and removes the smallest item that is greater than or equal to k . It is tempting to implement this by modifying `List::delete` so

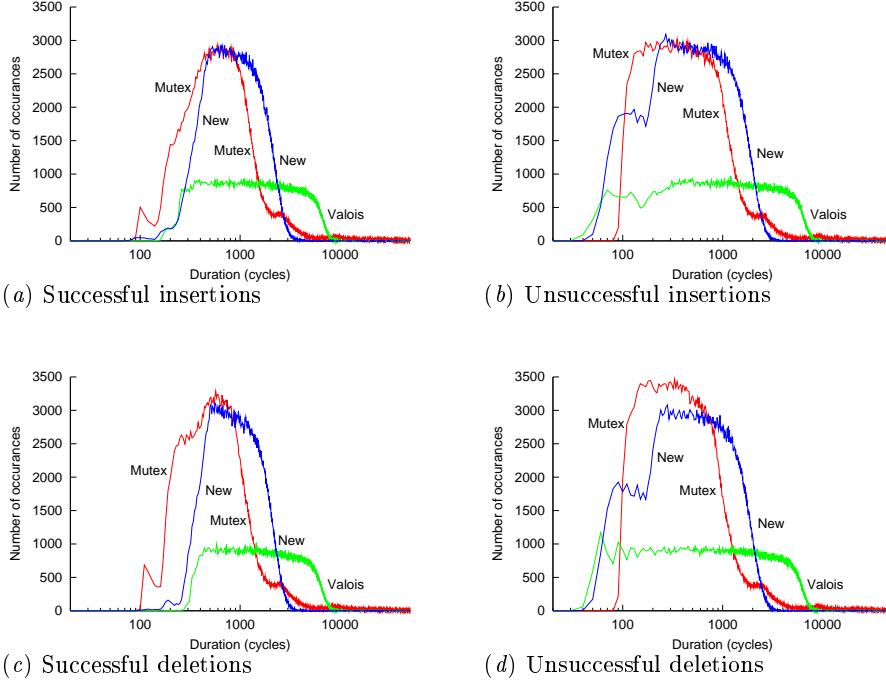


Fig. 7. Operation-time distributions. Each graph shows execution times (in processor cycles) on the x-axis and numbers of occurrences on the y-axis.

that the test T1 does not fail if the key of the right node is greater than the search key.

Unfortunately this implementation is not linearizable. Suppose that three insertion operations are executed in sequence: Insert(20), Insert(15), Insert(20). The first two succeed and the third must fail because 20 is already in the set. However, consider a concurrent DeleteGE(10) operation, attempting to delete any node with a key greater than or equal to 10. Concurrent execution may proceed as follows:

- `List::deleteGE` invokes `List::search` immediately after the first insertion of 20. It takes the head of the list as the left node and the node containing 20 as the right.
- The successful insertion of 15 occurs.
- The unsuccessful insertion of 20 occurs, observing 15 as the key of its left node and 20 as the key of its right node.
- `List::deleteGE(10)` completes after logically deleting the node containing 20.

We must order this `DeleteGE(10)` operation such that its result of 20 would be obtained by a sequential execution. This requires it to be placed before the insertion of 15 because otherwise the key 15 should have been returned in preference to 20. However, we must also linearize the deletion after the failed insertion of 20 because otherwise that insertion would have succeeded. These constraints are irreconcilable.

Intuitively the problem is that at the execution of C3 the right node need not be the immediate successor of the left node. This was acceptable when considering the basic `Delete(k)` operation because we were only concerned with concurrent updates affecting nodes with the same key. Such an update must have marked the right node and so C3 would have failed. In contrast, during the execution of `List::deleteGE`, we must be concerned with updates to any nodes whose keys are greater than or equal to the search key.

We can address this by retaining the implementation of `List::deleteGE` but changing `List::insert` in such a way that C3 must fail whenever a new node may have been inserted between the left and right nodes. This would mean that, whenever C3 succeeds, the key of the right node must still be the smallest key that is greater than or equal to the search key.

This is achieved by using a single CAS operation to (a) introduce a pair of new nodes, one that contains the value being inserted and another that duplicates the right node and (b) mark the original right node:



Such a CAS conceptually has two effects. Firstly, it introduces the new node into the list: beforehand the `next` field of the successor is unmarked and therefore the right node must still be the successor of the left node. Secondly, by marking the contents of that `next` field, the CAS will cause any concurrent `List::deleteGE` with the same right node to fail. Note that the key of the now-marked right node is not in the correct order. However, the existing implementations of `List::search`, `List::delete`, `List::find` and `List::deleteGE` are written so that they do not rely on the correct ordering of marked nodes.

8 Conclusion

This paper has presented a new non-blocking implementation of linked lists. We believe that the algorithms presented here are linearizable. They have also been implemented and we have shown that their measured performance improves both on previously published non-blocking data structures and also on a lock-based implementation.

8.1 Acknowledgments

The work described in this paper was carried out during an internship with the Java Technology Research Group at Sun Labs. The design presented here is

the result of much fruitful discussion with Dave Detlefs, Christine Flood, Alex Garthwaite, Steve Heller, Nir Shavit and Guy Steele. The implementation and evaluation have similarly benefitted from feedback from Mike Burrows, Keir Fraser, Steven Hand, Mark Moir and John Valois.

References

- [ABP98] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 119–129, Puerto Vallarta, Mexico, June 28–July 2, 1998. SIGACT/SIGARCH.
- [GC96] Michael Greenwald and David Cheriton. The synergy between non-blocking synchronization and operating system structure. In USENIX, editor, *2nd Symposium on Operating Systems Design and Implementation (OSDI '96), October 28–31, 1996. Seattle, WA*, pages 123–136, Berkeley, CA, USA, October 1996. USENIX.
- [Gre99] M Greenwald. *Non-blocking synchronization and system design*. PhD thesis, Stanford University, August 1999. Technical report STAN-CS-TR-99-1624.
- [Her91] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.
- [Her93] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.
- [Hol97] Gerard J Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
- [HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.
- [IS99] Radu Iosif and Riccardo Sisto. dSPIN: A dynamic extension of SPIN. In *Proc. of the 6th International SPIN Workshop*, volume 1680 of *LNCS*, pages 261–276. Springer-Verlag, September 1999.
- [JL96] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, July 1996.
- [LaM94] Anthony LaMarca. A performance evaluation of lock-free synchronization protocols. In *Proceedings of the Thirteenth Symposium on Principles of Distributed Computing*, pages 130–140, 1994.
- [MP91] Henry Massalin and Calton Pu. A lock-free multiprocessor OS kernel. Technical Report CU-CS-005-91, Columbia University, 1991.
- [MS95] Maged M. Michael and Michael L. Scott. Correction of a memory management method for lock-free data structures. Technical Report TR599, University of Rochester, Computer Science Department, December 1995.
- [PPr96] *Pentium Pro Family Developer's Manual, volume 2, programmer's reference manual*. Intel Corporation, 1996. Reference number 242691-001.
- [Val95] John D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 214–222, Ottawa, Ontario, Canada, 2–23 August 1995.
- [Val01] John D Valois. Personal communication. March 2001.

Lock-Free Linked Lists and Skip Lists

Mikhail Fomitchev

Department of Computer Science,
York University

Eric Ruppert

Department of Computer Science
York University

ABSTRACT

Lock-free shared data structures implement distributed objects without the use of mutual exclusion, thus providing robustness and reliability. We present a new lock-free implementation of singly-linked lists. We prove that the worst-case amortized cost of the operations on our linked lists is linear in the length of the list plus the contention, which is better than in previous lock-free implementations of this data structure. Our implementation uses backlinks that are set when a node is deleted so that concurrent operations visiting the deleted node can recover. To avoid performance problems that would arise from traversing long chains of backlink pointers, we introduce flag bits, which indicate that a deletion of the next node is underway. We then give a lock-free implementation of a skip list dictionary data structure that uses the new linked list algorithms to implement individual levels. Our algorithms use the single-word C&S synchronization primitive.

Categories and Subject Descriptors

E.1 [Data]: Data Structures—*Distributed Data Structures*; D.1.3 [Software]: Programming Techniques—*Concurrent Programming*; F.2.2 [Theory of Computation]: Analysis of Algorithms and Problem Complexity

General Terms

Algorithms, Performance, Design, Reliability, Theory

Keywords

distributed, fault-tolerant, lock-free, linked list, skip list, efficient, analysis, amortized analysis.

1. INTRODUCTION

A common way to implement shared data structures in distributed systems is to use *mutual exclusion locks*. However, this approach has a major weakness: when one process holds a lock, no other processes can modify the locked

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PODC'04, July 25–28, 2004, St. Johns, Newfoundland, Canada.
Copyright 2004 ACM 1-58113-802-4/04/0007 ...\$5.00.

part. Thus, a delay of one process can cause performance degradation and priority inversion. When halting failures can occur, this becomes particularly important, because the entire system can stop making progress if one process fails. By contrast, an implementation of a shared-memory object is *lock-free* (or *non-blocking*) if a finite number of steps taken by any process guarantees the completion of some operation. If an implementation is lock-free, delays or failures of individual processes do not block the progress of other processes in the system. Lock-free data structures also have the potential to have better performance, because several processes are allowed to modify a data structure at the same time.

Herlihy [4, 5] introduced the first *universal constructions* for designing lock-free data structures using the *Compare&Swap (C&S) synchronization primitive*. Others followed, but they suffer from several flaws, such as inefficiency, low parallelism, excessive copying, and generally high overhead, which often make them impractical. To achieve adequate performance, original algorithms, specific to a particular data structure, are usually required.

Implementing *linked lists* efficiently is very important, as they act as building blocks for many other data structures. We present a new lock-free implementation of a sorted singly-linked list, which handles all dictionary operations with a better average complexity than any prior implementation. Most recent implementations of lock-free linked lists [3, 8] were evaluated only by doing experimental testing. We believe that there exists a certain lack of theoretical development in this area, and our work addresses this problem. A *skip list* [12] is a dictionary data structure, that provides randomized algorithms for searches, insertions, and deletions that run in $O(\log n)$ expected time, where n is the number of elements in the skip-list. The expectation is taken over random choices made by the algorithms. We also give a lock-free implementation of a skip list that is based on using our linked list algorithms to maintain each level of the skip list. Recently, other lock-free skip list designs have been given independently of this work [2, 14, 15].

Our model is an *asynchronous shared-memory* distributed system of several processes, where an arbitrary number of process *halting failures* are allowed. Our algorithms use atomic single-word C&S synchronization primitives. The implementations that we present are *linearizable* [6].

Lock-free implementations allow individual operations to take arbitrarily many steps, so one generally cannot evaluate their worst-case cost. It is natural to analyze the average cost of operations instead, because this evaluates the performance of the system as a whole. To calculate the average

cost of operations in our linked list implementation, we use an amortized analysis that relies on a fairly complex technique of billing part of the cost of each operation S to concurrent operations that slow S down by modifying the data structure. The amortized cost of an operation S , denoted $\hat{t}(S)$, is equal to the actual cost of S plus the total cost billed to S from other operations minus the total cost billed from S to other operations. We measure the cost of operations as a function of the size of the list and the contention. The *point contention* at time T is the number of processes running concurrently at T . We define the *contention of operation S* , denoted $c(S)$, to be the maximum point contention during the execution of S . We prove that $\hat{t}(S) \in O(n(S) + c(S))$, where $n(S)$ is the number of elements in the list when S is invoked and $c(S)$ is the contention of S . The $O(n(S))$ term comes from the cost of traversing the list, while the overhead that comes from concurrency is bounded by $O(c(S))$. It then follows that for any execution E , the average cost of an operation in E is

$$\bar{t}_E \in O\left(\frac{\sum_{S \in E} (n(S) + c(S))}{m_E}\right) = O(\bar{n}_E + \bar{c}_E),$$

where the sum is taken over all operations S invoked during E , m_E is the total number of these operations. The values \bar{n}_E and \bar{c}_E are the average number of elements in the list during E and the average operation contention during E , which are defined as follows: $\bar{n}_E = \frac{\sum_{S \in E} n(S)}{m_E}$; $\bar{c}_E = \frac{\sum_{S \in E} c(S)}{m_E}$.

The rest of the paper is organized as follows. In Section 2 we discuss related work. We give our implementation of lock-free linked lists, including a sketch of the proof of correctness and analysis, in Section 3. We briefly present our implementation of lock-free skip lists in Section 4.

2. RELATED WORK

The first implementation designed for lock-free linked lists was presented by Valois [17]. The main idea of his approach was to maintain auxiliary nodes in between normal nodes of the list in order to resolve the problems that arise because of interference between concurrent operations. Also, each node in his list had a *backlink pointer* which was set to point to the predecessor when the node was deleted. These backlinks were then used to backtrack through the list when there was interference from a concurrent deletion. (A similar idea was used in an earlier, lock-based implementation of linked lists by Pugh [11].) Another lock-free implementation of linked lists was given by Harris [3]. His main idea was to *mark* a node before deleting it in order to prevent concurrent operations from changing its right pointer. We look at this implementation in detail in Section 3.1. Harris's algorithms are simpler than Valois's and his experimental results show that generally they also perform better. Yet another implementation of a lock-free linked list was proposed by Michael [8]. He used Harris's design to implement the underlying data structure, but his algorithms, unlike Harris's, were compatible with efficient memory management techniques, such as IBM freelists [7, 16] and the safe memory reclamation method [9].

Our linked lists are built combining the techniques of marking nodes [3] and using backlink pointers [11, 17], and also new ideas, such as the flag bits described in Section 3.1, which are introduced to improve the worst-case performance. We show that for any execution E , the average

cost of an operation in the execution is $O(\bar{n}_E + \bar{c}_E)$, where \bar{n}_E and \bar{c}_E were defined in the introduction. To compare, the average cost per operation in Valois's implementation can be $\Omega(m_E)$, where m_E is the total number of operations invoked during E . This is possible even when \bar{n}_E and \bar{c}_E are $O(1)$ [17]. It is not hard to see that $\bar{n}_E + \bar{c}_E \leq m_E$ (because m_E includes both completed operations and operations that are currently in progress), and the difference can be quite significant. As we show in Section 3.1, the average cost of operations in Harris's implementation can be $\Omega(\bar{n}_E \bar{c}_E)$, which is also strictly worse than in our implementation.

Pugh's skip list data structure, originally designed for sequential accesses [12], is a natural candidate for concurrent dictionary implementations, since it has good expected performance without requiring any explicit, centralized balancing. Lock-based concurrent implementations have been given by Pugh [11] and by Lotan and Shavit [13]. Valois claimed that his lock-free linked list can easily be used to obtain a lock-free skip lists [17], but it is not clear how: for example, a process traversing his linked list must maintain a collection of pointers called a cursor, and it is difficult to do so when one descends through the levels of a skip list.

Sundell and Tsigas recently gave the first lock-free implementation of a skip list [14]. Their implementation supports the INSERT, UPDATE and DELETEMIN operations. They later extended it to implement the full range of dictionary operations [15]. Another recent implementation of lock-free skip lists using single-word C&S's was presented by Fraser [2]. Although both of these designs were done independently of ours and of each other, there are some similarities between the three resulting skip list algorithms. All use the marking technique [3] to implement deletions on the individual levels of the skip list. Fraser's algorithms use Harris's design style where an operation restarts if it detects interference from a concurrent operation. Sundell and Tsigas's design allows processes to overcome the interference in some cases by using backlink pointers [11, 17]. Our design employs backlink pointers and flag bits in order to ensure that processes can always recover efficiently from such interference. All implementations use helping (in different ways) to complete deletions that could block the progress of other operations. Sundell and Tsigas incorporate a reference counting scheme to handle memory management.

Fraser gives other skip list designs that use more powerful primitives, such as multi-word C&S and software transactional memory [2]. Experimental results on lock-free linked lists [3, 8] and skip lists [2, 14, 15] suggest that they can be a practical alternative to lock-based implementations.

3. LINKED LISTS

We now present our singly-linked list implementation. Our algorithms use the C&S primitive, which atomically executes the following code.

```
C&S (Word* address, Word old_val, Word new_val) : Word
1 value = *address
2 if (value == old_val)
3   *address = new_val
4 return value
```

3.1 Linked List Design

The basic problem in designing a lock-free linked list is that when a process is deleting a node X by performing a

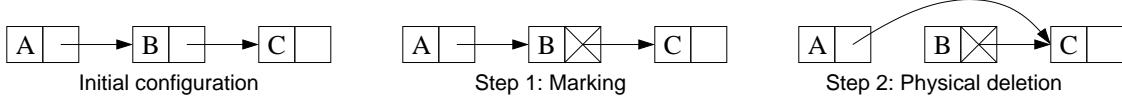


Figure 1: Harris’s two-step deletion of a node.

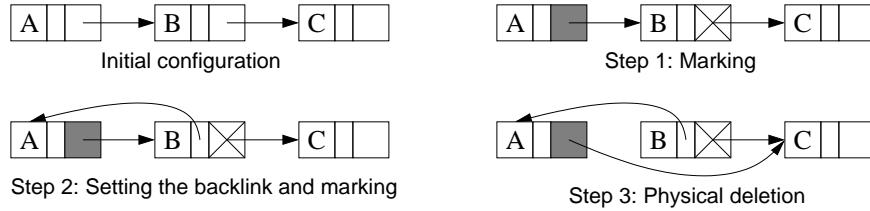


Figure 2: Three-step deletion of a node used in our implementation.

C&S on X ’s predecessor, there must be a guarantee that X ’s right pointer is not changed by a concurrent operation. Otherwise, incorrect executions can be constructed (see [17] or [3]). One of the ways to deal with this issue was given by Harris [3]. Our linked list implementation uses a similar technique, so we will look at Harris’s implementation in more detail.

Harris replaced the right pointer of each node with a composite field, which we will call a *successor field*. The successor field consists of a right pointer and a *mark bit*.¹ When a process needs to change the right pointer of a node, it applies a C&S’s to the successor field of that node. A mark bit acts as a toggle that is used to control when the right pointer of the node can be changed. Normally, the mark bit is 0. To delete a node B , a process uses two C&S’s: the first marks B ’s successor field by setting its mark bit to 1, and the second removes B from the list, as illustrated in Figure 1, where marked successor fields are crossed. A node is *logically deleted* after the first step, and *physically deleted* after the second step. All of the C&S’s performed by the algorithms modify only unmarked successor fields. Therefore, once the successor field of a node is marked, it never changes.

Harris’s approach, however, has certain performance-related problems. Consider two processes P_1 and P_2 performing concurrent operations: P_1 attempts to insert a new node after node X , and P_2 attempts to delete node X . Suppose that, just before P_1 is about to execute a C&S, P_2 marks node X , and so P_1 ’s C&S fails. When this happens, Harris’s algorithms require P_1 to restart from the beginning of the list, which can lead to poor performance. Consider an execution E in a system of q processes. First insert n keys into the list. Then make one process P_q repeatedly delete the last node of the list, while the rest of the processes P_1, \dots, P_{q-1} attempt to insert new nodes at the end of the list. In each round of the execution, P_q marks a node right after processes P_1, \dots, P_{q-1} have located the correct insertion position, but before any of them perform a C&S. Each time P_1, \dots, P_{q-1} attempt to insert the keys at the end of the list, they have to search through the whole list to locate the appropriate insertion position, and therefore the total work

done by the system is $\Omega(q \cdot (n + (n - 1) + \dots + 1)) = \Omega(qn^2)$. If we make $n > q$, then the average cost of an operation in this execution is $\Omega(qn) = \Omega(\bar{n}_E \bar{c}_E)$. (The variables \bar{n}_E and \bar{c}_E were defined in the introduction.)

Our implementation achieves better worst-case performance by making processes recover from failures instead of restarting. We augment each node of our data structure with an additional pointer field called *backlink*. When a node X gets deleted, its backlink is set to X ’s predecessor. If some process P then fails a C&S because X is marked, P follows X ’s backlink to X ’s predecessor. If the predecessor is also marked, P follows the predecessor’s backlink, and so on, until it reaches an unmarked node U . Then P resumes its operation from U rather than from the beginning of the list. The sequence of backlinks that P traverses before reaching U is called a *chain of backlinks*. The introduction of backlinks alone, however, does not guarantee the desired operation complexity. The problem is that long chains of backlinks can be traversed by the same process many times. This happens when these chains *grow towards the right*, i.e. when backlink pointers are set to marked nodes, and thus nodes are linked to the right end of the chains. We eliminate this possibility by introducing *flag bits*.

The flag bit can be thought of as a warning that a deletion of the next node is in progress. Like the mark bit, the flag bit is part of the successor field, and is initially set to 0. When a node is *flagged* (i.e. when its flag bit is set to 1), its successor field is fixed and cannot be marked or otherwise changed until the flag is removed. Also, a marked node can never get flagged, and therefore no node can be both flagged and marked. Before marking a node B , a process flags the predecessor node A , thus ensuring that when B ’s backlink is set to point to A , it will not be pointing to a marked node. Figure 2 illustrates how deletions are performed in our data structure. Shaded boxes denote flagged successor fields, and crossed boxes denote marked successor fields. The deletion of node B consists of three steps. (1) Flagging the predecessor node A by applying C&S to its successor field (Figure 2, Step 1). (2) Setting B ’s backlink to point to its predecessor A and then marking B by applying C&S to its successor field (Figure 2, Step 2). (3) Performing a physical deletion of node B and removing A ’s flag by applying C&S to A ’s successor field (Figure 2, Step 3).

To preserve the lock-freedom property, we allow processes to help one another with deletions. For example, if a process

¹In many modern architectures, a 32-bit word that stores a pointer has two unused bits. One of those can be used to store the mark bit and the other can be used to store the flag bit that we introduce later.

```

SEARCH (Key k) : Node
// Searches for a node with the supplied key.
1 (curr_node, next_node) = SEARCHFROM(k, head)
2 if (curr_node.key == k)
3   return curr_node
4 else
5   return NO_SUCH_KEY

SEARCHFROM (Key k, Node *curr_node) : (Node, Node)
// Finds two consecutive nodes n1 and n2
// such that n1.key ≤ k < n2.key.
1 next_node = curr_node.right
2 while (next_node.key ≤ k)
    // Ensure that either next_node is unmarked,
    // or both curr_node and next_node are
    // marked and curr_node was marked earlier.
3   while (next_node.mark == 1 and
          (curr_node.mark == 0 or
           curr_node.right ≠ next_node))
4     if (curr_node.right == next_node)
5       HELPMARKED(curr_node, next_node)
6     next_node = curr_node.right
7   if (next_node.key ≤ k)
8     curr_node = next_node
9     next_node = curr_node.right
10 return (curr_node, next_node)

HELMARKED (Node *prev_node, Node *del_node)
// Attempts to physically delete the marked
// node del_node and unflag prev_node.
1 next_node = del_node.right
2 C&S(prev_node.succ, (del_node, 0, 0), (next_node, 0, 0))

```

Figure 3: SEARCH, SEARCHFROM, and HELPMARKED.

cannot complete its operation because of a flagged node, it will try to complete the corresponding deletion, thus removing the flag, and then continue with its own operation.

3.2 Algorithms

The nodes in our linked list are ordered by their keys, and for simplicity our data structure does not allow users to insert duplicate keys. Each node has the following fields: key, element, backlink, and successor. The successor field is denoted *succ* in our pseudocode, and it is composed of three parts: a *right* pointer, a *mark* bit, and a *flag* bit. So, for each node *n*, *n.succ* = (*n.right*, *n.mark*, *n.flag*). The head node and the tail node of the list contain dummy keys $-\infty$ and $+\infty$, and are referenced by the shared variables *head* and *tail* respectively. The pseudocode for our algorithms is shown in Figures 3 to 5. The routines SEARCH, INSERT, and DELETE implement the corresponding dictionary operations.

The SEARCHFROM routine is used to perform searches in our data structure. It traverses the list starting from the specified node, and returns pointers to two nodes *n₁* and *n₂*, that satisfy the following condition at some time during the execution of SEARCHFROM: *n₁.right* = *n₂* and *n₁.key* ≤ *k* < *n₂.key*. SEARCHFROM also deletes any marked nodes that it sees by calling the HELPMARKED routine (line 5). We could also write a SEARCHFROM2 routine, identical to the SEARCHFROM, except that “≤” in lines 2 and 7 would be replaced with “<”. In our pseudocode, we

```

DELETE (Key k) : Node
// Attempts to delete a node with the supplied key.
1 (prev_node, del_node) = SEARCHFROM(k - ε, head)
2 if (del_node.key ≠ k) // k is not found in the list.
3   return NO_SUCH_KEY
4 (prev_node, result) = TRYFLAG(prev_node, del_node)
5 if (prev_node ≠ null)
6   HELPFLAGGED(prev_node, del_node)
7 if (result == false)
8   return NO_SUCH_KEY
9 return del_node

HELPFLAGGED (Node *prev_node, Node *del_node)
// Attempts to mark and physically delete node del_node,
// which is the successor of the flagged node prev_node.
1 del_node.backlink = prev_node
2 if (del_node.mark == 0)
3   TRYMARK(del_node)
4 HELPMARKED(prev_node, del_node)

TRYMARK (Node del_node)
// Attempts to mark the node del_node.
1 repeat
2   next_node = del_node.right
3   result = C&S(del_node.succ, (next_node, 0, 0),
              (next_node, 1, 0))
4   if (result == (*, 0, 1)) // failure due to flagging
5     HELPFLAGGED(del_node, result.right)
6 until (del_node.mark == 1)

```

Figure 4: DELETE, HELPFLAGGED, and TRYMARK.

use SEARCHFROM(*k* - ϵ , *n*) to denote SEARCHFROM2(*k*, *n*). The two nodes that SEARCHFROM(*k* - ϵ , *head*) returns satisfy *n₁.key* < *k* ≤ *n₂.key* (and *n₁.right* = *n₂*).

The SEARCH(*k*) routine simply uses SEARCHFROM to find the node with key *k* in the list, if it exists. The INSERT routine starts by calling SEARCHFROM to find where to insert the new key. Then it verifies that the new key is not a duplicate, creates a new node, and enters the loop in lines 5–22, from which it can exit only if it successfully inserts the new node or another process inserts a node with the same key (lines 20–22). In each iteration of the loop, it attempts to insert the new node between *prev_node* and *next_node* by performing a C&S in line 11. If the C&S fails, INSERT detects the reason, recovers from the failure, and enters the next iteration. The reason for the failure can only be the change of *prev_node*'s successor field. There are several possible ways in which this successor field can change: it can get redirected to another node, flagged, marked, or any two of the above, except that it cannot be both marked and flagged. If *prev_node* got flagged, it means that another process was performing a deletion of the successor node. In this case INSERT calls the HELPFLAGGED routine (lines 15–16), which helps to complete that deletion and remove the flag from *prev_node*. If *prev_node* got marked, INSERT traverses the backlinks until it finds an unmarked node and then sets *prev_node* to point to it (lines 17–18). In any case, in line 19 INSERT invokes SEARCHFROM starting from *prev_node* to find the correct location for the insertion in the updated list, and updates its *prev_node* and *next_node* pointers. Then INSERT enters the next iteration of the loop.

```

TRYFLAG (Node *prev_node, Node *target_node) : (Node, Boolean)
// Attempts to flag the predecessor of target_node. Prev_node is the last node known to be the predecessor.
1 while (true)
2   if (prev_node.succ == (target_node, 0, 1))           // Predecessor is already flagged. Report
3     return (prev_node, false)                         // the failure, return a pointer to prev_node.
4   result = c&zs(prev_node.succ, (target_node, 0, 0), (target_node, 0, 1)) // Flagging attempt
5   if (result == (target_node, 0, 0))                  // Successful flagging. Report the success,
6     return (prev_node, true)                           // // return a pointer to prev_node.
7   if (result == (target_node, 0, 1))                  // Failure due to flagging by a concurrent operation.
8     return (prev_node, false)                         // Report the failure, return a pointer to prev_node.
9   while (prev_node.mark == 1)                         // Possibly a failure due to marking. Traverse
10    prev_node = prev_node.backlink                    // a chain of backlinks to reach an unmarked node.
11  (prev_node, del_node) = SEARCHFROM(target_node.key - ε, prev_node)
12  if (del_node ≠ target_node)                      // target_node got deleted.
13  return (null, false)                           // Report the failure, return no pointer.

INSERT (Key k, Element e) : Node
// Attempts to insert a new node with the supplied key.
1  (prev_node, next_node) = SEARCHFROM(k, head)          // prev_node.key ≤ k < next_node.key
2  if (prev_node.key == k)                            // If the predecessor is flagged, help
3    return DUPLICATE_KEY                           // the corresponding deletion to complete.
4  newNode = new Node(key = k, element = e)
5  while (true)
6    prev_succ = prev_node.succ
7    if (prev_succ.flag == 1)                         // Insertion attempt.
8      HELPFLAGGED(prev_node, prev_succ.right)        // // Successful insertion.
9    else
10      newNode.succ = (next_node, 0, 0)
11      result = c&zs(prev_node.succ, (next_node, 0, 0), (newNode, 0, 0)) // Failure.
12      if (result == (next_node, 0, 0))                // Failure due to flagging.
13        return newNode
14      else
15        if (result == (*, 0, 1))                     // Help complete the corresponding deletion.
16          HELPFLAGGED(prev_node, result.right)        // // Possibly a failure due to marking. Traverse a
17          while (prev_node.mark == 1)                  // chain of backlinks to reach an unmarked node.
18            prev_node = prev_node.back_link
19  (prev_node, next_node) = SEARCHFROM(k, prev_node)    // prev_node.key ≤ k < next_node.key
20  if (prev_node.key == k)
21    free newNode
22  return DUPLICATE_KEY

```

Figure 5: TRYFLAG and INSERT.

The DELETE routine performs a three-step deletion of the node, as discussed in Section 3.1. DELETE starts by calling SEARCHFROM, and then calls TRYFLAG to perform the first deletion step (flagging the predecessor). TRYFLAG repeatedly attempts to flag *del_node*'s predecessor, until the flag is placed or *del_node* gets deleted. TRYFLAG returns two values: a node pointer *prev_node* and a boolean *result* value. There can be three ways the TRYFLAG routine can return. If TRYFLAG itself flags *del_node*'s predecessor, it returns a pointer to the predecessor and *result* = **true**. If TRYFLAG detects that another process flagged *del_node*'s predecessor (which means that another process is performing a deletion of *del_node*), it returns a pointer to the predecessor and *result* = **false**. If TRYFLAG detects that *del_node* got deleted from the list, it returns **null** and *result* = **false**. If *prev_node* returned by TRYFLAG is not **null**, DELETE proceeds by calling the HELPFLAGGED routine, which performs the second and the third deletion steps by calling TRYMARK and HELPMARKED. If TRYFLAG also returned *result* =

true, DELETE returns a pointer to the deleted node in line 9 (i.e. reports success). If *result* = **false**, it means that either *del_node* got deleted, or another process flagged *del_node*'s predecessor (and is going to report success). In this case DELETE returns NO SUCH KEY.

3.3 Correctness

We will now present a sketch of the proof of correctness. The complete proof is available in [1]. We first prove several invariants. To state these invariants we classify the nodes into three categories as follows.

DEF 1. A node is regular if it is was inserted into the list, and it is unmarked.

DEF 2. A node is logically deleted if it is marked and has a regular node linked to it, i.e. *n* is logically deleted if *n*.mark = 1 and there exists a regular node *m* such that *m*.right = *n*.

DEF 3. A node is physically deleted if it is marked and there is no regular node linked to it.

At any time, each node that was ever inserted into the list fits into exactly one of these three categories. We prove that the following invariants apply to all regular, logically deleted, and physically deleted nodes of the list.

INV 1. Keys are strictly sorted: for any two nodes n_1, n_2 , if $n_1.right = n_2$, then $n_1.key < n_2.key$.

INV 2. The union of regular and logically deleted nodes forms a linked list structure, i.e. if n is a regular or a logically deleted node and $n \neq \text{head}$, then there is exactly one regular or logically deleted node m such that $m.right = n$. Node m is called n 's predecessor. If $n \neq \text{tail}$, then node $n.right$ is regular or logically deleted, and it is called n 's successor. The head node has no predecessor, and the tail node has no successor.

INV 3. For any logically deleted node, its predecessor is flagged (and unmarked), and its successor is not marked, i.e. if n is logically deleted, and m is a node of the list such that m is not physically deleted and $m.right = n$, then $m.succ = (n, 0, 1)$ and $(n.right).mark = 0$.

INV 4. For any logically deleted node, its backlink is pointing to its predecessor, i.e. if n is logically deleted, and m is a node of the list such that m is not physically deleted and $m.right = n$, then $n.backlink = m$.

INV 5. No node can be both marked and flagged at the same time.

It follows from Inv 3, that if two marked nodes are adjacent, then at least one of them is physically deleted.

The proof of the invariants goes as follows. Inv 5 is trivial. Inv 1–3 are proved by induction on the number of successful C&S's. This proof is lengthy, but fairly straightforward. After this we use the proved invariants to show that once a node's backlink is set, it never changes. This fact is used to prove Inv 4 by induction on the number of successful C&S's. We then prove two important properties of our algorithms. First, we show that deletions in our data structure work as intended, i.e. they are performed in three steps: first flagging the predecessor, then marking the node, and finally physically deleting the node. The second proposition states SEARCHFROM postconditions: if SEARCHFROM(k, n) returns (n_1, n_2) and if $n.key \leq k$, then (1) $n_1.key \leq k < n_2.key$, (2) there exists a time during the execution of SEARCHFROM when $n_1.right = n_2$, and (3) if n is unmarked at some time T' before SEARCHFROM is invoked, then there exists time T between T' and the moment SEARCHFROM returns, when n is unmarked and $n.right = n_2$.

Finally, we use all these facts to prove the correctness of our implementation. At any time, we say that the set of elements currently stored in the dictionary is the set of the elements contained in the regular nodes, and we show that all operations can be linearized so that their return values are consistent with this definition. Specifically,

- The searches are linearized at time T specified by postcondition (3) of the SEARCHFROM routine they invoke. If the search is successful, the node it returns is a regular node at time T ; if the search is unsuccessful there are no regular nodes with key k in the list at T .

- Each successful insertion is linearized when it successfully performs a C&S (line 11 in the INSERT routine) that inserts the node created in line 4. Each unsuccessful insertion is linearized at time T when the third postcondition holds for the last SEARCHFROM routine it invokes (line 1 or 19 in INSERT routine). At that time there is a regular node with the same key in the list.
- We linearize a successful deletion when the node it returns becomes marked (and therefore logically deleted). Unsuccessful deletions are linearized as follows. If the SEARCHFROM called by DELETE in line 2 found no node with key k , linearize the deletion at the time T specified by postcondition (3) for that SEARCHFROM. If the TRYFLAG called by DELETE returned in line 3, 8, or 13 (which means that another process was executing a concurrent deletion of the same node, and performed at least the first step of the deletion — flagging the predecessor), then we linearize the deletion immediately after del_node gets marked. Note that lines 5–6 of DELETE ensure that del_node gets marked (and then physically deleted) before DELETE returns in line 8, so this linearization is valid. Also note that the concurrent deletion that flagged del_node 's predecessor reports success when it returns.

3.4 Performance Analysis

Here we present a sketch of the amortized analysis of our linked list data structure. We start by explaining our billing scheme, first giving a general intuition behind it, and then defining it formally using the mapping β in Def 4. We then explain how we use this billing scheme to prove the bound on the amortized cost of operations. The full version of our amortized analysis is available in [1].

It is not hard to show that in order to calculate the cost of our algorithms, it is only essential to calculate the number of C&S attempts, the number of backlink pointer traversals (line 10 in TRYFLAG and line 18 in INSERT), and the number of $next_node$ and $curr_node$ pointer updates by searches (lines 6 and 8 in SEARCHFROM respectively). Counting these steps gives an accurate picture of the required time (up to a constant factor), and therefore we ignore other steps in our amortized analysis. When, later on, we talk about steps taken by the processes, we mean one of these *essential steps*.

We classify the (essential) steps of each operation S into three categories: successful C&S's, *necessary steps*, and *extra steps*. The necessary steps are the (non-C&S) steps that S normally has to perform in order to complete (e.g. in order to complete a search for key k , S has to traverse all nodes with keys smaller than k). Intuitively, the necessary steps are the steps that an operation needs to perform even if it is executing on a sequential linked list. By contrast, the extra steps are the steps that S has to take because of interference from other operations (e.g. when S fails a C&S because of a change performed by a concurrent operation). The cost of the necessary steps of S is called the *necessary cost of S* , and the cost of the extra steps of S is called the *extra cost of S* . In our analysis, we show that the necessary cost of S is always $O(n(S))$ ($n(S)$ and $c(S)$ were defined in the introduction), and we use a mapping to bill all of the extra cost of S to successful C&S's that are *part of* operations concurrent with S . We say that a C&S is *part of operation S* if it is successful, and it logically belongs to that operation.

Specifically, each successful C&S that inserts a new node is part of the corresponding successful insertion, and successful C&S's that flag, mark, and physically delete nodes are part of the corresponding successful deletions. A (successful) C&S that is part of a given operation is not necessarily performed by the process that is executing this operation, because processes help one another with deletions.

We define the *amortized cost* of a successful C&S C , denoted $\hat{t}(C)$, to be (actual cost of C) + (total cost billed to C). Note that the first term is 1. We define the *amortized cost* of S , denoted $\hat{t}(S)$, to be (actual cost of S) - (total cost billed from S to successful C&S's) + (total cost billed to successful C&S's that are part of S). The second term is the extra cost of S , so

$$\begin{aligned}\hat{t}(S) &= ((\text{necessary cost of } S) + (\text{extra cost of } S) + \\ &\quad (\text{cost of successful C&S's performed by } S)) - \\ &\quad (\text{extra cost of } S) + (\text{total cost billed to} \\ &\quad \text{successful C&S's that are part of } S) \\ &= (\text{necessary cost of } S) + (\text{cost of successful} \\ &\quad \text{C&S's performed by } S) + (\text{total cost billed to} \\ &\quad \text{successful C&S's that are part of } S) \\ &= (\text{necessary cost of } S) + (\text{amortized cost of} \\ &\quad \text{successful C&S's that are part of } S).\end{aligned}$$

We prove that the first term is $O(n(S))$ and that, for any C&S C that is part of operation S , the total cost billed to C is $O(c(S))$. Since at most three C&S's can be part of any given operation, we conclude that the second term is $O(c(S))$. Therefore, $\hat{t}(S) = O(n(S) + c(S))$. Note that here the $O(n(S))$ term comes purely from the cost of the steps that even a sequential algorithm needs to perform, while the overhead that comes from concurrency is limited by an additive term of $O(c(S))$. We now describe all of the steps outlined above in more detail.

To define our billing scheme formally, we introduce a *mapping function* β , given below. This mapping also formally defines the set of the extra steps and the set of the necessary steps for every operation. Function β will map successful C&S's to themselves. All other steps mapped to themselves are necessary steps. The remaining steps are extra steps. The logic behind the design of this mapping function is that each extra step is mapped to the successful C&S that performed the change that causes this extra step to be taken. For example, the step of traversing node n that was inserted after S was invoked is mapped to the C&S that inserted n . To make it easier to define β , we categorize C&S's performed by our algorithms into four types: (1) insertion C&S (line 11 in INSERT), (2) flagging C&S (line 4 in TRYFLAG), (3) marking C&S (line 3 in TRYMARK), and (4) physical deletion C&S (line 2 in HELPMARKED).

DEF 4. Let Q be the set of essential steps in the entire execution E . Function β maps Q to itself. If some operation S performs step $s \in Q$, β maps this step either to itself, or to a successful C&S that is part of another operation as described below.

- **C&S's:** Suppose a C&S C on the successor field of node n was executed. If C is successful, then we map it to itself. If C fails, and it is not of the fourth type, we map it to the C&S that last modified $n.\text{succ}$. If C is of the fourth type and it fails, we map it to the C&S that

physically deleted the node that C was trying to delete. (We show that such a C&S had to be performed.)

- **Backlink traversals:** A backlink pointer traversal from node n to node m is mapped to the C&S that marked node n .
- **Next-node pointer updates:** Suppose the update changes *next-node* from m to m' . If m is physically deleted before the update, we map the update to the C&S that physically deleted m . (Note that even though this C&S could be performed by HELPMARKED called from this SEARCHFROM routine in line 5, it is part of another operation.) Otherwise we map the update to the C&S that inserted m' .

- **Curr-node pointer updates:** Suppose the update sets *curr-node* pointer to node n . If n was inserted into the list after operation S was invoked, then the update is mapped to the C&S that inserted n . Otherwise, the update is mapped to itself.

To prove our bound on the amortized cost of operations, we need to show that the amortized cost of each C&S that is part of an operation S , is $O(c(S))$. This is the most important and the most technical part of our amortized analysis. Below we briefly describe this proof.

There are four types of steps that Def 4 bills to successful C&S's. For each of them we prove that if a step of that type performed by operation S' is mapped by β to a (successful) C&S C , then (1) no other steps of the same type performed by S' are mapped to C , and (2) C was performed during the execution of S' . It then follows that no more than $c(S)$ steps of each type can be mapped to C , where S is the operation C is part of. Proving (1) and (2) for *next-node* updates is fairly straightforward. For *curr-node* pointer updates, we first show that no operation can set *curr-node* pointer (in line 8 of a SEARCHFROM) to a given node more than once, and then (1) and (2) follow. For backlink traversals, we show that if operation S traverses a backlink from node n , then n got marked during S , and S never traversed a backlink from n before, which leads to (1) and (2). In this part of the proof we rely on the fact that chains of backlinks never grow towards the right (see Section 3.1). For unsuccessful C&S's, we prove two lemmas. The first one states that if C' is an unsuccessful C&S of type four on the successor field of node n performed by operation S' , then there exists a time T during S' when $n.\text{succ}$ was such that C' would have succeeded, and S' performed no C&S's on $n.\text{succ}$ between T and C' . The second lemma states a similar, but slightly weaker claim for the C&S's of the first three types. Using these two lemmas, we show that (1) and (2) hold for unsuccessful C&S's as well.

Since no more than $c(S)$ steps of each type can be mapped by β to a successful C&S that is part of S , it follows that the amortized cost of a successful C&S is $O(c(S))$. Since at most three successful C&S's can be part of S , it follows that the amortized cost of successful C&S's that are part of S is $O(c(S))$. To prove that the amortized cost of S is $O(n(S) + c(S))$ we now only need to show that the total cost of the steps of S that are not mapped by β to the successful C&S's (i.e. the necessary cost of S) is $O(n(S))$.

First, note that the only steps of S that are not mapped to the successful C&S's are the *curr-node* pointer updates

in line 8 of `SEARCHFROM` routines called by S . Furthermore, by the definition of β such an update is mapped to itself (and not to a successful C&S) only if the node n to which the `curr_node` pointer is set to by this update is inserted before the invocation of S . It is also not hard to show that n must be unmarked at some moment during the execution of S , which means that n is a regular node when S is invoked (since nodes never get unmarked). Also, as mentioned above, no operation can set the `curr_node` pointer (in line 8 of a `SEARCHFROM` routine) to a given node more than once. Consequently, the total number of steps of S that are not mapped to the successful C&S's cannot be greater than the number of regular nodes when S is invoked, i.e. $n(S)$. This concludes our amortized analysis, yielding $\hat{t}(S) = O(n(S)) + O(c(S))$ (where the $O(n(S))$ term comes from the necessary cost of S , and the $O(c(S))$ term comes from the concurrency overhead).

4. SKIP LISTS

In this section we briefly discuss our lock-free implementation of a skip list data structure and give a sketch of the proof of correctness. The algorithms and the complete proof of correctness are available in [1].

A skip list [12] is a sequential dictionary data structure, in which searches, insertions, and deletions have an expected cost of $O(\log(n))$ (and worst-case cost of $O(n)$), where n is the number of elements in the dictionary. The expectation is taken over the random numbers generated inside the algorithms. Our lock-free skip list architecture has some differences from Pugh's original design to make it easier to reuse our linked list algorithms. As shown in Figure 6, we represent each key by a *tower* of nodes. A tower that has H nodes in it is said to have *height* H . The height of each tower is chosen randomly by coin flips. The bottom node of a tower is called the *root node*, and it acts as a representative of the whole tower. The *head tower* and the *tail tower* store dummy keys $-\infty$ and $+\infty$ respectively. Horizontally, the nodes of the skip list are arranged in *levels*: the root nodes are on level one, the nodes immediately above them are on level two, and so on. Nodes of the same level form a singly-linked list, sorted according to their keys.

In the original skip list design [12], Pugh uses a single node with an array of H forward pointers to represent a tower of height H . The difference between our architecture and Pugh's architecture is not very significant, but it makes it easier to explain our algorithms in terms of the linked list algorithms already described. For convenience, we use the same terminology when we compare our skip list implementation with others [2, 15], even though they use Pugh's architecture.

Other recent lock-free skip list designs [2, 15] implement individual levels using linked list algorithms that can exhibit bad worst-case behaviour, as described in Section 3.1. (Furthermore, although Sundell and Tsigas incorporate backlinks in their implementation, a backlink is not guaranteed to be set when it is needed, and their backlink is useful on a given level only if the tower it is pointing to is sufficiently high.) Because of the randomization used by the algorithm, it is unclear whether an adversary could exploit the worst-case behaviour on individual levels to force the skip list as a whole to experience bad worst-case behaviour. Our design was driven by an effort to ensure that individual levels of the skip list have good worst-case complexity by using our

new linked list algorithms, so that a tight analysis of the average expected complexity of the skip list operations would be feasible. However, new difficulties arise when attempting to do this, as explained in more detail below. Thus, the problem of proving a good upper bound on the complexity of a lock-free skip list implementation remains open.

In our data structure, a node Q that is not a node of the head or tail tower has the following fields: `key`, `backlink`, `succ`, `down`, and `tower_root`. The first three fields are the same as in our lock-free linked lists, `down` is a pointer to the node one level lower than Q (or `null` if Q is a root node), and `tower_root` is a pointer to the root node of Q 's tower. If Q is a root node, it also has an `element` field. Nodes of the head tower do not have elements, backlinks or `tower_root` pointers, but each of them has an `up` pointer, pointing to the node above. The top node of the head tower has its `up` pointer set to itself. Nodes of the tail tower contain only the key $+\infty$. A pointer to the bottom node of the head tower is referred to by a shared variable `head`.

We now give a high-level overview of our algorithms. An insertion builds the tower from bottom to top, i.e. first it inserts the root node, then, if necessary, the node at level two, and so on. An insertion is linearized when the root node is inserted, since after that moment, all the searches are able to find the key. A deletion first deletes the root node of the tower, and then deletes the rest of the nodes of the tower from top to bottom. A deletion is linearized when the root node gets marked. A tower whose root node is marked is called *superfluous*; all the nodes of such a tower are called *superfluous* as well.

Regardless of whether deletions delete the towers from top to bottom, or from bottom to top, superfluous nodes can still exist, because while a process P is constructing a tower Q , Q 's root node can get marked by another process, and P can add a new node to Q before it notices the marking. It is possible to solve this problem by marking uninherited nodes if Harris's design is used to implement individual levels of the skip list [2], but with our design this is not feasible because of flags.

The searches in our skip list help deletions by physically deleting superfluous nodes they encounter in order to avoid traversing superfluous towers. Our decision to implement searches this way was motivated by the observation that if searches traverse superfluous towers without physically deleting or marking their nodes, it is possible to construct an execution E where the average cost of operations would be $\Omega(m_E)$ by forcing operations to repeatedly traverse a chain of backlinks of length $\Omega(m_E)$ on the lowest level of the skip list (m_E was defined in the Introduction). Sundell and Tsigas [15] use a different method to deal with this problem: their searches can enter superfluous towers via unmarked nodes, but if a search detects a marked node in a tower it is traversing, it marks all the nodes of this tower. Subsequent searches physically delete these marked nodes if they encounter them (assuming the main Delete operation has not already done so), thus making numerous traversals of the same chain of backlinks impossible.

Even though searches in our implementation delete superfluous nodes whenever they encounter them, and therefore they cannot be forced so to traverse the same chain of backlinks repeatedly, there exist scenarios when an operation can be forced to traverse backlinks of the nodes that were deleted before the operation started (something that never happens

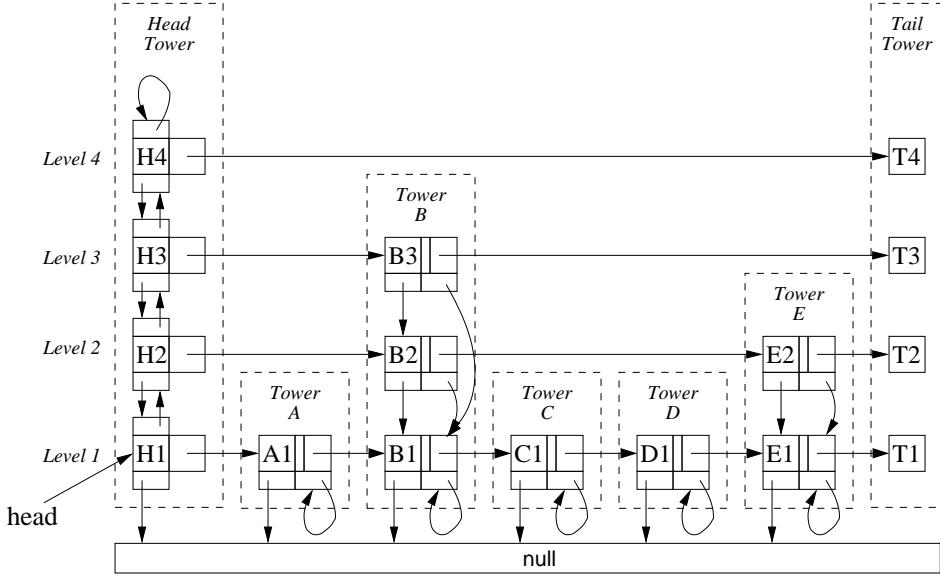


Figure 6: Lock-free skip list design.

in our linked list implementation). These scenarios can only be constructed by a very careful scheduling of processes tailored for a given distribution of the heights of the towers. Their existence, however, makes our correctness proofs quite complicated, but more importantly, it is not clear what effect they may have on the worst-case performance of our implementation.

The pseudocode for the skip list algorithms is available in [1], and here we describe them only briefly. Each level of the skip list can be viewed as a linked list. Therefore, the routines that we use to operate on the individual levels are similar to our linked list routines. The three major routines that implement the dictionary operations are `SEARCH_SL`, `INSERT_SL`, and `DELETE_SL`. The `SEARCH_SL` routine calls `SEARCHTOLEVEL_SL` to determine if there is a root node (and hence, a tower) with key k in the list. `SEARCHTOLEVEL_SL(k, v)` is used to locate the nodes on level v with keys closest to k . It traverses levels starting from the top one, and each time it reaches a key larger than k , it goes down one level. To traverse individual levels, it uses the `SEARCHRIGHT` routine, which is similar to the `SEARCHFROM` in our linked list algorithms. The only difference is that `SEARCHRIGHT` deletes the superfluous nodes along its way, performing all three deletion steps if necessary, whereas `SEARCHFROM` physically deletes only those nodes that are already logically deleted.

The `INSERT_SL` routine determines the height of the tower it needs to insert by flipping a coin, and enters a loop where it inserts the nodes of the tower one by one from bottom to top. If a concurrent process inserts a root node with the same key, `INSERT_SL` reports failure and returns. Each complete iteration of the loop increases the height of the new tower by one. `INSERT_SL` exits from that loop if it finishes the construction of the new tower, or if the construction of a new tower gets *interrupted* by a deletion: if `INSERT_SL` notices that the root node got marked, it exits reporting success. The `DELETE_SL` routine first deletes the root node of the tower with the supplied key k , making the rest of

the nodes in the tower superfluous. It then calls `SEARCHTOLEVEL_SL` for k , which deletes these superfluous nodes (from top to bottom).

We now briefly sketch the proof of correctness. The first part of the proof is similar to the correctness proof for the linked lists. We show that Inv 1–5 (see Section 3.3) hold for each level of our skip list, and that nodes never change levels. We also show that deletions of individual nodes are performed in three steps (flag, mark, physical deletion), and that the same postconditions that hold for `SEARCHFROM` hold for `SEARCHRIGHT` as well. These postconditions guarantee that if `SEARCHRIGHT` starts from a node n that is not superfluous at time T' , then the node m it ends in is not marked at T' . However, m may be superfluous at T' , but we show that this can happen only if `SEARCHRIGHT` enters m by traversing backlinks, and in this case $m.key < n.key$. The fact that `SEARCHRIGHT` may traverse superfluous nodes leads to the fact that `SEARCHTOLEVEL_SL` may enter marked nodes when it descends from one level to the next (although scenarios where this happens are fairly contrived). This is why an operation can traverse backlinks of the nodes that were deleted before the operation started. As mentioned earlier, this is an obstacle to applying the same kind of performance analysis to skip lists, as we used for linked lists. After proving some weaker postconditions for `SEARCHTOLEVEL_SL` and `INSERT_SL`, we then show that our skip list has the correct vertical structure within each tower, i.e. the nodes on different levels that contain the same key form a linked list. Then we prove the stronger `SEARCHTOLEVEL_SL(k, v)` postconditions: we show that the node n it ends in is unmarked, and, if $n.key = k$, n is also not superfluous (at some time during the search).

Finally, we say that the set of elements currently stored in the dictionary is the set of the elements of the regular root nodes, and we show that all operations can be linearized consistently with this definition. We prove that our implementation is lock-free by showing that the only way a

process's operation can be delayed indefinitely is if other processes continually perform successful C&S's.

We also investigate the distribution of the heights of the towers in our skip list. We call a tower *full* if its insertion has finished without an interruption; otherwise we say that a tower is *incomplete*. A non-deleted tower can be incomplete only if its insertion or its deletion is in progress, so the number of incomplete towers at any time is bounded by the point contention. The distribution of the heights of the full towers may be a little different from the heights distribution in a sequential skip list, because higher towers are more likely to be incomplete. However, we believe this would not affect the expected running time significantly.

5. CONCLUSION

We have presented new algorithms implementing lock-free linked lists. We proved that the average cost of operations on our linked lists is linear in the length of the list plus the contention, for any possible sequence of operations and any possible scheduling. To perform our analysis we used a billing technique that might be applicable to other distributed data structures. We showed that our linked list algorithms can be used in a fairly modular way as the basis for a lock-free implementation of skip lists.

We have not explicitly incorporated a memory management technique, but a possible approach is to use Valois's reference counting method [10, 17], which is applicable to both our linked lists and our skip lists, because there are no cycles among the physically deleted nodes.

There are a number of directions for future work in this area. It remains an open problem to get a good bound on the average expected complexity of lock-free implementations of a skip list (or, more generally, a dictionary data structure). We think the implementation given here and the amortized analysis technique may be useful in doing this. However some difficulties remain. For example, an adversary might choose to delete all of the tall towers that are used to traverse the skip list quickly. Although an oblivious adversary (who cannot see the outcomes of coin flips) cannot directly know the heights of the towers, in a distributed application it might indirectly get some information about them by seeing how many steps are required to do searches. It might be more realistic to separate the two roles of the adversary: choosing the operations and choosing the schedule.

On a more general note, it would be interesting to develop a usable and practical alternative to the worst-case amortized analysis, which can be overly pessimistic, in the context of lock-free data structures. A feasible way of doing an amortized analysis that bounds the average complexity over possible schedules would be of great interest.

Acknowledgements

This research was funded by the Natural Sciences and Engineering Research Council of Canada and by an Ontario Graduate Scholarship. We thank Håkan Sundell, Philippas Tsigas and the anonymous referees for pointing out related work and providing helpful comments.

6. REFERENCES

- [1] M. Fomitchev. Lock-free linked lists and skip lists. Master's thesis, York University, October 2003.
<http://www.cs.yorku.ca/~mikhail>.
- [2] K. A. Fraser. *Practical lock-freedom*. PhD thesis, University of Cambridge, December 2003. Technical Report UCAM-CL-TR-579.
- [3] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Symposium on Distributed Computing*, pages 300–314, 2001.
- [4] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.
- [5] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, 1993.
- [6] M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.
- [7] IBM System/370 extended architecture, principles of operation., 1983. IBM Publication No. SA22-7085.
- [8] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the 14th annual ACM Symposium on Parallel Algorithms and Architectures*, pages 73–82, 2002.
- [9] M. M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the 21st Annual Symposium on Principles of Distributed Computing*, 2002.
- [10] M. M. Michael and M. L. Scott. Correction of a memory management method for lock-free data structures. Technical Report TR599, Computer Science Department, University of Rochester, 1995.
- [11] W. Pugh. Concurrent maintenance of skip lists. Technical Report CS-TR-2222, Computer Science Department, University of Maryland, 1990.
- [12] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of ACM*, 33(6):668–676, 1990.
- [13] N. Shavit and I. Lotan. Skiplist-based concurrent priority queues. In *Proc. 14th IEEE/ACM International Parallel and Distributed Processing Symposium*, pages 263–268, 2000.
- [14] H. Sundell and P. Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. In *Proceedings of the 17th IEEE/ACM International Parallel and Distributed Processing Symposium*, pages 84–94, April 2003.
- [15] H. Sundell and P. Tsigas. Scalable and lock-free concurrent dictionaries. In *Proceedings of the 19th ACM Symposium on Applied Computing*, pages 1438–1445, March 2004.
- [16] R. K. Treiber. Systems programming: Coping with parallelism. Research report RJ 5118, IBM Almaden Research Center, 1986.
- [17] J. D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pages 214–222, 1995.

A Provably Correct Scalable Concurrent Skip List

Maurice Herlihy^{1,2}, Yossi Lev^{1,2}, Victor Luchangco², and Nir Shavit²

¹ Computer Science Department, Brown University, Providence, RI 02912

² Sun Microsystems Laboratories, 1 Network Drive, Burlington, MA 01803

Abstract. We propose a new concurrent skip list algorithm distinguished by a combination of simplicity and scalability. The algorithm employs optimistic synchronization, searching without acquiring locks, followed by short lock-based validation before adding or removing nodes. It also logically removes an item before physically unlinking it. Unlike some other concurrent skip list algorithms, this algorithm preserves the skip list properties at all times, which facilitates reasoning about its correctness. Experimental evidence shows that this algorithm performs as well as the best previously known algorithm under most circumstances.

1 Introduction

Skip lists [7] are an increasingly important data structure for storing and retrieving ordered in-memory data. In this paper, we propose a new concurrent skip-list algorithm that appears to perform as well as the best existing concurrent skip list implementation under most conditions. The principal advantage of our implementation is that it is much simpler, and much easier to reason about.

The `ConcurrentSkipListMap`, written by Doug Lea based on work by Fraser and Harris [2] and released as part of the Java™ SE 6 platform, is the best concurrent skip-list implementation that we are aware of. This algorithm is lock-free, and performs well in practice. The principal limitation of this implementation is that it is complicated. Certain interleavings can cause the usual skip list structure to be violated, sometimes transiently, and sometimes permanently. These violations do not affect performance or correctness, but they make it difficult to reason about the correctness of the algorithm. By contrast, the algorithm presented here preserves the skip list structure at all times. The algorithm is simple enough that we are able to provide a straightforward proof of correctness.

Our algorithm employs two complementary techniques. First, it is *optimistic*: methods traverse the list without acquiring locks. When a method discovers the items it is seeking, it locks the item and its predecessors, and then validates that the list is unchanged. Second, removing an item involves *logically* deleting it by marking it before it is *physically* removed (unlinked) from the list.

Experimental tests show that despite its simplicity, this algorithm performs as well as the lock-free Lea algorithm, except under conditions of extreme contention in multiprogrammed environments. In Section 6 we discuss some approaches for future work to address this issue.

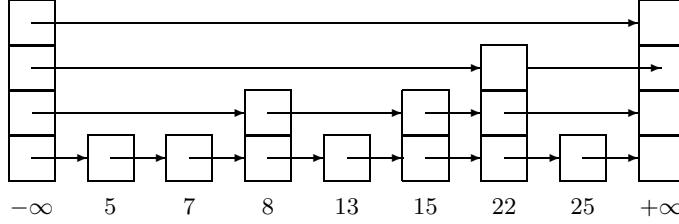


Fig. 1. A skip list with maximum height of 4. The number below each node (i.e., array of next pointers) is the key of that node, with $-\infty$ and $+\infty$ as the keys for the left and right sentinel nodes respectively.

2 Background

A skip list [7] is a linked list that is sorted by *key*, and in which nodes are assigned a random *height*, up to some maximum height, where the frequency of nodes of a particular height decreases exponentially with the height. A node in a skip list has not just one successor, but a number of successors equal to its height: each node stores a pointer to the next node in the list of each height up to its own. For example, a node of height 3 stores three “next pointers”, one to the next node of height 1, one to the next node of height 2, and one to the next node of height 3. Figure 1 illustrates a skip list in which the keys are integers.

We think of a skip list as having several *layers* of lists, and we talk about the predecessor and successor of a node at each layer. The list at each layer, other than the bottom layer, is a sublist of the list at the layer beneath it. Because there are exponentially fewer nodes of greater heights, we can find a key quickly by searching first at higher layers, skipping over large numbers of shorter nodes and progressively working downward until a node with the desired key is found, or else the bottom layer is reached. Thus, the expected time complexity of skip-list operations is logarithmic in the length of the list.

It is convenient to have *left sentinel* and *right sentinel* nodes, at the beginning and end of the lists respectively. These nodes have the maximum height, and initially, when the skip list is empty, the right sentinel is the successor of the left sentinel at every layer. The left sentinel’s key is smaller, and the right sentinel’s key is greater, than any key that may be added to the set. Searching the skip list thus always begins at the left sentinel.

3 Our Algorithm

We present our concurrent skip-list algorithm in the context of an implementation of a set object supporting three methods, `add`, `remove` and `contains`: `add(v)` adds v to the set and returns `true` iff v was not already in the set; `remove(v)` removes v from the set and returns `true` iff v was in the set; and `contains(v)` returns `true` iff v is in the set. We show that our implementation is *linearizable* [5]; that is, every operation appears to take place atomically at some point (the

```

4 class Node {
5     int key;
6     int topLayer;
7     Node** nexts;
8     bool marked;
9     bool fullyLinked;
10    Lock lock;
11 };

```

Figure 1.2. A node

linearization point) between its invocation and response. We also show that the implementation is deadlock-free, and that the `contains` operation is *wait-free*; that is, a thread is guaranteed to complete a `contains` operation as long as it keeps taking steps, regardless of the activity of other threads.

Our algorithm builds on the lazy-list algorithm of Heller et al. [3], a simple concurrent linked-list algorithm with an optimistic fine-grained locking scheme for the `add` and `remove` operations, and a wait-free `contains` operation: we use lazy lists at each layer of the skip list. As in the lazy list, the key of each node is strictly greater than the key of its predecessor, and each node has a `marked` flag, which is used to make `remove` operations appear atomic. However, unlike the simple lazy list, we may have to link the node in at several layers, and thus might not be able to insert a node with a single atomic instruction, which could serve as the linearization point of a successful `add` operation. Thus, for the lazy skip list, we augment each node with an additional flag, `fullyLinked`, which is set to `true` after a node has been linked in at all its layers; setting this flag is the linearization point of a successful `add` operation in our skip-list implementation. Figure 1.2 shows the fields of a node.

A key is in the abstract set if and only if there is an unmarked, fully linked node with that key in the list (i.e., reachable from the left sentinel).

To maintain the skip-list structure—that is, that each list is a sublist of the list at lower layers—changes are made to the list structure (i.e., the `nexts` pointers) only when locks are acquired for all nodes that need to be modified. (There is one exception to this rule involving the `add` operation, discussed below.)

In the following detailed description of the algorithm, we assume the existence of a garbage collector to reclaim nodes that are removed from the skip list, so nodes that are removed from the list are not recycled while any thread might still access them. In the proof (Section 4), we reason as though nodes are never recycled. In a programming environment without garbage collection, we can use solutions to the repeat offenders problem [4] or hazard pointers [6] to achieve the same effect. We also assume that keys are integers from `MinInt+1` to `MaxInt-1`. We use `MinInt` and `MaxInt` as the keys for `LSentinel` and `RSentinel`, which are the left and right sentinel nodes respectively.

Searching in the skip list is accomplished by the `findNode` helper function (see Figure 1.3), which takes a key `v` and two maximal-height arrays `preds` and

```

33 int findNode(int v,
34             Node* preds[],
35             Node* succs[]) {
36     int lFound = -1;
37     Node* pred = &LSentinel;
38     for (int layer = MaxHeight-1;
39          layer ≥ 0;
40          layer--) {
41         Node* curr = pred->nexts[layer];
42         while (v > curr->key) {
43             pred = curr; curr = pred->nexts[layer];
44         }
45         if (lFound == -1 && v == curr->key) {
46             lFound = layer;
47         }
48         preds[layer] = pred;
49         succs[layer] = curr;
50     }
51     return lFound;
52 }
```

Figure 1.3. The `findNode` helper function

`succs` of node pointers, and searches exactly as in a sequential skip list, starting at the highest layer and proceeding to the next lower layer each time it encounters a node whose key is greater than or equal to `v`. The thread records in the `preds` array the last node with a key less than `v` that it encountered at each layer, and that node's successor (which must have a key greater than or equal to `v`) in the `succs` array. If it finds a node with the sought-after key, `findNode` returns the index of the first layer at which such a node was found; otherwise, it returns `-1`. For simplicity of presentation, we have `findNode` continue to the bottom layer even if it finds a node with the sought-after key at a higher level, so all the entries in both `preds` and `succs` arrays are filled in after `findNode` terminates (see Section 3.4 for optimizations used in the real implementation). Note that `findNode` does not acquire any locks, nor does it retry in case of conflicting access with some other thread. We now consider each of the operations in turn.

3.1 The add operation

The `add` operation, shown in Figure 1.4, calls `findNode` to determine whether a node with the key is already in the list. If so (lines 59–66), and the node is not marked, then the `add` operation returns `false`, indicating that the key is already in the set. However, if that node is not yet fully linked, then the thread waits until it is (because the key is not in the abstract set until the node is fully linked). If the node is marked, then some other thread is in the process of deleting that node, so the thread doing the `add` operation simply retries.

```

54 bool add(int v) {
55     int topLayer = randomLevel(MaxHeight);
56     Node* preds[MaxHeight], succs[MaxHeight];
57     while (true) {
58         int lFound = findNode(v, preds, succs);
59         if (lFound != -1) {
60             Node* nodeFound = succs[lFound];
61             if (!nodeFound->marked) {
62                 while (!nodeFound->fullyLinked) {};
63                 return false;
64             }
65             continue;
66         }
67         int highestLocked = -1;
68         try {
69             Node *pred, *succ, *prevPred = null;
70             bool valid = true;
71             for (int layer = 0;
72                  valid && (layer <= topLayer);
73                  layer++) {
74                 pred = preds[layer];
75                 succ = succs[layer];
76                 if (pred != prevPred) {
77                     pred->lock.lock();
78                     highestLocked = layer;
79                     prevPred = pred;
80                 }
81                 valid = !pred->marked && !succ->marked &&
82                     pred->nexsts[layer]==succ;
83             }
84             if (!valid) continue;
85
86             Node* newNode = new Node(v, topLayer);
87             for (int layer = 0;
88                  layer <= topLayer;
89                  layer++) {
90                 newNode->nexsts[layer] = succs[layer];
91                 preds[layer]->nexsts[layer] = newNode;
92             }
93
94             newNode->fullyLinked = true;
95             return true;
96         }
97         finally { unlock(preds, highestLocked); }
98     }

```

Figure 1.4. The add method

If no node was found with the appropriate key, then the thread locks and *validates* all the predecessors returned by `findNode` up to the height of the new node (lines 69–84). This height, denoted by `topNodeLayer`, is determined at the very beginning of the `add` operation using the `randomLevel` function.³ Validation (lines 81–83) checks that for each layer $i \leq \text{topNodeLayer}$, `preds[i]` and `succs[i]` are still adjacent at layer i , and that neither is marked. If validation fails, the thread encountered a conflicting operation, so it releases the locks it acquired (in the `finally` block at line 97) and retries.

If the thread successfully locks and validates the results of `findNode` up to the height of the new node, then the `add` operation is guaranteed to succeed because the thread holds all the locks until it fully links its new node. In this case, the thread allocates a new node with the appropriate key and height, links it in, sets the `fullyLinked` flag of the new node (this is the linearization point of the `add` operation), and then returns `true` after releasing all its locks (lines 86–97). The thread writing `newNode->nexts[i]` is the one case in which a thread modifies the `nexts` field for a node it has not locked. It is safe because `newNode` will not be linked into the list at layer i until the thread sets `preds[i]->nexts[i]` to `newNode`, *after* it writes `newNode->nexts[i]`.

3.2 The `remove` operation

The `remove` operation shown in Figure 1.5, likewise calls `findNode` to determine whether a node with the appropriate key is in the list. If so, the thread checks whether the node is “okay to delete” (Figure 1.6), which means it is fully linked, not marked, and it was found at its top layer.⁴ If the node meets these requirements, the thread locks the node and verifies that it is still not marked. If so, the thread marks the node, which logically deletes it (lines 111–121); that is, the marking of the node is the linearization point of the `remove` operation.

The rest of the procedure accomplishes the “physical” deletion, removing the node from the list by first locking its predecessors at all layers up to the height of the deleted node (lines 124–138), and splicing the node out one layer at a time (lines 140–142). To maintain the skip-list structure, the node is spliced out of higher layers before being spliced out of lower ones (though, to ensure freedom from deadlock, as discussed in Section 4, the locks are acquired in the opposite order, from lower layers up). As in the `add` operation, before changing any of the deleted node’s predecessors, the thread validates that those nodes are indeed still the deleted node’s predecessors. This is done using the `weakValidate` function, which is the same as `validate` except that it does not fail if the successor

³ This function is taken from Lea’s algorithm to ensure a fair comparison in the experiments presented in Section 5. It returns 0 with probability $\frac{3}{4}$, i with probability $2^{-(i+2)}$ for $i \in [1, 30]$, and 31 with probability 2^{-32} .

⁴ A node found not in its top layer was either not yet fully linked, or marked and partially unlinked, at some point when the thread traversed the list at that layer. We could have continued with the `remove` operation, but the subsequent validation would fail.

```

101 bool remove(int v) {
102     Node* nodeToDelete = null;
103     bool isMarked = false;
104     int topLayer = -1;
105     Node* preds[MaxHeight], succs[MaxHeight];
106     while (true) {
107         int lFound = findNode(v, preds, succs);
108         if (isMarked ||
109             (lFound != -1 && okToDelete(succs[lFound], lFound))) {
110
111             if (!isMarked) {
112                 nodeToDelete = succs[lFound];
113                 topLayer = nodeToDelete->topLayer;
114                 nodeToDelete->lock.lock();
115                 if (nodeToDelete->marked) {
116                     nodeToDelete->lock.unlock();
117                     return false;
118                 }
119                 nodeToDelete->marked = true;
120                 isMarked = true;
121             }
122             int highestLocked = -1;
123             try {
124                 Node *pred, *succ, *prevPred = null;
125                 bool valid = true;
126                 for (int layer = 0;
127                     valid && (layer ≤ topLayer);
128                     layer++) {
129                     pred = preds[layer];
130                     succ = succs[layer];
131                     if (pred ≠ prevPred) {
132                         pred->lock.lock();
133                         highestLocked = layer;
134                         prevPred = pred;
135                     }
136                     valid = !pred->marked && pred->nexsts[layer]==succ;
137                 }
138                 if (!valid) continue;
139
140                 for (int layer = topLayer; layer ≥ 0; layer--) {
141                     preds[layer]->nexsts[layer] = nodeToDelete->nexsts[layer];
142                 }
143                 nodeToDelete->lock.unlock();
144                 return true;
145             }
146             finally { unlock(preds, highestLocked); }
147         }
148         else return false;
149     }
150 }

```

Figure 1.5. The remove method

```

152 bool okToDelete(Node* candidate, int lFound) {
153     return (candidate->fullyLinked
154         && candidate->topLayer==lFound
155         && !candidate->marked);
156 }

```

Figure 1.6. The `okToDelete` method

```

158 bool contains(int v) {
159     Node* preds[MaxHeight], succs[MaxHeight];
160     int lFound = findNode(v, preds, succs);
161     return (lFound != -1
162             && succs[lFound]->fullyLinked
163             && !succs[lFound]->marked);
164 }

```

Figure 1.7. The `contains` method

is marked, since the successor in this case should be the node to be removed that was just marked. If the validation fails, then the thread releases the locks on the old predecessors (but not the deleted node) and tries to find the new predecessors of the deleted node by calling `findNode` again. However, at this point it has already set the local `isMarked` flag so that it will not try to mark another node. After successfully removing the deleted node from the list, the thread releases all its locks and returns `true`.

If no node was found, or the node found was not “okay to delete” (i.e., was marked, not fully linked, or not found at its top layer), then the operation simply returns `false` (line 148). It is easy to see that this is correct if the node is not marked because for any key, there is at most one node with that key in the skip list (i.e., reachable from the left sentinel) at any time, and once a node is put in the list (which it must have been to be found by `findNode`), it is not removed until it is marked. However, the argument is trickier if the node is marked, because at the time the node is found, it might not be in the list, and some unmarked node with the same key may be in the list. However, as we argue in Section 4, in that case, there must have been some time during the execution of the `remove` operation at which the key was not in the abstract set.

3.3 The `contains` operation

Finally, we consider the `contains` operation, shown in Figure 1.7, which just calls `findNode` and returns `true` if and only if it finds a unmarked, fully linked node with the appropriate key. If it finds such a node, then it is immediate from the definition that the key is in the abstract set. However, as mentioned above, if the node is marked, it is not so easy to see that it is safe to return `false`. We argue this in Section 4.

3.4 Implementation Issues

We implemented the algorithm in the JavaTM programming language, in order to compare it with Doug Lea’s nonblocking skip-list implementation in the `java.util.concurrent` package. The array stack variables in the pseudocode are replaced by thread-local variables, and we used a straightforward lock implementation (we could not use the built-in object locks because our acquire and release pattern could not always be expressed using synchronized blocks).

The pseudocode presented was optimized for simplicity, not efficiency, and there are numerous obvious ways in which it can be improved, many of which we applied to our implementation. For example, if a node with an appropriate key is found, the `add` and `contains` operations need not look further; they only need to ascertain whether that node is fully linked and unmarked. If so, the `contains` operation can return `true` and the `add` operation can return `false`. If not, then the `contains` operation can return `false`, and the `add` operation either waits before returning `false` (if the node is not fully linked) or else must retry. The `remove` operation does need to search to the bottom layer to find all the predecessors of the node to be deleted, however, once it finds and marks the node at some layer, it can search for that exact node at lower layers rather than comparing keys.⁵ This is correct because once a thread marks a node, no other thread can unlink it.

Also, in the pseudocode, `findNode` always starts searching from the highest possible layer, though we expect most of the time that the highest layers will be empty (i.e., have only the two sentinel nodes). It is easy to maintain a variable that tracks the highest nonempty layer because whenever that changes, the thread that causes the change must have the left sentinel locked. This ease is in contrast to the nonblocking version, in which a race between concurrent `remove` and `add` operations may result in the recorded height of the skip list being less than the actual height of its tallest node.

4 Correctness

In this section, we sketch a proof for our skip-list algorithm. There are four properties we want to show: that the algorithm implements a linearizable set, that it is deadlock-free, that the `contains` operation is wait-free, and that the underlying data structure maintains a correct skip-list structure, which we define more precisely below.

4.1 Linearizability

For the proof, we make the following simplifying assumption about initialization: Nodes are initialized with their key and height, their `nexts` arrays are initialized to all `null`, and their `fullyLinked` and `marked` fields are initialized to `false`.

⁵ Comparing keys is expensive because, to maintain compatibility with Lea’s implementation, comparison invokes the `compareTo` method of the `Comparable` interface.

Furthermore, we assume for the purposes of reasoning that nodes are never reclaimed, and there is an inexhaustible supply of new nodes (otherwise, we would need to augment the algorithm to handle running out of nodes).

We first make the following observations: The key of a node never changes (i.e., `key = k` is stable), and the `marked` and `fullyLinked` fields of a node are never set to `false` (i.e., `marked` and `fullyLinked` are stable). Though initially `null`, `nexts[i]` is never written to `null` (i.e., `nexts[i] ≠ null` is stable). Also, a thread writes a node's `marked` or `nexts` fields only if it holds the node's lock (with the one exception of an `add` operation writing `nexts[i]` of a node before linking it in at layer i).

From these observations, and by inspection of the code, it is easy to see that in any operation, after calling `findNode`, we have `preds[i] → key < v` and `succs[i] → key ≥ v` for all i , and `succs[i] → key > v` for $i > 1$ (the value returned by `findNode`). Also, for a thread in `remove`, `nodeToDelete` is only set once, and that unless that node was marked by some other thread, this thread will mark the node, and thereafter, until it completes the operation, the thread's `isMarked` variable will be `true`. We also know by `okToDelete` that the node is fully linked (and indeed that only fully linked nodes can be marked).

Furthermore, validation and the requirement to lock nodes before writing them ensures that after successful validation, the properties checked by the validation (which are slightly different for `add` and `remove`) remain true until the locks are released.

We can use these properties to derive the following fundamental lemma:

Lemma 1. *For a node n and $0 \leq i \leq n \rightarrow topLayer$:*

$$n \rightarrow nexts[i] \neq null \implies n \rightarrow key < n \rightarrow nexts[i] \rightarrow key$$

We define the relation \rightarrow_i so that $m \rightarrow_i n$ (read “ m leads to n at layer i ”) if $m \rightarrow nexts[i] = n$ or there exists m' such that $m \rightarrow_i m'$ and $m' \rightarrow nexts[i] = n$; that is, \rightarrow_i is the transitive closure of the relation that relates nodes to their immediate successors at layer i . Because a node has (at most) one immediate successor at any layer, the \rightarrow_i relation “follows” a linked list at layer i , and in particular, the layer- i list of the skip list consists of those nodes n such that `LSentinel →i n` (plus `LSentinel` itself). Also, by Lemma 1, if $m \rightarrow_i n$ and $m \rightarrow_i n'$ and $n \rightarrow key < n' \rightarrow key$ then $n \rightarrow_i n'$.

Using these observations, we can show that if $m \rightarrow_i n$ in any reachable state of the algorithm, then $m \rightarrow_i n$ in any subsequent state unless there is an action that splices n out of the layer- i list, that is, an execution of line 141. This claim is proved formally for the lazy-list algorithm in a recent paper [1], and that proof can be adapted to this algorithm. Because n must already be marked before being spliced out of the list, and because the `fullyLinked` flag is never set to `false` (after its initialization), this claim implies that a key can be removed from the abstract set only by marking its node, which we argued earlier is the linearization point of a successful `remove` operation.

Similarly, we can see that if `LSentinel →i n` does *not* hold in some reachable state of the algorithm, then it does not hold in any subsequent state unless there

is some execution of line 91 with $n = \text{newNode}$ (as discussed earlier, the previous line doesn't change the list at layer- i because `newNode` is not yet linked in then). However, the execution of that line occurs while `newNode` is being inserted and before `newNode` is fully linked. Thus, the only action that adds a node to a list at any level is the setting of the node's `fullyLinked` flag.

Finally, we argue that if a thread finds a marked node then the key of that node must have been absent from the list at some point during the execution of the thread's operation. There are two cases: If the node was marked when the thread invoked the operation, the node must have been in the skip list at that time because marked nodes cannot be added to the skip list (only a newly allocated node can be added to the skip list), and because no two nodes in the skip list can have the same key, no unmarked node in the skip list has that key. Thus, at the invocation of the operation, the key is not in the skip list. On the other hand, if the node was not marked when the thread invoked the operation, then it must have been marked by some other thread before the first thread found it. In this case, the key is not in the abstract set immediately after the other thread marked the node. This claim is also proved formally for the simple lazy list [1], and that proof can be adapted to this algorithm.

4.2 Maintaining the skip-list structure

Our algorithm guarantees that the skip-list structure is preserved at all times. By "skip-list structure", we mean that the list at each layer is a sublist of the lists at lower layers. It is important to preserve this structure, as the complexity analysis for skip lists requires this structure.

To see that the algorithm preserves the skip-list structure, note that linking new nodes into the skip list always proceeds from bottom to top, and while holding the locks on all the soon-to-be predecessors of the node being inserted. On the other hand, when a node is being removed from the list, the higher layers are unlinked before the lower layers, and again, while holding locks on all the immediate predecessors of the node being removed.

This property is not guaranteed by the lock-free algorithm. In that algorithm, after linking a node in the bottom layer, links the node in the rest of the layers from top to bottom. This may result in a state of a node that is linked only in its top and bottom layers, so that the list at the top layer is *not* a sublist of the list at the layer immediately beneath it, for example. Moreover, attempts to link in a node at any layer other than the bottom are not retried, and hence this state of nonconformity to the skip-list structure may persist indefinitely.

4.3 Deadlock freedom and wait-freedom

The algorithm is deadlock-free because a thread always acquires locks on nodes with larger keys first. More precisely, if a thread holds a lock on a node with key v then it will not attempt to acquire a lock on a node with key greater than or equal to v . We can see that this is true because both the `add` and `remove` methods acquire locks on the predecessor nodes from the bottom layer up, and

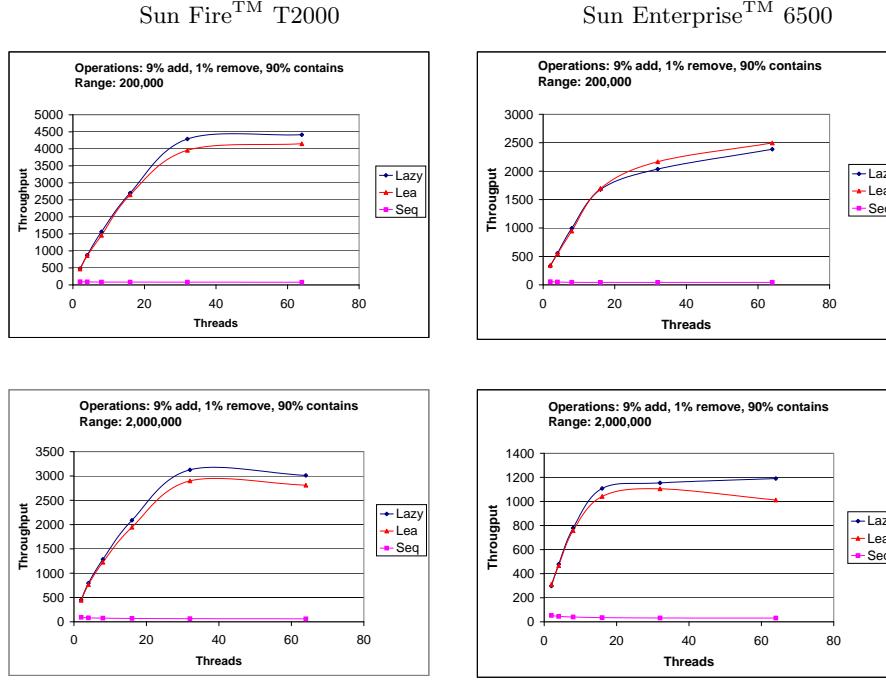


Fig. 8. Throughput in operations per millisecond of 1,000,000 operations, with 9% `add`, 1% `remove`, and 90% `contains` operations, and a range of either 200,000 or 2,000,000.

the key of a predecessor node is less than the key of a different predecessor node at a lower layer. The only other lock acquisition is for the node that a `remove` operation deletes. This is the first lock acquired by that operation, and its key is greater than that of any of its predecessors.

That the `contains` operation is wait-free is also easy to see: it does not acquire any locks, nor does it ever retry; it searches the list only once.

5 Performance

We evaluated our skip-list algorithm by implementing in the Java programming language, as described earlier. We compared our implementation against Doug Lea's nonblocking skip-list implementation in the `ConcurrentSkipListMap` class of the `java.util.concurrent` package, which is part of the JavaTM SE 6 platform; to our knowledge, this is the best widely available concurrent skip-list implementation. We also implemented a straightforward sequential skip list, in which methods were `synchronized` to ensure thread safety, for use as a baseline in these experiments. We describe some of the results we obtained from these experiments in this section.

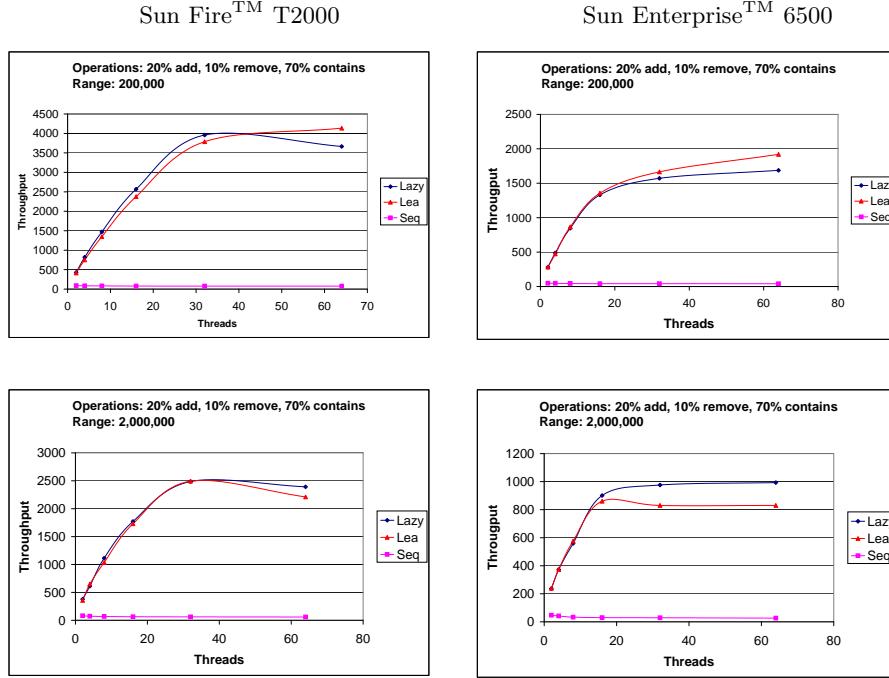


Fig. 9. Throughput in operations per millisecond of 1,000,000 operations with 20% `add`, 10% `remove`, and 70% `contains` operations, and range of either 200,000 or 2,000,000.

We present results from experiments on two multiprocessor systems with quite different architectures. The first system is a Sun Fire™ T2000 server, which is based on a single UltraSPARC® T1 processor containing eight computing cores, each with four hardware strands, clocked at 1200 MHz. Each four-strand core has a single 8-KByte level-1 data cache and a single 16-KByte instruction cache. All eight cores share a single 3-MByte level-2 unified (instruction and data) cache, and a four-way interleaved 32-GByte main memory. Data access latency ratios are approximately 1:8:50 for L1:L2:Memory accesses. The other system is an older Sun Enterprise™ 6500 server, which contains 15 system boards, each with two UltraSPARC® II processors clocked at 400 MHz and 2 Gbytes of RAM for a total of 30 processors and 60 Gbytes of RAM. Each processor has a 16-KByte data level-1 cache and a 16-Kbyte instruction cache on chip, and a 8-MByte external cache. The system clock frequency is 80 MHz.

We present results from experiments in which, starting from an empty skip list, each thread executes one million (1,000,000) randomly chosen operations. We varied the number of threads, the relative proportion of `add`, `remove` and `contains` operations, and the range from which the keys were selected. The key for each operation was selected uniformly at random from the specified range.

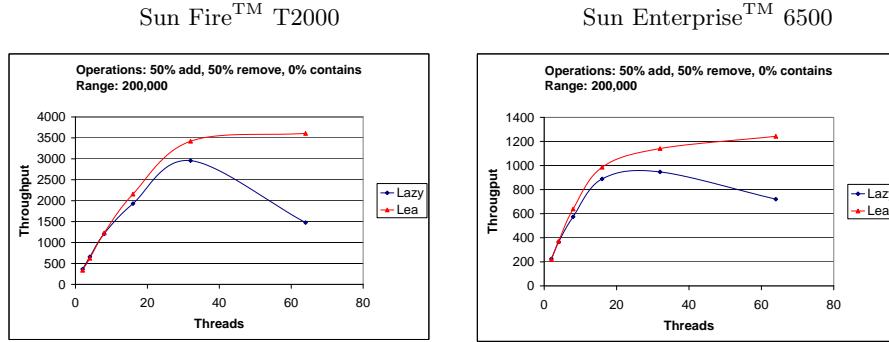


Fig. 10. Throughput in operations per millisecond of 1,000,000 operations, with 50% `add` and 50% `remove` operations, and a range of 200,000

In the graphs that follow, we compare the throughput in operations per millisecond, and the results shown are the average over six runs for each set of parameters.

Figure 8 presents the results of experiments in which 9% of the operations were `add` operations, 1% were `remove` operations, and the remaining 90% were `contains` operations, where the range of the keys was either two hundred thousand or two million. The different ranges give different levels of contention, with significantly higher contention with the 200,000 range, compared with the 2,000,000 range. As we can see from these experiments, both our implementation and Lea's scale well (and the sequential algorithm, as expected, is relatively flat). In all but one case (with 200,000 range on the older system), our implementation has a slight advantage.

In the next set of experiments, we ran with higher percentages of `add` and `remove` operations, 20% and 10% respectively (leaving 70% `contains` operations). The results are shown in Figure 9. As can be seen, on the T2000 system, the two implementations have similar performance, with a slight advantage to Lea in a multiprogrammed environment when the range is smaller (higher contention). The situation is reversed with the larger range. This phenomenon is more noticeable on the older system: there we see a 13% advantage to Lea's implementation on the smaller range with 64 threads, and 20% advantage to our algorithm with the same number of threads when the range is larger.

To explore this phenomenon, we conducted an experiment with a significantly higher level of contention: half `add` operations and half `remove` operations with a range of 200,000. The results are presented in Figure 10. As can be clearly seen, under this level of contention, our implementation's throughput degrades rapidly when approaching the multiprogramming zone, especially on the T2000 system. This degradation is not surprising: In our current implementation, when an `add` or `remove` operation fails validation, or fails to acquire a lock immediately, it simply calls `yield`; there is no proper mechanism for managing contention.

Since the `add` and `remove` operations require that the predecessors seen during the search phase be unchanged until they are locked, we expect that under high contention, they will repeatedly fail. Thus, we expect that a back-off mechanism, or some other means of contention control, would greatly improve performance in this case. To verify that a high level of conflict is indeed the problem, we added counters to count the number of retries executed by each thread during the experiment. The counters indeed show that many retries are executed on a 64 threads run, especially on the T2000. Most of the retries are executed by the `add` method, which makes sense because the `remove` method marks the node to be removed before searching its predecessors in lower layers, which prevents change of these predecessor's `next` pointers by a concurrent `add` operation.

6 Conclusions

We have shown how to construct a scalable, highly concurrent skip list using a remarkably simple algorithm. The principal open question is whether we can improve the algorithm's performance at high levels of contention. One simple approach is to use a more sophisticated back-off scheme when synchronization conflicts are detected. Another intriguing approach is to allow the randomness of the skip-list's height to be compromised under high contention. In the algorithm presented here, when a thread adds a new item, it gives up and retries if it encounters a synchronization conflict when linking the item at any level. In principle, a thread that encounters a conflict when linking the item at layer $\ell > 0$ could simply stop there, leaving the item linked at levels zero to $\ell - 1$, acting as if it had randomly chosen $\ell - 1$. The resulting data structure, while structurally still a skip list, would be a little flatter than it should be. Whether this approach is effective is the subject of future work.

References

- COLVIN, R., GROVES, L., LUCHANGCO, V., AND MOIR, M. Formal verification of a lazy concurrent list-based set. In *Proceedings of Computer-Aided Verification* (Aug. 2006).
- FRASER, K. *Practical Lock-Freedom*. PhD thesis, University of Cambridge, 2004.
- HELLER, S., HERLIHY, M., LUCHANGCO, V., MOIR, M., SHAVIT, N., AND SCHERER III, W. N. A lazy concurrent list-based set algorithm. In *Proceedings of 9th International Conference on Principles of Distributed Systems* (Dec. 2005).
- HERLIHY, M., LUCHANGCO, V., AND MOIR, M. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures. In *Proceedings of Distributed Computing: 16th International Conference* (2002).
- HERLIHY, M., AND WING, J. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems* 12, 3 (July 1990), 463–492.
- MICHAEL, M. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems* 15, 6 (June 2004), 491–504.
- PUGH, W. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM* 33, 6 (June 1990), 668–676.

A Scalable Lock-free Stack Algorithm

Danny Hendler

Ben-Gurion University

Nir Shavit

Tel-Aviv University

Lena Yerushalmi

Tel-Aviv University

The literature describes two high performance concurrent stack algorithms based on combining funnels and elimination trees. Unfortunately, the funnels are linearizable but blocking, and the elimination trees are non-blocking but not linearizable. Neither is used in practice since they perform well only at exceptionally high loads. The literature also describes a simple lock-free linearizable stack algorithm that works at low loads but does not scale as the load increases. The question of designing a stack algorithm that is non-blocking, linearizable, and scales well throughout the concurrency range, has thus remained open.

This paper presents such a concurrent stack algorithm. It is based on the following simple observation: that a single elimination array used as a backoff scheme for a simple lock-free stack is lock-free, linearizable, and scalable. As our empirical results show, the resulting *elimination-backoff stack* performs as well as the simple stack at low loads, and increasingly outperforms all other methods (lock-based and non-blocking) as concurrency increases. We believe its simplicity and scalability make it a viable practical alternative to existing constructions for implementing concurrent stacks.

⁰A preliminary version of this paper appeared in the proceedings of the *16th Annual ACM Symposium on Parallelism in Algorithms*, Barcelona, Spain, 2004, pages 206-215.

1. INTRODUCTION

Shared stacks are widely used in parallel applications and operating systems. As shown in [28], LIFO-based scheduling not only reduces excessive task creation, but also prevents threads from attempting to dequeue and execute a task which depends on the results of other tasks. A concurrent shared stack is a data structure that supports the usual **push** and **pop** operations with linearizable LIFO semantics. Linearizability [16] guarantees that operations appear atomic and can be combined with other operations in a modular way.

When threads running a parallel application on a shared memory machine access the shared stack object simultaneously, a synchronization protocol must be used to ensure correctness. It is well known that concurrent access to a single object by many threads can lead to a degradation in performance [1; 14]. Therefore, in addition to correctness, synchronization methods should offer efficiency in terms of scalability, and robustness [12] in the face of scheduling constraints. Scalability at high loads should not, however, come at the price of good performance in the more common low contention cases.

Unfortunately, the two known methods for parallelizing shared stacks do not meet these criteria. The combining funnels of Shavit and Zemach [27] are linearizable LIFO stacks that offer scalability through combining, but perform poorly at low loads because of the combining overhead. They are also blocking and thus not robust in the face of scheduling constraints [18]. The elimination trees of Shavit and Touitou [24] are non-blocking and thus robust, but the stack they provide is not linearizable, and it too has large overheads that cause it to perform poorly at low loads. On the other hand, the results of Michael and Scott [21] show that the best known low load method, the simple linearizable lock-free stack introduced by IBM [17] (a variant of which was later presented by Treiber [29]), scales poorly due to contention and an inherent sequential bottleneck.

This paper presents the *elimination backoff stack*, a new concurrent stack algorithm that overcomes the combined drawbacks of all the above methods. The algorithm is linearizable and thus easy to modularly combine with other algorithms; it is lock-free and hence robust; it is parallel and hence scalable; and it utilizes its parallelization construct adaptively, which allows it to perform well at low loads. The *elimination backoff stack* is based on the following simple observation: that a single elimination array [24], used as a backoff scheme for a lock-free stack [17], is both lock-free and linearizable. The introduction of elimination into the backoff process serves a dual purpose of adding parallelism and reducing contention, which, as our empirical results show, allows the *elimination-backoff stack* to outperform all algorithms in the literature at both high and low loads.

We believe its simplicity and scalability make it a viable practical alternative to existing constructions for implementing concurrent stacks.

1.1 Background

Generally, algorithms for concurrent data structures fall into two categories: blocking and non-blocking. There are several lock-based concurrent stack implementations in the literature. Typically, lock-based stack algorithms are expected to offer limited robustness as they are susceptible to long delays and priority inversions [10].

The first non-blocking implementation of a concurrent list-based stack appeared in the IBM System 370 principles of operation manual in 1983 [17] and used the double-width compare-and-swap (CAS) primitive. A variant of that algorithm in

which push operations use unary CAS instead of double-width compare-and-swap appeared in a report by Treiber in 1986 [29]. We henceforth refer to that algorithm as the IBM/Treiber algorithm. The IBM/Treiber algorithm represented a stack as a singly-linked list with a top pointer and used (either unary or double) compare-and-swap (CAS) to modify the value of the top atomically. No performance results were reported by Treiber for his non-blocking stack. Michael and Scott in [21] compared the IBM/Treiber stack to an optimized non-blocking algorithm based on Herlihy’s general methodology [13], and to lock-based stacks. They showed that the IBM/Treiber algorithm yields the best overall performance, and that the performance gap increases as the amount of multiprogramming in the system increases. However, from their performance data it is clear that because of its inherent sequential bottleneck, the IBM/Treiber stack offers little scalability.

Shavit and Touitou [24] introduced elimination trees, scalable tree like data structures that behave “almost” like stacks. Their elimination technique (which we will elaborate on shortly as it is key to our new algorithm) allows highly distributed coupling and execution of operations with reverse semantics like the pushes and pops on a stack. Elimination trees are lock-free, but not linearizable. In a similar fashion, Shavit and Zemach introduced combining funnels [27], and used them to provide scalable stack implementations. Combining funnels employ both combining [8; 9] and elimination [24] to provide scalability. They improve on elimination trees by being linearizable, but unfortunately they are blocking. As noted earlier, both [24] and [27] are directed at high-end scalability, resulting in overheads which severely hinder their performance under low loads.

The question of designing a practical lock-free linearizable concurrent stack that will perform well at both high and low loads has thus remained open.

1.2 The New Algorithm

Consider the following simple observation due to Shavit and Touitou [24]: if a **push** followed by a **pop** are performed on a stack, the data structure’s state does not change. This means that if one can cause pairs of pushes and pops to meet and pair up in separate locations, the threads can exchange values without having to touch a centralized structure since they have “eliminated” each other’s effect on it. Elimination can be implemented by using a collision array in which threads pick random locations in order to try and collide. Pairs of threads that “collide” in some location run through a lock-free synchronization protocol, and all such disjoint collisions can be performed in parallel. If a thread has not met another in the selected location or if it met a thread with an operation that cannot be eliminated (such as two **push** operations), an alternative scheme must be used. In the elimination trees of [24], the idea is to build a tree of elimination arrays and use the diffracting tree paradigm of Shavit and Zemach [26] to deal with non-eliminated operations. However, as we noted, the overhead of such mechanisms is high, and they are not linearizable.

The new idea (see Figure 1) in this paper is strikingly simple: use a single elimination array as a backoff scheme on a shared lock-free stack. If the threads fail on the stack, they attempt to eliminate on the array, and if they fail in eliminating, they attempt to access the stack again and so on. The surprising result is that this structure is linearizable: any operation on the shared stack can be linearized at the access point, and any pair of eliminated operations can be linearized when they meet.

Because it is a backoff scheme, it delivers the same performance as the simple

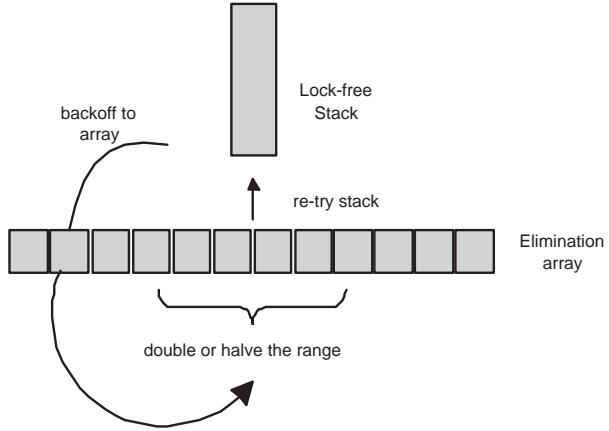


Fig. 1. Schematic depiction of the elimination-backoff cycle.

stack at low loads. However, unlike the simple stack it scales well as load increases because: (1) the number of successful eliminations grows, allowing many operations to complete in parallel; and (2) contention on the head of the shared stack is reduced beyond levels achievable by the best exponential backoff schemes [1] since scores of backed off operations are eliminated in the array and *never* re-attempt to access the shared structure.

1.3 Performance

We compared our new *elimination-backoff stack* algorithm to a lock-based implementation using Mellor-Crummey and Scott's MCS-lock [19] and to several non-blocking implementations: the linearizable IBM/Treiber [17; 29] algorithm with and without backoff, and the elimination tree of Shavit and Touitou [24]. Our comparisons were based on a collection of synthetic microbenchmarks executed on a 14-node shared memory machine. Our results, presented in Section 4, show that the elimination-backoff stack outperforms all three methods, and specifically the two lock-free methods, exhibiting almost three times the throughput at peak load. Unlike the other methods, it maintains constant latency throughout the concurrency range, and performs well also in experiments with unequal ratios of pushes and pops.

The remainder of this paper is organized as follows. In the next section we describe the new algorithm in depth. In Section 3, we give the sketch of adaptive strategies we used in our implementation. In Section 4, we present our empirical results. In Section 5, we provide a proof that our algorithm has the required properties of a stack, is linearizable, and lock-free. We conclude with a discussion in Section 6.

2. THE ELIMINATION BACKOFF STACK

2.1 Data Structures

We now present our elimination backoff stack algorithm. Figure 2 specifies some type definitions and global variables.

```

struct Cell {
    Cell *pnex;
    void *pdata;
};

struct Simple_Stack {
    Cell *ptop;
};

struct AdaptParams {
    int count;
    float factor;
};

struct ThreadInfo {
    u_int id;
    char op;
    Cell *cell;
    AdaptParams *adapt;
};

Simple_Stack S;
void **location;
int *collision;

```

Fig. 2. Types and Structures

Our central stack object follows IBM/Treiber [17; 29] and is implemented as a singly-linked list with a top pointer. The elimination layer follows Shavit and Touitou [24] and is built of two arrays: a global `location[1..n]` array has an element per thread $t \in \{1..n\}$, holding the pointer to the `ThreadInfo` structure, and a global `collision[1..size]` array, that holds the IDs of the threads trying to collide. The elements of both these arrays are initialized to `NULL`. Each `ThreadInfo` record contains the thread id, the type of the operation to be performed by the thread (`push` or `pop`), a pointer to the cell for the operation, and a pointer to the `adapt` structure that is used for dynamic adaptation of the algorithm's behavior (see Section 3).

2.2 Elimination Backoff Stack Code

We now provide the code of our algorithm. It is shown in Figures 3 and 4. As can be seen from the code, first each thread tries to perform its operation on the central stack object (line P1). If this attempt fails, a thread goes through the collision layer in the manner described below.

Initially, thread t announces its arrival at the collision layer by writing its current information to the `location` array (line S2). It then chooses a random location in the `collision` array (line S3). Thread t reads into `him` the id of the thread written at `collision[pos]` and tries to write its own id in place (lines S4 and S5). If it fails, it retries until it succeeds (lines S5 and S6).

After that, there are three main scenarios for thread actions, according to the information the thread has read. They are illustrated in Figure 5. If t reads an id of another thread q (i.e., `him!=EMPTY`), t attempts to collide with q . The collision

```

void StackOp(ThreadInfo* p) {
P1: if(TryPerformStackOp(p)==FALSE)
P2:   LesOP(p);
P3:   return;
}
void LesOP(ThreadInfo *p) {
S1: while (1) {
S2:   location[mypid]=p;
S3:   pos=GetPosition(p);
S4:   him=collision[pos];
S5:   while(!CAS(&collision[pos],him,mypid))
S6:     him=collision[pos];
S7:   if (him!=EMPTY) {
S8:     q=location[him];
S9:     if(q!=NULL&&q->id==him&&q->op!=p->op) {
S10:       if(CAS(&location[mypid],p,NULL)) {
S11:         if(TryCollision(p,q)==TRUE)
S12:           return;
S13:         else
S14:           goto stack;
}
S15:       else {
S16:         FinishCollision(p);
S17:         return
}
S18:     delay(spin);
S19:     AdaptWidth(SHRINK);
S20:     if (!CAS(&location[mypid],p,NULL)) {
S21:       FinishCollision(p);
S22:       return;
}
stack:
S23:   if (TryPerformStackOp(p)==TRUE)
S24:     return;
}
}
}

boolean TryPerformStackOp(ThreadInfo* p){
Cell *ptop,*pnxt;
T1: if(p->op==PUSH) {
T2:   ptop=S.ptop;
T3:   p->cell->pnext=ptop;
T4:   if(CAS(&S.ptop,ptop,p->cell))
T5:     return TRUE;
T6:   else
T7:     return FALSE; }
T8: if(p->op==POP) {
T9:   ptop=S.ptop;
T10:  if(ptop==NULL) {
T11:    p->cell=EMPTY;
T12:    return TRUE;
}
T13:  pnxt=ptop->pnext;
T14:  if(CAS(&S.ptop,ptop,pnxt))
T15:    p->cell=ptop;
T16:    return TRUE;
}
T17:  else
T18:    return FALSE; }
}

```

Fig. 3. Elimination Backoff Stack Code - part 1

is accomplished by t first executing a read operation (line S8) to determine the type of the thread being collided with. As two threads can collide only if they have opposing operations, if q has the same operation as t , t waits for another collision (line S18). If no other thread collides with t during its waiting period, t calls the *AdaptWidth* procedure (line S19) that dynamically changes the width of t 's collision layer according to the perceived contention level (see Section 3). Thread t then clears its entry in the *location* array (line S20), and tries once again to perform its operation on the central stack object (line S23). If p 's entry cannot be cleared, it follows that t has been collided with, in which case t completes its operation and returns.

If q does have a complementary operation, t tries to eliminate by performing two CAS operations on the *location* array. The first (line S10) clears t 's entry, assuring no other thread will collide with it during its collision attempt (this eliminates race

```

void TryCollision(ThreadInfo*p,ThreadInfo *q) {
C1: if(p->op==PUSH) {
C2:   if(CAS(&location[him],q,p))
C3:     return TRUE;
C4:   else
C5:     adaptWidth(ENLARGE)
C6:   return FALSE;
C7: }
C8: if(p->op==POP) {
C9:   if(CAS(&location[him],q,NULL)){
C10:    p->cell=q->cell;
C11:    location[mypid]=NULL;
C12:    return TRUE
C13:  }
C14: else
C15:  adaptWidth(ENLARGE)
C16: return FALSE;
C17: }
}
}

void FinishCollision(ThreadInfo *p) {
F1: if (p->op==POP_OP) {
F2:   p->cell=location[mypid]->cell;
F3:   location[mypid]=NULL;
F4: }
}
}

```

Fig. 4. Elimination Backoff Stack Code - part 2

conditions). The second (lines C2 or C8, see Figure 4) attempts to mark q 's entry as “collided with t ”. If both CAS operations succeed, the collision is successful. Therefore t can return (in case of a pop operation it stores the value of the popped cell).

If the first CAS fails, it follows that some other thread has already managed to collide with t . In this case, thread t acts as in case of a successful collision, mentioned above. If the first CAS succeeds but the second fails, then the thread with which t is trying to collide is no longer available for collision. In that case, t calls the *AdaptWidth* procedure (lines C5 or C13, see section 3), and then tries once more to perform its operation on the central stack object; t returns in case of success, and repeatedly goes through the collision layer in case of failure.

2.3 Memory Management and ABA Issues

As our algorithm is based on the compare-and-swap (CAS) operation, it must deal with the “ABA problem” [17]: if a thread reads the top of the stack, computes a new value, and then attempts a CAS on the top of the stack, the CAS may succeed when it should not, if between the read and the CAS some other thread changes the value to the previous one again. Similar scenarios are possible with CAS operations

that access the `location` array.

Since the only dynamic-memory structures used by our algorithms are Cell and ThreadInfo structures, the ABA problem can be prevented by making sure that no Cell or ThreadInfo structure is recycled (freed and returned to a pool so that it can be used again) while a thread performing a stack operation holds a pointer to it which it will later access.

Some runtime environments, such as that provided by Java®, implement automatic mechanisms for dynamic memory reclamation. Such environments seamlessly prevent ABA problems in algorithms such as ours. In environments that do not implement such mechanisms, the simplest and most common ABA-prevention technique is to include a tag with the target memory location, such that both the application value and the tag are manipulated together atomically, and the tag is incremented with each update of the target memory location [17]. The CAS operation is sufficient for such manipulation, as most current architectures that support CAS (Intel x86, Sun SPARC®) support their operation on aligned 64-bit blocks. One can also use general techniques to eliminate ABA issues through memory management schemes such as Safe Memory Reclamation (SMR) [20] or ROP [15].

We now provide a brief description of the SMR technique and then describe in detail how it can be used to eliminate ABA issues in our algorithm.

2.3.1 ABA Elimination Using SMR. The SMR technique [20] uses *hazard pointers* for implementing safe memory reclamation for lock-free recyclable structures. Hazard pointers are single-writer multi-reader registers. The key idea is to associate a (typically small) number of hazard pointers with each thread that intends to access lock-free recyclable structures. A hazard pointer is either null or points to a structure that may be accessed later by that thread without further validation that the structure was not recycled in the interim; such an access is called a *hazardous reference*.

Hazard pointers are used in the following manner. Before making an hazardous reference, the address of the structure about to be referenced is written to an available hazard pointer; this assignment announces to other threads that the structure about to be accessed must not be recycled. Then, the thread must validate that the structure about to be accessed is still *safe*, that is, that it is still logically part of the algorithm's data-structure. If this is not the case, then the access is not made; otherwise, the thread is allowed to access the structure. After the hazardous access is made, the thread can safely nullify the hazard pointer or re-use it to protect against additional hazardous references.

When a thread logically removes a structure from the algorithm's data-structure, it calls the *RetireNode* method with a pointer to that structure. The structure will not be recycled, however, until after no hazard pointers are pointing to it. The *RetireNode* method scans a list of structures that were previously removed from the data-structure and only recycles structures that are no longer pointed at by hazard pointers. For more details, please refer to [20].

The SMR technique can be used by our algorithm as follows. Each thread maintains two pools: a pool of Cell structures and a pool of ThreadInfo structures. In addition, a single hazard pointer is allocated per thread. We now describe the changes that are introduced to the code of Figures 3, 4 for ensuring safe memory reclamation.

It is easily seen that the only hazardous references that exist in the code are CAS

operations that attempt to swap structure pointers. Specifically, these are lines T4, T14, S10, S20, C2 and C8. This is how these hazardous references are dealt with.

- (1) Line T4: Immediately after line T2, the executing thread's hazard pointer is set to point to *p*. Then, the condition *S.ptop==p* is checked. If the condition is not satisfied, the procedure returns FALSE, since the CAS of line T4 must fail. Otherwise the code proceeds as in the pseudo-code of Figure 3. It is now ensured that the structure pointed at by *p* cannot be recycled before the CAS of T4 is performed.
- (2) Line T14: Immediately before line T13, the executing thread's hazard pointer is set to point to *p*. Then, the condition *S.ptop==p* is checked. If the condition is not satisfied, the procedure returns FALSE, since the CAS of line T14 must fail. Otherwise the code proceeds as in the pseudo-code of Figure 3. It is now ensured that the structure pointed at by *p* cannot be recycled before the CAS of T14 is performed.
- (3) Line S10: Although the CAS of line S10 is technically a hazardous reference, there is no need to protect it: if the ThreadInfo structure pointed by *p* is recycled between when the executing thread *t* starts performing the **LesOP** procedure and when it performs the CAS of line S10, then it must be that another operation collided with this operation, changed the corresponding value of the location array to a value other than *p*, and nullified the entry of the collision array where *t*'s ID was written. Since no other **LesOP** operation can write *t*'s ID again to the collision array before *t* performs the CAS of line S10, no operation will write to the entry of the *location* array corresponding to *t* before *t* executes the CAS of line S10. Hence this CAS will fail.
- (4) Line S20: By exactly the same rationale applied in the case of line S10, there is no need to protect line S20.
- (5) Lines C2, C8: Immediately after line S8, the executing thread's hazard pointer is set to point to *q*. Then, the condition *q==location[him]* is checked. If the condition is not satisfied, the collision attempt has failed and a **goto stack** command is executed. Otherwise the procedure proceeds as in the pseudo-code of Figure 3. We are now ensured that the structure pointed at by *q* cannot be recycled before the CAS of line C2 or C8 is performed.

The following additional changes are introduced in the code of Figures 3, 4 for correct memory recycling.

- (6) Before starting a *push* operation, a ThreadInfo structure *i* is removed from the ThreadInfo pool and a Cell structure *c* is removed from the Cell pool. Both are initialized, and *i->cell* is initialized to point to *c*.
- (7) After successfully pushing a Cell to the central stack in line T4 and before returning, the ThreadInfo structure used by the operation is initialized and returned to the ThreadInfo pool from which it originated. There is no need to call the *RetireNode* method here, since at this point the executing thread's ThreadInfo structure is not accessible for collisions.
- (8) After returning from a *pop* operation on an empty stack (line T12), the ThreadInfo and Cell structures used by the *pop* operation are initialized and returned to the respective pools from which they originated. There is no need to call the *RetireNode* method here, since at this point the ThreadInfo structure is not accessible for collisions and hence also the Cell structure will not be accessed.

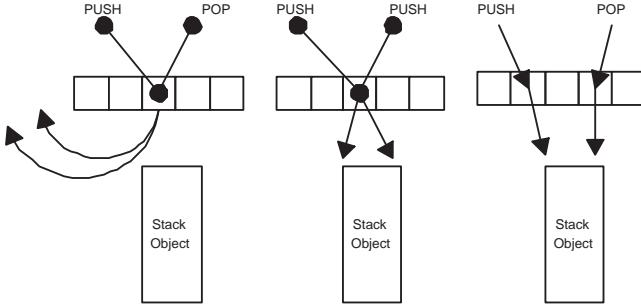


Fig. 5. Collision scenarios

- (9) After completing a *pop* operation that returned a cell (in line S12, S17 or T16), and upon consuming the application value pointed at from that cell, The *RetireNode* method is called to indicate that both the ThreadInfo structure used by the *pop* operation and the Cell structure it returned are now logically not part of the data-structure.

From [20], Theorem 1, we get:

LEMMA 1. *The following holds for all algorithm lines that perform a CAS operation to swap a structure pointer (lines T4, T14, S10, S20, C2 and C8): a structure is not recycled between when its old value is read and when the CAS operation using this old value is performed.*

Quoting [14]: "the expected amortized time complexity of processing each retired node until it is eligible for reuse is constant". Thus the expected effect of the SMR technique on performance is small. Specifically, in the absence of contention, time complexity remains constant.

3. ADAPTATIVE ELIMINATION BACKOFF

The classical approach to handling load is backoff, and specifically exponential backoff [1]. In a regular backoff scheme, once contention is detected on the central stack, threads back off in time. Here, threads will back off in both *time* and *space*, in an attempt to both reduce the load on the centralized data structure and to increase the probability of concurrent colliding. Our backoff parameters are thus the width of the collision layer, and the delay at the layer.

The elimination backoff stack has a simple structure that naturally fits with a localized *adaptive* policy for setting parameters similar to the strategy used by Shavit and Zemach for combining funnels in [27]. Decisions on parameters are made locally by each thread, and the collision layer does not actually grow or shrink. Instead, each thread independently chooses a sub-range of the collision layer it will map into, centered around the middle of the array, and limited by the maximal array width. It is possible for threads to have different ideas about the collision layer's width, and particularly bad scenarios might theoretically lead to bad performance, but as we will show, in practice the overall performance is superior to that of exponential backoff schemes [1]. Our policy is to first attempt to access the central stack object, and only if that fails to back off to the elimination

array. This allows us, in case of low loads, to avoid the collision array altogether, thus achieving the latency of a simple stack (in comparison with this, [27] are at best three times slower than a simple stack).

Our algorithm adaptively changes the width of the collision layer in the following way. Each thread t keeps a local variable called **factor**, $0 < \text{factor} \leq 1$, by which it multiplies the collision layer width to choose the interval into which it will randomly map to try and collide (e.g., if **factor**=0.5 only half the width is used). The dynamic adaptation of the collision layer's width is performed by the **AdaptWidth** procedure (see Figure 6). If a thread t fails to collide because it did not encounter any other thread, then it calls the **AdaptWidth** procedure with the **SHRINK** parameter to indicate this situation (line S19). In this case, the **AdaptWidth** procedure decrements a counter local to thread t . If the counter reaches 0, it is reset to its initial value (line A5), and the width of t 's collision layer is being halved (line A6). The width of the collision layer is not allowed to decrease bellow the **MIN_FACTOR** parameter. The rationale of shrinking the collision layer's width is the following. If t fails to perform its operation on the central stack object, but does not encounter other threads to collide with, then t 's collision layer's width should eventually be decreased so as to increase the probability of a successful collision.

In a symmetric manner, if a thread t does encounter another thread, but fails to collide with it because of contention, then t calls the **AdaptWidth** procedure with the **ENLARGE** parameter to indicate this situation (lines C5, C13). In this case, the **AdaptWidth** procedure increments the local counter. If it reaches the value **MAX_COUNT**, it is reset to its initial value (line A10), and the width of t 's collision layer is being doubled (line A11). The width of the collision layer is not allowed to surpass the **MAX_FACTOR** parameter. The rational of enlarging the collision layer's width is the following. If t fails to collide with some thread u because u collides with some other thread v , then the width of t 's collision layer should eventually be enlarged so as to decrease the contention on t 's collision layer and increase the probability of a successful collision. Finally, the **GetPosition** procedure, called on line S3, uses the value of the thread's **factor** variable to compute the current width of its collision layer.

```

void AdaptWidth(enum direction) {
A1:  if (direction==SHRINK)
A2:    if (p->adapt->count > 0)
A3:      p->adapt->count--
A4:    else {
A5:      p->adapt->count=ADAPT_INIT
A6:      p->adapt->factor=max(p->adapt->factor/2, MIN_FACTOR);
      }
A7:  else if (p->adapt->count < MAX_COUNT)
A8:    p->adapt->count++;
A9:  else {
A10:   p->adapt->count=ADAPT_INIT;
A11:   p->adapt->factor=min(2*p->adapt->factor, MAX_FACTOR);
      }
}

```

Fig. 6. Pseudo-code of the **AdaptWidth** procedure. This procedure dynamically changes the width of a thread's collision layer according to the level of contention it encounters when trying to collide.

The second part of our strategy is the dynamic update of the delay time for attempting to collide in the array, a technique used by Shavit and Zemach for diffracting trees in [25; 26]. This is being done by the `delay` function called at line S18, which simply implements exponential backoff. Note, that whereas the changes in the collision layer width are kept between invocations of `StackOp`, the updates to the delay time are internal to `StackOp`, and so the delay time is reset to its default value whenever `StackOp` is called.

There are obviously other conceivable ways of adaptively updating these two parameters and this is a subject for further research.

4. PERFORMANCE

We evaluated the performance of our *elimination-backoff stack* algorithm relative to other known methods by running a collection of synthetic benchmarks on a 14 node Sun Enterprise™ E6500, an SMP machine formed from 7 boards of two 400MHz UltraSPARC® processors, connected by a crossbar UPA switch, and running Solaris™ 9 Operating Environment. Our C code was compiled by a Sun `cc` compiler 5.3, with flags `-x05 -xarch=v8plusa`. All our tests use kernel-space threads rather than user-space threads.

4.1 The Benchmarked Algorithms

We compared our stack implementation to the lock-free but non-linearizable elimination tree of Shavit and Touitou [24] and to two linearizable methods: a serial stack protected by MCS lock [19], and the non-blocking IBM/Treiber algorithm [17; 29].

- MCS** A serial stack protected by an MCS-queue-lock [19]. Each processor locks the top of the stack, changes it according to the type of the operation, and then unlocks it. The lock code was taken directly from the article.
- IBM/Treiber** Our implementation of the IBM/Treiber non-blocking stack followed the code given in [29]. We added to it exponential backoff scheme, as introduced in [2].
- ETree** An elimination tree [24] based stack. Its parameters were chosen so as to optimize its performance, based on empirical testing.

4.2 The Produce-Consume Benchmark

In the produce-consume benchmark, each thread alternately performs a push or pop operation and then waits for a period of time whose length is chosen uniformly at random from the range: $[0 \dots \text{workload}]$. The waiting period simulates the local work that is typically done by threads in real applications between stack operations (see Figure 7). In all our experiments the stack was initialized as sufficiently filled to prevent it from becoming empty during the run.

4.3 Measuring the performance of benchmarked algorithms

We ran the produce-consume benchmark specified above varying the number of threads and measuring *latency*, the average amount of time spent per operation, and *throughput*, the number of operations per second. We compute throughput and latency by measuring the total time required to perform the specific amount of operations by each thread. We refer to the longest time as the time needed to complete the specified amount of work.

```

repeat
    op:=random(push,pop)
    perform op
    w:=random(0..workload)
    wait w millisecs
until 500000 operations performed

```

Fig. 7. Produce-Consume benchmark

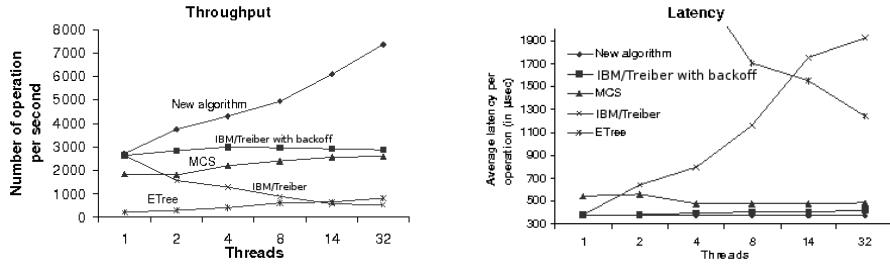


Fig. 8. Throughput and latency of different stack implementations with varying number of threads. Each thread performs 50% pushes, 50% pops.

To counteract transient startup effects, we synchronized the start of the threads (i.e., no thread can start before all other threads finished their initialization phase). Each data point is the average of three runs, with the results varying by at most 1.4% throughout all our benchmarks.

4.4 Empirical Results

Figure 8 shows the results of a benchmark in which half a million operations were performed by every working thread, with each thread performing 50% pushes and 50% pops on average. Figure 9 provides a detailed view of the three best performers. From Figure 8 it can be seen that our results for known structures generally conform with those of [21; 23], and that the IBM/Treiber algorithm with added exponential backoff is the best among known techniques. It can also be seen that the new algorithm provides superior scalable performance at all tested concurrency levels. The throughput gap between our algorithm and the IBM/Treiber algorithm with backoff grows as concurrency increases, and at 32 threads the new algorithm is almost three times faster. Such a significant gap in performance can be explained by reviewing the difference in latency for the two algorithms.

Table 1 shows latency measured on a single dedicated processor. The new algorithm and the IBM/Treiber algorithm with backoff have about the same latency, and outperform all others. The reason the new algorithm achieves this good performance is due to the fact that elimination backoff (unlike the elimination used in structures such as combining funnels and elimination trees) is used only as a backoff scheme and introduces no overhead. The gap of the two algorithms, with respect to MCS and ETree, is mainly due to the fact that a push or a pop in our algorithm and in the IBM/Treiber algorithm typically needs to access only two cache lines in the data structure, while a lock-based algorithm has the overhead of accessing lock variables as well. The ETree has an overhead of travelling through the tree.

As Figure 9 shows, as the level of concurrency increases, the latency of the

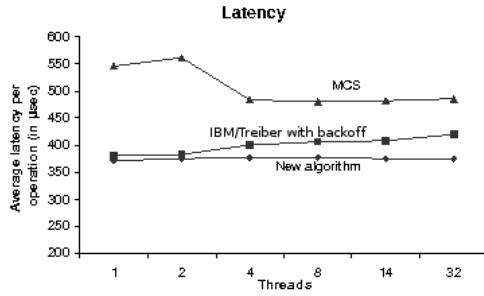


Fig. 9. Detailed graph of latency with threads performing 50% pushes, 50% pops.

IBM/Treiber algorithm grows since the head of the stack, even with contention removed, is a sequential bottleneck. On the other hand, the new algorithm has increased the rate of successful collisions on the elimination array as concurrency increases. As Table 2 shows, the fraction of successfully eliminated operations increases from only 11% for two threads up to 43% for 32 threads. The increased elimination level means that increasing numbers of threads complete their operations quickly and in parallel, keeping latency fixed and increasing overall throughput.

We also tested the robustness of the algorithms under workloads with an imbalanced distribution of push and pop operations. Such imbalanced workloads are not favorable for the new algorithm because of the smaller chance of successful collision. From Figure 10 it can be seen that the new algorithm still scales, but at a slower rate. The slope of the latency curve for our algorithm is 0.13 μ sec per thread, while the slope of the latency curve for the IBM/Treiber algorithm is 0.3 μ sec per thread, explaining the difference in throughput as concurrency increases.

In Figure 11 we compare the various methods under sparse access patterns and low load, by setting `workload` = 1000. In these circumstances, all the algorithms (with the exception of the elimination tree) maintain an almost constant latency as the level of concurrency increases because of the low contention. The decrease in the latency of elimination tree w.r.t. the case of `workload` = 0 is smaller, because it achieves lower levels of elimination. In contrast, the adverse effect of the sparse

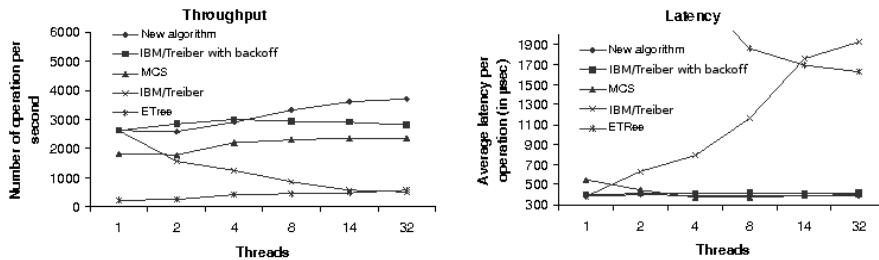


Fig. 10. Throughput and latency under varying distribution of operations: 25% push, 75%pop

Table 1. Latency on a single processor (no contention).

New algorithm	370
IBM/Treiber with backoff	380
MCS	546
IBM/Treiber	380
ETree	6850

Table 2. Fraction of successfully eliminated operations per concurrency level

2 threads	11%
4 threads	24%
8 threads	32%
14 threads	37%
32 threads	43%

access pattern on our algorithm’s latency is small, because our algorithm uses the collision layer only as a backup if it failed to access the central stack object, and the rate of such failures is low when the overall load is low.

To further test the effectiveness of our policy of using elimination as a backoff scheme, we measured the fraction of operations that failed on their first attempt to change the top of the stack. As seen in Figure 12, this fraction is low under low loads (as can be expected) and grows together with load, and, perhaps unexpectedly, is lower than in the IBM/Treiber algorithm. This is a result of using the collision layer as the backoff mechanism in the new algorithm as opposed to regular backoff, since in the new algorithm some of the failed threads are eliminated and do not interfere with the attempts of newly arrived threads to modify the stack. These results further justify the choice of elimination as a backoff scheme.

To study the behavior of our adaptation strategy we conducted a series of experiments to hand-pick the “optimized parameter set” for each level of concurrency. We then compared the performance of elimination backoff with an adaptive strategy to an optimized elimination backoff stack. These results are summarized in Figure 13. Comparing the latency of the best set of parameters to those achieved using adaptation, we see that the adaptive strategy is about 2.5% - 4% slower.

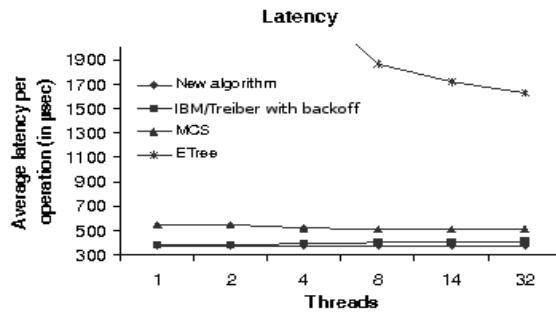


Fig. 11. Workload=1000

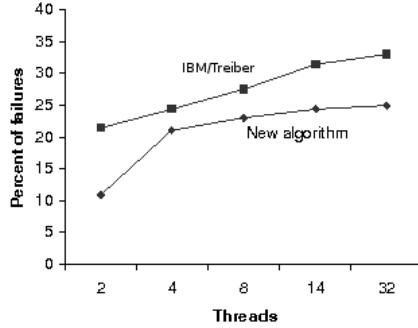


Fig. 12. Fraction of failures on first attempt

From these results we conclude that our adaptation techniques appear to work reasonably well. Based on the above benchmarks, we conclude that for the concurrency range tested, elimination backoff is the algorithm of choice for implementing linearizable stacks.

5. CORRECTNESS PROOFS

This section contains a formal proof that our algorithm is a lock-free linearizable implementation of a stack. It is organized as follows. In Section 5.1, we describe the model used by our proofs. In section 5.2, we prove basic correctness properties of our algorithm. Proofs of linearizability and lock-freedom are then provided in Sections 5.3 and 5.4, respectively.

5.1 Model

Our model for multithreaded computation follows [16], though for brevity and accessibility we will use operational style arguments. A concurrent system models an asynchronous shared memory system where a set P of n deterministic *threads* communicate by executing atomic *operations* on shared *variables* from some finite set

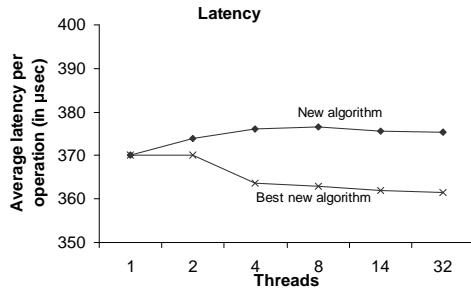


Fig. 13. Comparison of algorithm latency achieved by hand-picked parameters with that achieved by using an adaptive strategy

V . Each thread is a sequential execution path that performs a sequence of *steps*. In each step, a thread may perform some local computation and may invoke at most a single atomic operation on a shared variable. The atomic operations allowed in our model are *read*, *write*, and *compare-and-swap*. The compare-and-swap operation (abbreviated CAS) is defined as follows: $CAS(v, \text{expected}, \text{new})$ changes the value of variable v to new only if its value just before CAS is applied is expected ; in this case, the *CAS* operation returns *true* and we say it is *successful*. Otherwise, CAS does not change the value of v and returns *false*; in this case, we say that the *CAS* was *unsuccessful*.

A *configuration* of the system is a vector of size $n + |V|$, that stores the states of all threads of P and the values of all variables of V .¹ We say that *thread t is enabled to execute line L at configuration s* if t's state in s implies that, when next scheduled, t executes line L .

An *execution* is a (finite or infinite) sequence of steps that starts from an *initial configuration*. This is a configuration in which all variables in V have their initial values and all threads are in their initial states. If $o \in \mathbf{B}$ is a base object and E is a finite execution, then $\text{value}(E, o)$ denotes the value of o at the end of E . If no event in E changes the value of o , then $\text{value}(E, o)$ is the initial value of o . In other words, in the configuration resulting from executing E , each base object $o \in \mathbf{B}$ has value $\text{value}(E, o)$. For any finite execution fragment E and any execution fragment E' , the execution fragment EE' denotes the concatenation of E and E' .

A concurrent stack is a data structure whose operations are linearizable [16] to those of the sequential stack as defined in [6]. The following is a sequential specification of a stack object.

DEFINITION 1. A stack S is an object that supports two types of operations: *push* and *pop*. The state of a stack is a sequence of items $S = \langle v_0, \dots, v_k \rangle$. The stack is initially empty. The push and pop operations induce the following state transitions of the sequence $S = \langle v_0, \dots, v_k \rangle$, with appropriate return values:

- $\text{push}(v_{\text{new}})$, changes S to be $\langle v_0, \dots, v_k, v_{\text{new}} \rangle$
- $\text{pop}()$, if S is not empty, changes S to be $\langle v_0, \dots, v_{k-1} \rangle$ and returns v_k ; if S is empty, it returns empty and S remains unchanged.

We note that a *pool* is a relaxation of a stack that does not require LIFO ordering. We start by proving that our algorithm implements a concurrent pool, without considering a linearization order. We then prove that our stack implementation is linearizable to the sequential stack specification of Definition 1. Finally we prove that our implementation is lock-free.

5.2 Correct Pool Semantics

We first prove that our algorithm has correct pool semantics, i.e., that *pop* operations can only pop items that were previously pushed, and that items pushed by *push* operations are not duplicated and can be popped out. This is formalized in the following definition.²

DEFINITION 2. A stack algorithm has correct pool semantics if the following requirements are met for all stack operations:

¹The state of each thread consists of the values of the thread's local variables, registers and program-counter.

²For simplicity we assume all items are unique, but the proof can easily be modified to work without this assumption.

- (1) Let Op be a pop operation that returns an item i , then i was previously pushed by a push operation.
- (2) Let Op be a push operation that pushed an item i to the stack, then there is at most a single pop operation that returns i .
- (3) Let Op be a pop operation, then if the number of push operations preceding Op is larger than the number of pop operations preceding it, Op returns a value.

We call any operation that complies with the above requirement a correct pool operation.

LEMMA 2. Operations that modify the central stack object are correct pool operations.

PROOF. Follows from the correctness of Treiber's algorithm [29]. \square

In the following, we prove that operations that exchange their values through collisions are also correct pool operations, thus we show that our algorithm has correct pool semantics. We first need the following definitions.

DEFINITION 3. We say that an operation op is a colliding operation if it returns in line S12, S17 or S22 of *LesOP*. If op performs a push then we say it is a push colliding operation, otherwise we say that it is a pop colliding operation.

DEFINITION 4. Let op_1 be a push operation and op_2 be a pop operation. We say that op_1 and op_2 have collided if op_2 obtains the value pushed by op_1 without accessing the central stack object. More formally, we require that one of the following conditions hold:

- Operation op_2 performs a successful CAS in line C8 of *TryCollision* and q points to the *ThreadInfo* structure representing op_1 at that time.
- Operation op_2 performs a CAS operation in line S20 of *TryPerformStackOp* and the CAS fails because the entry of the location array corresponding to the thread executing op_2 points at that time to the *ThreadInfo* structure representing op_1 .

DEFINITION 5. We say that a colliding operation op is active if it executes a successful CAS in lines C2 or C8 of *TryCollision*. We say that a colliding operation is passive if op performs an unsuccessful CAS operation in lines S10 or S20 of *LesOP*.

DEFINITION 6. Let op be an operation performed by thread t and let s be a configuration. If t is enabled to execute a line of *LesOP*, *TryCollision* or *FinishCollision* in s , then we say that t is trying to collide at s . Otherwise, we say that op is not trying to collide at s .

We next prove that operations can only collide with operations of the opposite type. In the proofs that follow, we let l_t denote the element corresponding to t in the *location* array. First we need the following technical lemma.

LEMMA 3. Every colliding operation op is either active or passive, but not both.

PROOF. Let Op be a colliding operation. From Definition 3, we only need to consider the following cases.

- Op returns in line S12. In this case, op performed a successful CAS in line C2 or C8 of *TryCollision*. Thus, from Definition 5, Op is active. To obtain a contradiction, assume that Op is also passive. It follows from Definition 5, that

Op performed an unsuccessful CAS operation in line S10 or S20 before calling `TryCollision`. In each of these cases, however, *Op* returns after performing `FinishCollision` and does not call `TryCollision` after that. This is a contradiction.

- Op* returns in line S17. It follows that *Op* performed an unsuccessful CAS operation in line S10. Hence, from Definition 5, *Op* is passive. To obtain a contradiction, assume that *Op* is also active. It follows from Definition 5, that *Op* executed a successful CAS operation in lines C2 or C8 of `TryCollision` before its unsuccessful CAS in line S10. In this case, however, `TryCollision` returns *true* and so *Op* immediately returns in line S12. This is a contradiction.
- Op* returns in line S22. It follows that *Op* performs an unsuccessful CAS operation in line S20. Hence, from Definition 5, *Op* is passive. The proof proceeds in a manner identical to that of the corresponding proof for line S17.

□

LEMMA 4. *Operations can only collide with operations of the opposite type: an operation that performs a `push` can only collide with operations that perform a `pop`, and vice versa.*

PROOF. Let *op* be a colliding operation. From the code and from Definition 3, *op* either returns *true* from `TryCollision` or executes `FinishCollision`. We now examine both these cases.

- (1) `TryCollision` can succeed only in case of a successful CAS in line C2 (for a `push` operation) or in line C8 (for a `pop` operation). Such a CAS changes the value of the other thread's cell in the *location* array, thus exchanging values with it and returns without modifying the central stack object. From the code, before calling `TryCollision` *op* has to execute line S9, thus verifying that it collides with an operation of the opposite type. Finally, from Lemma 1, *q* points to the same `ThreadInfo` structure starting from when it is assigned in line S8 by *Op* and until either line C2 or C8 is performed by *Op*.
- (2) If *op* is a passive colliding operation, then *op* performs `FinishCollision`, which implies that *op* failed in resetting its entry in the location array (in line S10 or s20). Let *op1* be the operation that has caused *op*'s failure by writing to its entry. From the code, *op1* must have succeeded in `TryCollision`. The proof now follows from case (1).

□

LEMMA 5. *An operation terminates without modifying the central stack object if and only if it is a colliding operation.*

PROOF. If *Op* modifies the central stack object, then its call of `TryPerformStackOp` returns *true* and it returns in line S24. It follows from Definition 3 that *Op* is not a colliding operation. As for the other direction, if *Op* is a colliding operation, then it returns in lines S12, S17 or S22. It follows that it does not return in line S24, hence it could not have changed the central stack object. □

LEMMA 6. *Let *s* be a configuration, and let *t* be a thread . Then *t* is trying to collide in *s* if and only if $l_t \neq \text{NULL}$ holds in *s*.*

PROOF. In the initial configuration, $l_t = \text{NULL}$ holds. In the beginning of each collision attempt performed by operation *op*, the value of l_t is set to a non-NUL

value in line S2. We need to show that op changes the value of l_t back to NULL before completing the collision attempt, either successfully or unsuccessfully. We now check both these cases.

- Suppose that op fails in its collision attempt, that is, op reaches line S23. Clearly from the code, to reach line S23, op has to perform a successful CAS in either line S10 or in line S20, thus it sets the value of l_t to NULL upon finishing its unsuccessful collision attempt.
- Otherwise, op succeeds in its collision attempt. Consequently, it exits `LesOp` in the same iteration, in lines S12, S17 or S22. Clearly from the code, to reach line S12 op has to perform a successful CAS in line S10, thus setting the value of l_t to NULL. If op returns in lines S17 or S22, then it returns after it executes `FinishCollision`. From the code of `FinishCollision`, if op is a pop operation, then `FinishCollision` sets l_t to NULL at line F3. Finally, if op is a push operation and reaches `FinishCollision`, then op must have failed to perform a CAS in lines S10 or S20. This failure implies that some other operation changed the value of l_t . As op is a push operation, we have by Lemma 4, that the value of l_t was changed by another operation, op_1 , that performed a pop operation; thus op_1 changed l_t to NULL in line C8.

□

LEMMA 7. *Let Op be a **push** operation by some thread t , and let s be a configuration. If it holds in s that $l_t \neq \text{NULL}$, then Op is trying in s to push the value $l_t \rightarrow \text{cell}.pdata$.*

PROOF. Clearly from the code and from Lemma 1, only Op can write a value different than NULL to l_t . From Lemma 6, l_t is NULL after Op exits, hence Op is in the midst of a collision attempt in configuration s . From Lemma 1, it follows that the value of l_t in s is a pointer to t 's `ThreadInfo` structure written on line S2. As Op is a **push** operation, the cell Op is trying to push is pointed at from the `cell` field of that structure. □

LEMMA 8. *Every passive colliding operation collides with exactly one active colliding operation and vice versa.*

PROOF. Immediate from Definition 4 and from Lemma 1. □

LEMMA 9. *Every colliding operation op collides with exactly one operation of the opposite type.*

PROOF. Follows from Lemmas 3 and 8. □

We now prove that, upon colliding, opposite operations exchange values in a proper way.

LEMMA 10. *If a pop operation collides, it obtains the value of the single push operation it collided with.*

PROOF. Let op_1 denote a pop operation performed by thread t . If op_1 is a passive colliding operation, then, from Lemma 9 and Definition 5, it collides with a single active push colliding operation, op_2 . As op_1 succeeds in colliding, from Definition 4 and from Lemma 1, it obtains in line F2 the cell that was written to l_t by op_2 .

Assume now that op_1 is an active colliding operation, then, from Lemma 9 it collides with a single passive push colliding operation, op_2 . As op_1 succeeds in

colliding, it succeeds in the CAS of line C8 and thus, from Lemma 1, returns the cell that was written by op_2 . \square

LEMMA 11. *If a push operation collides, its value is obtained by the single pop operation it collided with.*

PROOF. Let op_1 denote a push operation performed by thread t . If op_1 is a passive colliding operation, then, from Lemma 9 and Definition 5, it collides with a single active pop colliding operation, op_2 . As op_1 is passive, from Definition 5, op_1 performs an unsuccessful CAS in line S10 or S20. From Lemma 1, it follows that the value of l_t was previously set to NULL by op_2 as it obtained in line F2 the cell that was written by op_1 to l_t .

Assume now that op_1 is a active colliding operation, then, from Lemma 9 and Definition 5, it collides with a single passive pop colliding operation, op_2 . Let q be the thread performing op_2 . As op_1 is active, from Definition 5, it performs a successful CAS operation in line C2, thus writing a pointer to its ThreadInfo structure to l_q . From Lemma 1, it follows that, if and when op_2 returns, it returns the cell field of this structure. \square

We can now prove that our algorithm has correct pool semantics.

THEOREM 1. *The elimination-backoff stack has correct pool semantics.*

PROOF. From Lemma 2, all operations that modify the central stack object are correct pool operations. From Lemmas 10 and 11, all colliding operations are correct pool operations. Thus, all operations on the elimination-backoff stack are correct pool operations. It follows from Definition 2 that the elimination-backoff stack has correct pool semantics. \square

5.3 Linearizability

Given a sequential specification of a stack, we provide specific linearization points mapping operations in our concurrent implementation to sequential operations so that the histories meet the specification.

DEFINITION 7. *The elimination backoff stack linearization points are selected as follows. All operations, except for passive colliding operations, are linearized in the following lines, executed in their (single) successful iteration:*

- push operation are linearized in lines T4 or C2.
- pop operations are linearized in lines T10, T14 or C8.

For passive colliding operations, we set the linearization point to be at the time of linearization of the matching active colliding operation, and the push colliding operation is linearized before the pop colliding operation.

Each push or pop operation consists of a while loop that repeatedly attempts to complete the operation. We say that an iteration is a *successful iteration* if the operation returns at that iteration; otherwise, another iteration will be performed. Every completed operation has exactly one successful iteration (its last one), and the linearization point of the operation occurs in the course of performing that iteration.

From definition 1, it follows that a successful collision does not change the state of the central stack object. Consequently, at any point in time, the state of the stack is determined solely by the state of its central stack object. We proceed by

proving that the aforementioned code lines are correct linearization points both for operations that complete by modifying the central stack object, and for operations that exchange values through collisions.

LEMMA 12. *The lines specified in Definition 7 are correct linearization points for operations that complete by modifying the central stack object.*

PROOF. The linearization points specified in Definition 7 for operations that complete by modifying the central stack object are:

- Line T4 (for a push operation)
- Line T10 (in case of empty stack) or line T14 (for a pop operation)

Since colliding operations do not change the state of the stack, the claim follows directly from the linearizability of Treiber’s algorithm [29]. \square

Before establishing the correctness of the linearization points for colliding operations, we need the following technical lemma.

LEMMA 13. *Let op_1 be an active colliding operation and let op_2 be the passive colliding operation with which it collides. Then the linearization point of op_1 , as specified in Definition 7, is within the time interval of op_2 .*

PROOF. From definition 5, op_1 performs a successful CAS in line C2 (if it is a *push* operation) or in line C8 (if it is a *pop* operation). From Definition 7, this is op_1 ’s linearization point. From Lemma 9, op_1 collides only with op_2 and these two operations have opposite types.

Let s be the configuration immediately preceding the execution of the linearization line of op_1 . The success of the CAS in line C2 or C8 and Lemma 1 imply that the value of op_2 ’s entry in the location array is non-NULL in s (otherwise the check at line S9 would have failed). Thus, from Lemma 6 and definition 6, op_2 is trying to collide in configuration s . \square

LEMMA 14. *The lines specified in Definition 7 are correct linearization points for colliding operations.*

PROOF. To simplify the proof and avoid the need for backward simulation style arguments, we consider only complete execution histories, that is, ones in which all abstract operations have completed, so we can look “back” at the execution and say for each operation where it happened.

We first note that according to Lemma 13, the linearization point of passive colliding operations is well-defined (it is obviously well-defined for active colliding operations). We need to prove that correct LIFO ordering is maintained between any two linearized colliding operations and between these operations and operations that modify the central stack object.

As we linearize a passive colliding operation in the linearization point of its (single) counterpart active colliding operation, no other operations can be linearized between these two operations. Moreover, since the *push* operation is linearized just before the *pop* operation, this is a legal LIFO matching that cannot interfere with the LIFO matching of other pairs of colliding operations or with that of operations that modify the central stack object. Finally, from Lemma 10, the *pop* operation indeed obtains the value of the *push* operation it collides with. \square

THEOREM 2. *The elimination-backoff stack is a correct linearizable implementation of a stack object.*

PROOF. Immediate from Lemmas 12 and 14. \square

5.4 Lock Freedom

THEOREM 3. *The elimination-backoff stack algorithm is lock-free.*

PROOF. Let op be some operation. We show that in every iteration made by op , some operation performs its linearization point, thus the system as a whole makes progress. If op manages to collide, then op 's linearization has occurred, as have the linearization of the operation op collided with.

Otherwise, op calls *TryPerformStackOp*. If *TryPerformStackOp* returns TRUE, op immediately returns, and its linearization has occurred. If, on the other hand, *TryPerformStackOp* returns FALSE, then it must be that the CAS operation it applied to the central stack object was unsuccessful. This implies, in turn, that a CAS operation applied to $S.ptop$ by some other operation op_1 was successful. Hence, op_1 was linearized. It follows that whenever op completes a full iteration, some operation is linearized. \square

6. DISCUSSION

Shared stacks are widely used in parallel applications and operating systems. In this paper, we presented the *elimination backoff stack*, the first concurrent stack algorithm that is both linearizable, lock-free and can achieve high throughput in high contention executions. The *elimination backoff stack* is based on the following simple observation: that a single elimination array [24], used as a backoff scheme for a lock-free stack [29], is both lock-free and linearizable. The introduction of elimination into the backoff process serves a dual purpose of adding parallelism and reducing contention, which, as our empirical results show, allows the *elimination-backoff stack* to outperform all algorithms in the literature at both high and low loads.

We observe that, unlike the simple algorithm of [29] in which threads can be anonymous, our algorithm requires that all threads that concurrently perform collision attempts have unique identifiers. The same thread can use different identifiers, however, in different collision attempts. It is therefore easy to support applications in which threads are created and deleted dynamically: threads can get and release unique identifiers from a small name space by using any long-lived renaming algorithm (see, e.g., [3; 4]); since accessing the central stack does not require an identifier, there is no adverse effect on time complexity in the absence of contention. There is also no need of a-priori knowledge of the maximum number of concurrently participating threads, as a lock-free dynamically resizable array (see [7]) can be used instead of the static `location` array.

Our algorithm includes a tight “busy-waiting” loop in lines S5-S6, performed by a process as it tries to apply a successful CAS operation to an entry of the `collision` array. In general, long busy-waiting loops have adverse effect on performance; as our empirical results establish, however, this is not the case with our algorithm. The reason for this is that our algorithm uses a mechanism for dynamically adapting the width of the `collision` array; when the load is high, this mechanism increases the width of the `collision` array and reduces the probability of CAS failures. This ensures that the loop of lines S5-S6 is short in practice.

Several related works have appeared since the preliminary version of this paper was published. Colvin and Grobes [5] presented a somewhat simplified version of our algorithm and proved its correctness by using the PVS [22] theorem prover. Recently, Handler and Kutten [11] introduced *bounded-wait combining*, a technique by which asymptotically high-throughput lock-free linearizable implementations of

objects that support combinable operations (such as counters, stacks, and queues) can be constructed.

Acknowledgments

We would like to thank the anonymous referees for many valuable comments on a previous draft of this article, which helped improve its quality.

REFERENCES

- [1] A. Agarwal and M. Cherian. Adaptive backoff synchronization techniques. In *Proceedings of the 16th Symposium on Computer Architecture*, pages 41–55, June 1989.
- [2] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.
- [3] H. Attiya and A. Fournier. Adaptive and efficient algorithms for lattice agreement and renaming. *SIAM J. Comput.*, 31(2):642–664, 2001.
- [4] A. Brodsky, F. Ellen, and P. Woelfel. Fully-adaptive algorithms for long-lived renaming. In *DISC*, pages 413–427, 2006.
- [5] R. Colvin and L. Groves. A scalable lock-free stack algorithm and its verification. In *SEFM*, pages 339–348, 2007.
- [6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. MIT Press, Cambridge, Massachusetts, 2002.
- [7] D. Dechev, P. Pirkelbauer, and B. Stroustrup. Lock-free dynamically resizable arrays. In *OPODIS*, pages 142–156, 2006.
- [8] R. Goodman and M. K. V. P. J. Woest. Efficient synchronisation primitives for large-scale cache-coherent multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS-III*, pages 64–75, 1989.
- [9] A. Gottlieb, B. D. Lubachevsky, and L. Rudolph. Efficient techniques for coordinating sequential processors. *ACM TOPLAS*, 5(2):164–189, April 1983.
- [10] M. Greenwald. *Non-Blocking Synchronization and System Design*. PhD thesis, Stanford University Technical Report STAN-CS-TR-99-1624, Palo Alto, CA, 8 1999.
- [11] D. Hendler and S. Kutten. Bounded-wait combining: constructing robust and high-throughput shared objects. *Distributed Computing*, 21(6):405–431, 2009.
- [12] M. Herlihy. Wait-free synchronization. *ACM Transactions On Programming Languages and Systems*, 13(1):123–149, Jan. 1991.
- [13] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.
- [14] M. Herlihy, B.-H. Lim, and N. Shavit. Scalable concurrent counting. *ACM Transactions on Computer Systems*, 13(4):343–364, 1995.
- [15] M. Herlihy, V. Luchangco, and M. Moir. The repeat-offender problem, a mechanism for supporting dynamic-sized,lock-free data structures. Technical Report TR-2002-112, Sun Microsystems, September 2002.
- [16] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.
- [17] IBM. *IBM System/370 Extended Architecture, Principles of Operation, publication no. SA22-7085*. 1983.
- [18] B.-H. Lim and A. Agarwal. Waiting algorithms for synchronization in large-scale multiprocessors. *ACM Transactions on Computer Systems*, 11(3):253–294, August 1993.
- [19] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, 1991.
- [20] M. M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing*, pages 21–30. ACM Press, 2002.

- [21] M. M. Michael and M. L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared — memory multiprocessors. *Journal of Parallel and Distributed Computing*, 51(1):1–26, 1998.
- [22] S. Owre, J. M. Rushby, and N. Shankar. Pvs: A prototype verification system. In *CADE*, pages 748–752, 1992.
- [23] M. Scott and W. Scherer. User-level spin locks for large commercial applications. In *SOSP, Work-in-progress talk*, 2001.
- [24] N. Shavit and D. Touitou. Elimination trees and the construction of pools and stacks. *Theory of Computing Systems*, (30):645–670, 1997.
- [25] N. Shavit, E. Upfal, and A. Zemach. A steady state analysis of diffracting trees. *Theory of Computing Systems*, 31(4):403–423, 1998.
- [26] N. Shavit and A. Zemach. Diffracting trees. *ACM Transactions on Computer Systems*, 14(4):385–428, 1996.
- [27] N. Shavit and A. Zemach. Combining funnels: A dynamic approach to software combining. *Journal of Parallel and Distributed Computing*, (60):1355–1387, 2000.
- [28] K. Taura, S. Matsuoka, and A. Yonezawa. An efficient implementation scheme of concurrent object-oriented languages on stock multicomputers. In *Principles Practice of Parallel Programming*, pages 218–228, 1993.
- [29] R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, April 1986.

Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms*

Maged M. Michael Michael L. Scott

Department of Computer Science

University of Rochester

Rochester, NY 14627-0226

{michael, scott}@cs.rochester.edu

Abstract

Drawing ideas from previous authors, we present a new non-blocking concurrent queue algorithm and a new two-lock queue algorithm in which one enqueue and one dequeue can proceed concurrently. Both algorithms are simple, fast, and practical; we were surprised not to find them in the literature. Experiments on a 12-node SGI Challenge multiprocessor indicate that the new non-blocking queue consistently outperforms the best known alternatives; it is the clear algorithm of choice for machines that provide a universal atomic primitive (e.g. `compare_and_swap` or `load_linked/store_conditional`). The two-lock concurrent queue outperforms a single lock when several processes are competing simultaneously for access; it appears to be the algorithm of choice for busy queues on machines with non-universal atomic primitives (e.g. `test_and_set`). Since much of the motivation for non-blocking algorithms is rooted in their immunity to large, unpredictable delays in process execution, we report experimental results both for systems with dedicated processors and for systems with several processes multiprogrammed on each processor.

Keywords: concurrent queue, lock-free, non-blocking, `compare_and_swap`, multiprogramming.

*This work was supported in part by NSF grants nos. CDA-94-01142 and CCR-93-19445, and by ONR research grant no. N00014-92-J-1801 (in conjunction with the DARPA Research in Information Science and Technology—High Performance Computing, Software Science and Technology program, ARPA Order no. 8930).

1 Introduction

Concurrent FIFO queues are widely used in parallel applications and operating systems. To ensure correctness, concurrent access to shared queues has to be synchronized. Generally, algorithms for concurrent data structures, including FIFO queues, fall into two categories: *blocking* and *non-blocking*. Blocking algorithms allow a slow or delayed process to prevent faster processes from completing operations on the shared data structure indefinitely. Non-blocking algorithms guarantee that if there are one or more active processes trying to perform operations on a shared data structure, some operation will complete within a finite number of time steps. On asynchronous (especially multiprogrammed) multiprocessor systems, blocking algorithms suffer significant performance degradation when a process is halted or delayed at an inopportune moment. Possible sources of delay include processor scheduling preemption, page faults, and cache misses. Non-blocking algorithms are more robust in the face of these events.

Many researchers have proposed lock-free algorithms for concurrent FIFO queues. Hwang and Briggs [7], Sites [17], and Stone [20] present lock-free algorithms based on `compare_and_swap`.¹ These algorithms are incompletely specified; they omit details such as the handling of empty or single-item queues, or concurrent enqueues and dequeues. Lamport [9] presents a wait-free algorithm that restricts concurrency to a single enqueue and a single dequeuer.²

Gottlieb *et al.* [3] and Mellor-Crummey [11] present algorithms that are lock-free but not non-blocking: they do not use locking mechanisms, but they allow a slow process to delay faster processes indefinitely.

¹`Compare_and_swap`, introduced on the IBM System 370, takes as arguments the address of a shared memory location, an expected value, and a new value. If the shared location currently holds the expected value, it is assigned the new value atomically. A Boolean return value indicates whether the replacement occurred.

²A *wait-free* algorithm is both non-blocking and starvation free: it guarantees that every active process will make progress within a bounded number of time steps.

Treiber [21] presents an algorithm that is non-blocking but inefficient: a dequeue operation takes time proportional to the number of the elements in the queue. Herlihy [6]; Prakash, Lee, and Johnson [15]; Turek, Shasha, and Prakash [22]; and Barnes [2] propose general methodologies for generating non-blocking versions of sequential or concurrent lock-based algorithms. However, the resulting implementations are generally inefficient compared to specialized algorithms.

Massalin and Pu [10] present lock-free algorithms based on a `double_compare_and_swap` primitive that operates on two arbitrary memory locations simultaneously, and that seems to be available only on later members of the Motorola 68000 family of processors. Herlihy and Wing [4] present an array-based algorithm that requires infinite arrays. Valois [23] presents an array-based algorithm that requires either an unaligned `compare_and_swap` (not supported on any architecture) or a Motorola-like `double_compare_and_swap`.

Stone [18] presents a queue that is lock-free but non-linearizable³ and not non-blocking. It is non-linearizable because a slow enqueuer may cause a faster process to enqueue an item and subsequently observe an empty queue, even though the enqueued item has never been dequeued. It is not non-blocking because a slow enqueue can delay dequeues by other processes indefinitely. Our experiments also revealed a race condition in which a certain interleaving of a slow dequeue with faster enqueues and dequeues by other process(es) can cause an enqueued item to be lost permanently. Stone also presents [19] a non-blocking queue based on a circular singly-linked list. The algorithm uses one anchor pointer to manage the queue instead of the usual head and tail. Our experiments revealed a race condition in which a slow dequeuer can cause an enqueued item to be lost permanently.

Prakash, Lee, and Johnson [14, 16] present a linearizable non-blocking algorithm that requires enqueueing and dequeuing processes to take a snapshot of the queue in order to determine its “state” prior to updating it. The algorithm achieves the non-blocking property by allowing faster processes to complete the operations of slower processes instead of waiting for them.

Valois [23, 24] presents a list-based non-blocking algorithm that avoids the contention caused by the snapshots of Prakash *et al.*’s algorithm and allows more concurrency by keeping a dummy node at the head (dequeue end) of a singly-linked list, thus simplifying the special cases associated with empty and single-item queues (a technique suggested by Sites [17]). Unfortunately, the algorithm allows the tail pointer to lag behind the head pointer, thus preventing dequeuing processes from safely freeing or re-

using dequeued nodes. If the tail pointer lags behind and a process frees a dequeued node, the linked list can be broken, so that subsequently enqueued items are lost. Since memory is a limited resource, prohibiting memory reuse is not an acceptable option. Valois therefore proposes a special mechanism to free and allocate memory. The mechanism associates a reference counter with each node. Each time a process creates a pointer to a node it increments the node’s reference counter atomically. When it does not intend to access a node that it has accessed before, it decrements the associated reference counter atomically. In addition to temporary links from process-local variables, each reference counter reflects the number of links in the data structure that point to the node in question. For a queue, these are the head and tail pointers and linked-list links. A node is freed only when no pointers in the data structure or temporary variables point to it.

We discovered and corrected [13] race conditions in the memory management mechanism and the associated non-blocking queue algorithm. Even so, the memory management mechanism and the queue that employs it are impractical: no finite memory can guarantee to satisfy the memory requirements of the algorithm all the time. Problems occur if a process reads a pointer to a node (incrementing the reference counter) and is then delayed. While it is not running, other processes can enqueue and dequeue an arbitrary number of additional nodes. Because of the pointer held by the delayed process, neither the node referenced by that pointer nor any of its successors can be freed. It is therefore possible to run out of memory even if the number of items in the queue is bounded by a constant. In experiments with a queue of maximum length 12 items, we ran out of memory several times during runs of ten million enqueues and dequeues, using a free list initialized with 64,000 nodes.

Most of the algorithms mentioned above are based on `compare_and_swap`, and must therefore deal with the ABA problem: if a process reads a value *A* in a shared location, computes a new value, and then attempts a `compare_and_swap` operation, the `compare_and_swap` may succeed when it should not, if between the read and the `compare_and_swap` some other process(es) change the *A* to a *B* and then back to an *A* again. The most common solution is to associate a modification counter with a pointer, to always access the counter with the pointer in any read-modify-`compare_and_swap` sequence, and to increment it in each successful `compare_and_swap`. This solution does not guarantee that the ABA problem will not occur, but it makes it extremely unlikely. To implement this solution, one must either employ a double-word `compare_and_swap`, or else use array indices instead of pointers, so that they may share a single word with a counter. Valois’s reference counting technique guarantees preventing the ABA problem without the need for modification counters or the double-word `compare_and_swap`. Mellor-Crummey’s lock-free queue [11] requires no special precautions to avoid

³An implementation of a data structure is linearizable if it can always give an external observer, observing only the abstract data structure operations, the illusion that each of these operations takes effect instantaneously at some point between its invocation and its response [5].

the ABA problem because it uses `compare_and_swap` in a `fetch_and_store-modify-compare_and_swap` sequence rather than the usual `read-modify-compare_and_swap` sequence. However, this same feature makes the algorithm blocking.

In section 2 we present two new concurrent FIFO queue algorithms inspired by ideas in the work described above. Both of the algorithms are simple and practical. One is non-blocking; the other uses a pair of locks. Correctness of these algorithms is discussed in section 3. We present experimental results in section 4. Using a 12-node SGI Challenge multiprocessor, we compare the new algorithms to a straightforward single-lock queue, Mellor-Crummey’s blocking algorithm [11], and the non-blocking algorithms of Prakash *et al.* [16] and Valois [24], with both dedicated and multiprogrammed workloads. The results confirm the value of non-blocking algorithms on multiprogrammed systems. They also show consistently superior performance on the part of the new lock-free algorithm, both with and without multiprogramming. The new two-lock algorithm cannot compete with the non-blocking alternatives on a multiprogrammed system, but outperforms a single lock when several processes compete for access simultaneously. Section 5 summarizes our conclusions.

2 Algorithms

Figure 1 presents commented pseudo-code for the non-blocking queue data structure and operations. The algorithm implements the queue as a singly-linked list with *Head* and *Tail* pointers. As in Valois’s algorithm, *Head* always points to a dummy node, which is the first node in the list. *Tail* points to either the last or second to last node in the list. The algorithm uses `compare_and_swap`, with modification counters to avoid the ABA problem. To allow dequeuing processes to free dequeued nodes, the dequeue operation ensures that *Tail* does not point to the dequeued node nor to any of its predecessors. This means that dequeued nodes may safely be re-used.

To obtain consistent values of various pointers we rely on sequences of reads that re-check earlier values to be sure they haven’t changed. These sequences of reads are similar to, but simpler than, the snapshots of Prakash *et al.* (we need to check only one shared variable rather than two). A similar technique can be used to prevent the race condition in Stone’s blocking algorithm. We use Treiber’s simple and efficient non-blocking stack algorithm [21] to implement a non-blocking free list.

Figure 2 presents commented pseudo-code for the two-lock queue data structure and operations. The algorithm employs separate *Head* and *Tail* locks, to allow complete concurrency between enqueues and dequeues. As in the non-blocking queue, we keep a dummy node at the beginning of the list. Because of the dummy node, enqueueers never have to access *Head*, and dequeuers never have to access *Tail*, thus avoiding potential deadlock problems that

arise from processes trying to acquire the locks in different orders.

3 Correctness

3.1 Safety

The presented algorithms are safe because they satisfy the following properties:

1. The linked list is always connected.
2. Nodes are only inserted after the last node in the linked list.
3. Nodes are only deleted from the beginning of the linked list.
4. *Head* always points to the first node in the linked list.
5. *Tail* always points to a node in the linked list.

Initially, all these properties hold. By induction, we show that they continue to hold, assuming that the ABA problem never occurs.

1. The linked list is always connected because once a node is inserted, its *next* pointer is not set to NULL before it is freed, and no node is freed until it is deleted from the beginning of the list (property 3).
2. In the lock-free algorithm, nodes are only inserted at the end of the linked list because they are linked through the *Tail* pointer, which always points to a node in the linked-list (property 5), and an inserted node is linked only to a node that has a NULL *next* pointer, and the only such node in the linked list is the last one (property 1).

In the lock-based algorithm nodes are only inserted at the end of the linked list because they are inserted after the node pointed to by *Tail*, and in this algorithm *Tail* always points to the last node in the linked list, unless it is protected by the tail lock.

3. Nodes are deleted from the beginning of the list, because they are deleted only when they are pointed to by *Head* and *Head* always points to the first node in the list (property 4).
4. *Head* always points to the first node in the list, because it only changes its value to the next node atomically (either using the head lock or using `compare_and_swap`). When this happens the node it used to point to is considered deleted from the list. The new value of *Head* cannot be NULL because if there is one node in the linked list the dequeue operation returns without deleting any nodes.

```

structure pointer_t      {ptr: pointer to node_t, count: unsigned integer}
structure node_t        {value: data type, next: pointer_t}
structure queue_t       {Head: pointer_t, Tail: pointer_t}

initialize(Q: pointer to queue_t)
    node = new_node()                                # Allocate a free node
    node->next.ptr = NULL                           # Make it the only node in the linked list
    Q->Head = Q->Tail = node                        # Both Head and Tail point to it

enqueue(Q: pointer to queue_t, value: data type)
E1:   node = new_node()                            # Allocate a new node from the free list
E2:   node->value = value                         # Copy enqueueued value into node
E3:   node->next.ptr = NULL                       # Set next pointer of node to NULL
E4:   loop                                         # Keep trying until Enqueue is done
E5:     tail = Q->Tail                           # Read Tail.ptr and Tail.count together
E6:     next = tail.ptr->next                     # Read next ptr and count fields together
E7:     if tail == Q->Tail                      # Are tail and next consistent?
E8:       if next.ptr == NULL                      # Was Tail pointing to the last node?
E9:         if CAS(&tail.ptr->next, next, <node, next.count+1>) # Try to link node at the end of the linked list
E10:        break                                # Enqueue is done. Exit loop
E11:    endif
E12:  else                                         # Tail was not pointing to the last node
E13:    CAS(&Q->Tail, tail, <next.ptr, tail.count+1>) # Try to swing Tail to the next node
E14:  endif
E15: endif
E16: endloop
E17: CAS(&Q->Tail, tail, <node, tail.count+1>) # Enqueue is done. Try to swing Tail to the inserted node

dequeue(Q: pointer to queue_t, pvalue: pointer to data type): boolean
D1:   loop                                         # Keep trying until Dequeue is done
D2:     head = Q->Head                           # Read Head
D3:     tail = Q->Tail                           # Read Tail
D4:     next = head->next                      # Read Head.ptr->next
D5:     if head == Q->Head                      # Are head, tail, and next consistent?
D6:       if head.ptr == tail.ptr                 # Is queue empty or Tail falling behind?
D7:         return FALSE                          # Is queue empty?
D8:       endif                                 # Queue is empty, couldn't dequeue
D9:     endif
D10:    CAS(&Q->Tail, tail, <next.ptr, tail.count+1>) # Tail is falling behind. Try to advance it
D11:    else                                         # No need to deal with Tail
D12:      # Read value before CAS, otherwise another dequeue might free the next node
D13:      *pvalue = next.ptr->value
D14:      if CAS(&Q->Head, head, <next.ptr, head.count+1>) # Try to swing Head to the next node
D15:        break                                # Dequeue is done. Exit loop
D16:      endif
D17:    endif
D18:  endloop
D19:  free(head.ptr)                            # It is safe now to free the old dummy node
D20:  return TRUE                            # Queue was not empty, dequeue succeeded

```

Figure 1: Structure and operation of a non-blocking concurrent queue.

```

structure node_t { value: data type, next: pointer to node_t }
structure queue_t { Head: pointer to node_t, Tail: pointer to node_t, H_lock: lock type, T_lock: lock type }

initialize(Q: pointer to queue_t)
    node = new_node()          # Allocate a free node
    node->next.ptr = NULL      # Make it the only node in the linked list
    Q->Head = Q->Tail = node  # Both Head and Tail point to it
    Q->H_lock = Q->T_lock = FREE # Locks are initially free

enqueue(Q: pointer to queue_t, value: data type)
    node = new_node()          # Allocate a new node from the free list
    node->value = value        # Copy enqueued value into node
    node->next.ptr = NULL      # Set next pointer of node to NULL
    lock(&Q->T_lock)          # Acquire T_lock in order to access Tail
    Q->Tail->next = node      # Link node at the end of the linked list
    Q->Tail = node             # Swing Tail to node
    unlock(&Q->T_lock)         # Release T_lock

dequeue(Q: pointer to queue_t, pvalue: pointer to data type): boolean
    lock(&Q->H_lock)          # Acquire H_lock in order to access Head
    node = Q->Head              # Read Head
    new_head = node->next        # Read next pointer
    if new_head == NULL          # Is queue empty?
        unlock(&Q->H_lock)       # Release H_lock before return
        return FALSE            # Queue was empty
    endif
    *pvalue = new_head->value     # Queue not empty. Read value before release
    Q->Head = new_head           # Swing Head to next node
    unlock(&Q->H_lock)          # Release H_lock
    free(node)                   # Free node
    return TRUE                # Queue was not empty, dequeue succeeded

```

Figure 2: Structure and operation of a two-lock concurrent queue.

5. *Tail* always points to a node in the linked list, because it never lags behind *Head*, so it can never point to a deleted node. Also, when *Tail* changes its value it always swings to the next node in the list and it never tries to change its value if the *next* pointer is NULL.

3.2 Linearizability

The presented algorithms are linearizable because there is a specific point during each operation at which it is considered to “take effect” [5]. An enqueue takes effect when the allocated node is linked to the last node in the linked list. A dequeue takes effect when *Head* swings to the next node. And, as shown in the previous subsection (properties 1, 4, and 5), the queue variables always reflect the state of the queue; they never enter a transient state in which the state of the queue can be mistaken (e.g. a non-empty queue

appears to be empty).

3.3 Liveness

The Lock-Free Algorithm is Non-Blocking

The lock-free algorithm is non-blocking because if there are non-delayed processes attempting to perform operations on the queue, an operation is guaranteed to complete within finite time.

An enqueue operation loops only if the condition in line E7 fails, the condition in line E8 fails, or the `compare_and_swap` in line E9 fails. A dequeue operation loops only if the condition in line D5 fails, the condition in line D6 holds (and the queue is not empty), or the `compare_and_swap` in line D13 fails.

We show that the algorithm is non-blocking by showing that a process loops beyond a finite number of times only if

another process completes an operation on the queue.

- The condition in line E7 fails only if *Tail* is written by an intervening process after executing line E5. *Tail* always points to the last or second to last node of the linked list, and when modified it follows the *next* pointer of the node it points to. Therefore, if the condition in line E7 fails more than once, then another process must have succeeded in completing an enqueue operation.
- The condition in line E8 fails if *Tail* was pointing to the second to last node in the linked-list. After the `compare_and_swap` in line E13, *Tail* must point to the last node in the list, unless a process has succeeded in enqueueing a new item. Therefore, if the condition in line E8 fails more than once, then another process must have succeeded in completing an enqueue operation.
- The `compare_and_swap` in line E9 fails only if another process succeeded in enqueueing a new item to the queue.
- The condition in line D5 and the `compare_and_swap` in line D13 fail only if *Head* has been written by another process. *Head* is written only when a process succeeds in dequeuing an item.
- The condition in line D6 succeeds (while the queue is not empty) only if *Tail* points to the second to last node in the linked list (in this case it is also the first node). After the `compare_and_swap` in line D10, *Tail* must point to the last node in the list, unless a process succeeded in enqueueing a new item. Therefore, if the condition of line D6 succeeds more than once, then another process must have succeeded in completing an enqueue operation (and the same or another process succeeded in dequeuing an item).

The Two-Lock Algorithm is Livelock-Free

The two-lock algorithm does not contain any loops. Therefore, if the mutual exclusion lock algorithm used for locking and unlocking the head and tail locks is livelock-free, then the presented algorithm is livelock-free too. There are many mutual exclusion algorithms that are livelock-free [12].

4 Performance

We use a 12-processor Silicon Graphics Challenge multiprocessor to compare the performance of the new algorithms to that of a single-lock algorithm, the algorithm of Prakash *et al.* [16], Valois's algorithm [24] (with corrections to the memory management mechanism [13]), and Mellor-Crummey's algorithm [11]. We include the algorithm of Prakash *et al.* because it appears to be the best of the known non-blocking alternatives. Mellor-Crummey's

algorithm represents non-lock-based but blocking alternatives; it is simpler than the code of Prakash *et al.*, and could be expected to display lower constant overhead in the absence of unpredictable process delays, but is likely to degenerate on a multiprogrammed system. We include Valois's algorithm to demonstrate that on multiprogrammed systems even a comparatively inefficient non-blocking algorithm can outperform blocking algorithms.

For the two lock-based algorithms we use `test-and-test_and_set` locks with bounded exponential back-off [1, 12]. We also use backoff where appropriate in the non-lock-based algorithms. Performance was not sensitive to the exact choice of backoff parameters in programs that do at least a modest amount of work between queue operations. We emulate both `test_and_set` and the atomic operations required by the other algorithms (`compare_and_swap`, `fetch_and_increment`, `fetch_and_decrement`, etc.) using the MIPS R4000 `load_linked` and `store_conditional` instructions.

To ensure the accuracy of the experimental results, we used the multiprocessor exclusively and prevented other users from accessing it during the experiments. To evaluate the performance of the algorithms under different levels of multiprogramming, we used a feature in the Challenge multiprocessor that allows programmers to associate processes with certain processors. For example, to represent a dedicated system where multiprogramming is not permitted, we created as many processes as the number of processors we wanted to use and locked each process to a different processor. And in order to represent a system with a multiprogramming level of 2, we created twice as many processes as the number of processors we wanted to use, and locked each pair of processes to an individual processor.

C code for the tested algorithms can be obtained from ftp://ftp.cs.rochester.edu/pub/packages/sched_conscious_synch/concurrent_queues. The algorithms were compiled at the highest optimization level, and were carefully hand-optimized. We tested each of the algorithms in hours-long executions on various numbers of processors. It was during this process that we discovered the race conditions mentioned in section 1.

All the experiments employ an initially-empty queue to which processes perform a series of enqueue and dequeue operations. Each process enqueues an item, does “other work”, dequeues an item, does “other work”, and repeats. With p processes, each process executes this loop $\lceil 10^6/p \rceil$ or $\lceil 10^6/p \rceil$ times, for a total of one million enqueues and dequeues. The “other work” consists of approximately $6\ \mu s$ of spinning in an empty loop; it serves to make the experiments more realistic by preventing long runs of queue operations by the same process (which would display overly-optimistic performance due to an unrealistically low cache miss rate). We subtracted the time required for one processor to com-

plete the “other work” from the total time reported in the figures.

Figure 3 shows net elapsed time in seconds for one million enqueue/dequeue pairs. Roughly speaking, this corresponds to the time in microseconds for one enqueue/dequeue pair. More precisely, for k processors, the graph shows the time one processor spends performing $10^6/k$ enqueue/dequeue pairs, plus the amount by which the critical path of the other $10^6(k-1)/k$ pairs performed by other processors exceeds the time spent by the first processor in “other work” and loop overhead. For $k = 1$, the second term is zero. As k increases, the first term shrinks toward zero, and the second term approaches the critical path length of the overall computation; i.e. one million times the *serial portion* of an enqueue/dequeue pair. Exactly how much execution will overlap in different processors depends on the choice of algorithm, the number of processors k , and the length of the “other work” between queue operations.

With only one processor, memory references in all but the first loop iteration hit in the cache, and completion times are very low. With two processors active, contention for head and tail pointers and queue elements causes a high fraction of references to miss in the cache, leading to substantially higher completion times. The queue operations of processor 2, however, fit into the “other work” time of processor 1, and vice versa, so we are effectively measuring the time for one processor to complete 5×10^5 enqueue/dequeue pairs. At three processors, the cache miss rate is about the same as it was with two processors. Each processor only has to perform $10^6/3$ enqueue/dequeue pairs, but some of the operations of the other processors no longer fit in the first processor’s “other work” time. Total elapsed time decreases, but by a fraction less than 1/3. Toward the right-hand side of the graph, execution time rises for most algorithms as smaller and smaller amounts of per-processor “other work” and loop overhead are subtracted from a total time dominated by critical path length. In the single-lock and Mellor-Crummey curves, the increase is probably accelerated as high rates of contention increase the average cost of a cache miss. In Valois’s algorithm, the plotted time continues to decrease, as more and more of the memory management overhead moves out of the critical path and into the overlapped part of the computation.

Figures 4 and 5 plot the same quantity as figure 3, but for a system with 2 and 3 processes per processor, respectively. The operating system multiplexes the processor among processes with a scheduling quantum of 10 ms. As expected, the blocking algorithms fare much worse in the presence of multiprogramming, since an inopportune preemption can block the progress of every process in the system. Also as expected, the degree of performance degradation increases with the level of multiprogramming.

In all three graphs, the new non-blocking queue outperforms all of the other alternatives when three or more processors are active. Even for one or two processors, its per-

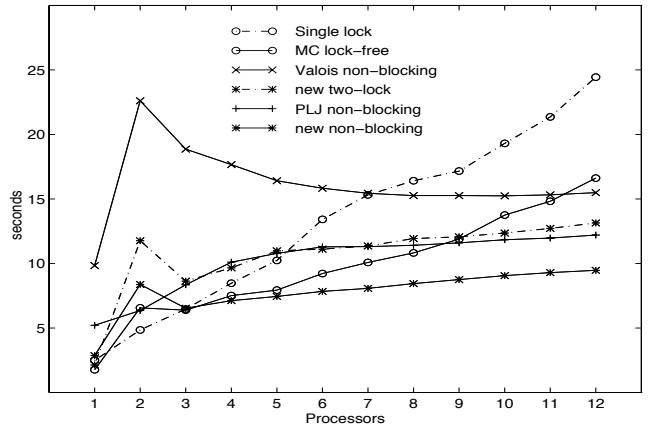


Figure 3: Net execution time for one million enqueue/dequeue pairs on a dedicated multiprocessor.

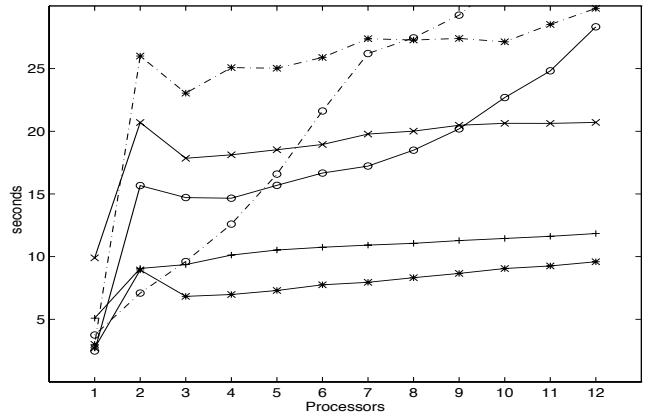


Figure 4: Net execution time for one million enqueue/dequeue pairs on a multiprogrammed system with 2 processes per processor.

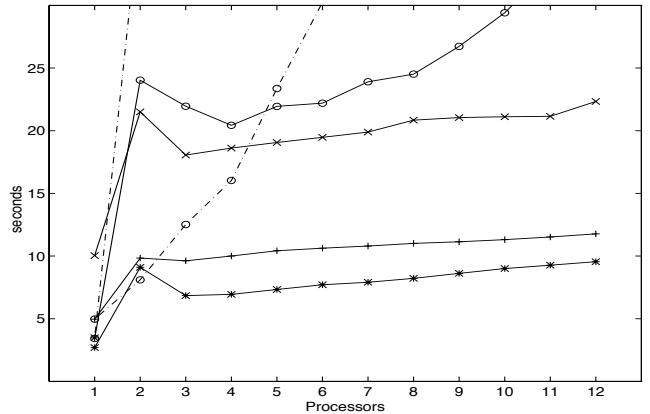


Figure 5: Net execution time for one million enqueue/dequeue pairs on a multiprogrammed system with 3 processes per processor.

formance is good enough that we can comfortably recommend its use in all situations. The two-lock algorithm outperforms the one-lock algorithm when more than 5 processors are active on a dedicated system: it appears to be a reasonable choice for machines that are not multiprogrammed, and that lack a universal atomic primitive (`compare_and_swap` or `load_linked/store_conditional`).

5 Conclusions

Queues are ubiquitous in parallel programs, and their performance is a matter of major concern. We have presented a concurrent queue algorithm that is simple, non-blocking, practical, and fast. We were surprised not to find it in the literature. It seems to be the algorithm of choice for any queue-based application on a multiprocessor with universal atomic primitives (e.g. `compare_and_swap` or `load_linked/store_conditional`).

We have also presented a queue with separate head and tail pointer locks. Its structure is similar to that of the non-blocking queue, but it allows only one enqueue and one dequeue to proceed at a given time. Because it is based on locks, however, it will work on machines with such simple atomic primitives as `test_and_set`. We recommend it for heavily-utilized queues on such machines (For a queue that is usually accessed by only one or two processors, a single lock will run a little faster.)

This work is part of a larger project that seeks to evaluate the tradeoffs among alternative mechanisms for atomic update of common data structures. Structures under consideration include stacks, queues, heaps, search trees, and hash tables. Mechanisms include single locks, data-structure-specific multi-lock algorithms, general-purpose and special-purpose non-blocking algorithms, and function shipping to a centralized manager (a valid technique for situations in which remote access latencies dominate computation time).

In related work [8, 25, 26], we have been developing general-purpose synchronization mechanisms that cooperate with a scheduler to avoid inopportune preemption. Given that immunity to processes delays is a primary benefit of non-blocking parallel algorithms, we plan to compare these two approaches in the context of multiprogrammed systems.

References

- [1] T. E. Anderson. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.
- [2] G. Barnes. A Method for Implementing Lock-Free Data Structures. In *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*, Velen, Germany, June –July 1993.
- [3] A. Gottlieb, B. D. Lubachevsky, and L. Rudolph. Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors. *ACM Transactions on Programming Languages and Systems*, 5(2):164–189, April 1983.
- [4] M. P. Herlihy and J. M. Wing. Axions for Concurrent Objects. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, pages 13–26, January 1987.
- [5] 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, July 1990.
- [6] M. Herlihy. A Methodology for Implementing Highly Concurrent Data Objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.
- [7] K. Hwang and F. A. Briggs. *Computer Architecture and Parallel Processing*. McGraw-Hill, 1984.
- [8] L. Kontothanassis and R. Wisniewski. Using Scheduler Information to Achieve Optimal Barrier Synchronization Performance. In *Proceedings of the Fourth ACM Symposium on Principles and Practice of Parallel Programming*, May 1993.
- [9] L. Lamport. Specifying Concurrent Program Modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, April 1983.
- [10] H. Massalin and C. Pu. A Lock-Free Multiprocessor OS Kernel. Technical Report CU-CS-005-91, Computer Science Department, Columbia University, 1991.
- [11] J. M. Mellor-Crummey. Concurrent Queues: Practical Fetch-and-Φ Algorithms. TR 229, Computer Science Department, University of Rochester, November 1987.
- [12] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.
- [13] M. M. Michael and M. L. Scott. Correction of a Memory Management Method for Lock-Free Data Structures. Technical Report 599, Computer Science Department, University of Rochester, December 1995.
- [14] S. Prakash, Y. H. Lee, and T. Johnson. A Non-Blocking Algorithm for Shared Queues Using Compare-and-Swap. In *Proceedings of the 1991 International Conference on Parallel Processing*, pages II:68–75, 1991.

- [15] S. Prakash, Y. H. Lee, and T. Johnson. Non-Blocking Algorithms for Concurrent Data Structures. Technical Report 91-002, University of Florida, 1991.
- [16] S. Prakash, Y. H. Lee, and T. Johnson. A Nonblocking Algorithm for Shared Queues Using Compare-and-Swap. *IEEE Transactions on Computers*, 43(5):548–559, May 1994.
- [17] R. Sites. Operating Systems and Computer Architecture. In *H. Stone, editor, Introduction to Computer Architecture, 2nd edition, Chapter 12*, 1980. Science Research Associates.
- [18] J. M. Stone. A Simple and Correct Shared-Queue Algorithm Using Compare-and-Swap. In *Proceedings Supercomputing '90*, November 1990.
- [19] J. M. Stone. A Non-Blocking Compare-and-Swap Algorithm for a Shared Circular Queue. In *S. Tzafestas et al., editors, Parallel and Distributed Computing in Engineering Systems*, pages 147–152, 1992. Elsevier Science Publishers.
- [20] H. S. Stone. *High Performance Computer Architecture*. Addison-Wesley, 1993.
- [21] R. K. Treiber. Systems Programming: Coping with Parallelism. In *RJ 5118, IBM Almaden Research Center*, April 1986.
- [22] J. Turek, D. Shasha, and S. Prakash. Locking without Blocking: Making Lock Based Concurrent Data Structure Algorithms Nonblocking. In *Proceedings of the 11th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 212–222, 1992.
- [23] J. D. Valois. Implementing Lock-Free Queues. In *Seventh International Conference on Parallel and Distributed Computing Systems*, Las Vegas, NV, October 1994.
- [24] J. D. Valois. Lock-Free Data Structures. Ph.D. dissertation, Rensselaer Polytechnic Institute, May 1995.
- [25] R. W. Wisniewski, L. Kontothanassis, and M. L. Scott. Scalable Spin Locks for Multiprogrammed Systems. In *Proceedings of the Eighth International Parallel Processing Symposium*, pages 583–589, Cancun, Mexico, April 1994. Earlier but expanded version available as TR 454, Computer Science Department, University of Rochester, April 1993.
- [26] R. W. Wisniewski, L. I. Kontothanassis, and M. L. Scott. High Performance Synchronization Algorithms for Multiprogrammed Multiprocessors. In *Proceedings of the Fifth ACM Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.

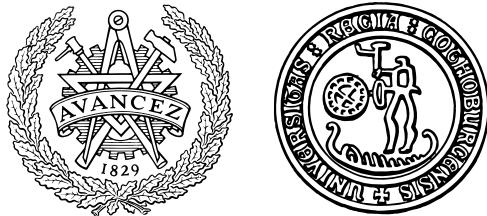
Technical Report no. 2003-01

Fast and Lock-Free Concurrent Priority Queues for Multi-Thread Systems¹

Håkan Sundell

Philippas Tsigas

CHALMERS | GÖTEBORG UNIVERSITY



Department of Computing Science
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

Göteborg, 2003

¹This work is partially funded by: i) the national Swedish Real-Time Systems research initiative ARTES (www.artes.uu.se) supported by the Swedish Foundation for Strategic Research and ii) the Swedish Research Council for Engineering Sciences.



Technical Report in Computing Science at
Chalmers University of Technology and Göteborg University

Technical Report no. 2003-01
ISSN: 1650-3023

Department of Computing Science
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

Göteborg, Sweden, 2003

Abstract

We present an efficient and practical lock-free implementation of a concurrent priority queue that is suitable for both fully concurrent (large multi-processor) systems as well as pre-emptive (multi-process) systems. Many algorithms for concurrent priority queues are based on mutual exclusion. However, mutual exclusion causes blocking which has several drawbacks and degrades the system's overall performance. Non-blocking algorithms avoid blocking, and are either lock-free or wait-free. Previously known non-blocking algorithms of priority queues did not perform well in practice because of their complexity, and they are often based on non-available atomic synchronization primitives. Our algorithm is based on the randomized sequential list structure called SkipList, and a real-time extension of our algorithm is also described. In our performance evaluation we compare our algorithm with some of the most efficient implementations of priority queues known. The experimental results clearly show that our lock-free implementation outperforms the other lock-based implementations in all cases for 3 threads and more, both on fully concurrent as well as on pre-emptive systems.

1 Introduction

Priority queues are fundamental data structures. From the operating system level to the user application level, they are frequently used as basic components. For example, the ready-queue that is used in the scheduling of tasks in many real-time systems, can usually be implemented using a concurrent priority queue. Consequently, the design of efficient implementations of priority queues is a research area that has been extensively researched. A priority queue supports two operations, the *Insert* and the *DeleteMin* operation. The abstract definition of a priority queue is a set of key-value pairs, where the key represents a priority. The *Insert* operation inserts a new key-value pair into the set, and the *DeleteMin* operation removes and returns the value of the key-value pair with the lowest key (i.e. highest priority) that was in the set.

To ensure consistency of a shared data object in a concurrent environment, the most common method is to use mutual exclusion, i.e. some form of locking. Mutual exclusion degrades the system's overall performance [14] as it causes blocking, i.e. other concurrent operations can not make any progress while the access to the shared resource is blocked by the lock. Using mutual exclusion can also cause deadlocks, priority inversion (which can be solved efficiently on uni-processors [13] with the cost of more difficult analysis, although not as efficient on multiprocessor systems [12]) and even starvation.

To address these problems, researchers have proposed non-blocking algorithms for shared data objects. Non-blocking methods do not involve mutual exclusion, and therefore do not suffer from the problems that blocking can cause. Non-blocking algorithms are either lock-free or wait-free. Lock-free implementations guarantee that regardless of the contention caused by concurrent operations and the interleaving of their sub-operations, always at least one operation will progress. However, there is a risk for starvation as the progress of other operations could cause one specific operation to never finish. This is although different from the type of starvation that could be caused by blocking, where a single operation could block every other operation forever, and cause starvation of the whole system. Wait-free [6] algorithms are lock-free and moreover they avoid starvation as well, in a wait-free algorithm every operation is guaranteed to finish in a limited number of steps, regardless of the actions of the concurrent operations. Non-blocking algorithms have been shown to be of big practical importance in practical applications [17, 18], and recently NOBLE, which is a non-blocking inter-process communication library, has been introduced [16].

There exist several algorithms and implementations of concurrent priority queues. The majority of the algorithms are lock-based, either with a single lock on top of a sequential algorithm, or specially constructed algorithms using multiple locks, where each lock protects a small part of the shared data structure. Several different representations of the shared data structure are used, for example: Hunt *et al.* [7] presents an implementation which is based on heap structures, Grammatikakis *et al.* [3] compares different structures including cyclic arrays and heaps, and most recently Lotan and Shavit [9] presented an implementation based on the SkipList structure [11]. The algorithm by Hunt *et al.* locks each node separately and uses a technique to scatter the accesses to the heap, thus reducing the contention. Its implementation is publicly available and its performance has been documented on multi-processor systems. Lotan and Shavit extend the functionality of the concurrent priority queue and assume the availability of a global high-accuracy clock. They apply a lock on each pointer, and as the multi-pointer based SkipList structure is used, the number of locks is significantly more than the number of nodes. Its performance has previously only been documented by simulation, with very promising results.

Israeli and Rappoport have presented a wait-free algorithm for a concurrent priority queue [8]. This algorithm makes use of strong atomic synchronization primitives that have not been implemented in any currently existing platform. However, there exists an attempt for a wait-free algorithm by Barnes [2] that uses existing atomic primitives, though this algorithm does not comply with the generally accepted definition of the wait-free property. The algo-

rithm is not yet implemented and the theoretical analysis predicts worse behavior than the corresponding sequential algorithm, which makes it not of practical interest.

One common problem with many algorithms for concurrent priority queues is the lack of precise defined semantics of the operations. It is also seldom that the correctness with respect to concurrency is proved, using a strong property like linearizability [5].

In this paper we present a lock-free algorithm of a concurrent priority queue that is designed for efficient use in both pre-emptive as well as in fully concurrent environments. Inspired by Lotan and Shavit [9], the algorithm is based on the randomized SkipList [11] data structure, but in contrast to [9] it is lock-free. It is also implemented using common synchronization primitives that are available in modern systems. The algorithm is described in detail later in this paper, and the aspects concerning the underlying lock-free memory management are also presented. The precise semantics of the operations are defined and a proof is given that our implementation is lock-free and linearizable. We have performed experiments that compare the performance of our algorithm with some of the most efficient implementations of concurrent priority queues known, i.e. the implementation by Lotan and Shavit [9] and the implementation by Hunt *et al.* [7]. Experiments were performed on three different platforms, consisting of a multiprocessor system using different operating systems and equipped with either 2, 4 or 64 processors. Our results show that our algorithm outperforms the other lock-based implementations for 3 threads and more, in both highly pre-emptive as well as in fully concurrent environments. We also present an extended version of our algorithm that also addresses certain real-time aspects of the priority queue as introduced by Lotan and Shavit [9].

The rest of the paper is organized as follows. In Section 2 we define the properties of the systems that our implementation is aimed for. The actual algorithm is described in Section 3. In Section 4 we define the precise semantics for the operations on our implementations, as well showing correctness by proving the lock-free and linearizability property. The experimental evaluation that shows the performance of our implementation is presented in Section 5. In Section 6 we extend our algorithm with functionality that can be needed for specific real-time applications. We conclude the paper with Section 7.

2 System Description

A typical abstraction of a shared memory multiprocessor system configuration is depicted in Figure 1. Each node of the system contains a processor together with its local memory. All nodes are connected to the shared memory via an interconnection network. A set of co-

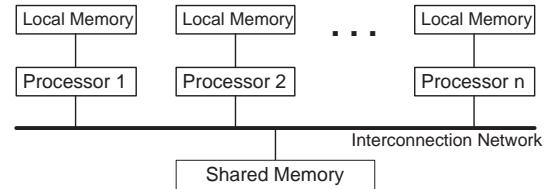


Figure 1. Shared Memory Multiprocessor System Structure

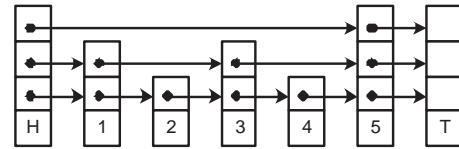


Figure 2. The SkipList data structure with 5 nodes inserted.

operating tasks is running on the system performing their respective operations. Each task is sequentially executed on one of the processors, while each processor can serve (run) many tasks at a time. The co-operating tasks, possibly running on different processors, use shared data objects built in the shared memory to co-ordinate and communicate. Tasks synchronize their operations on the shared data objects through sub-operations on top of a cache-coherent shared memory. The shared memory may not though be uniformly accessible for all nodes in the system; processors can have different access times on different parts of the memory.

3 Algorithm

The algorithm is based on the sequential SkipList data structure invented by Pugh [11]. This structure uses randomization and has a probabilistic time complexity of $O(\log N)$ where N is the maximum number of elements in the list. The data structure is basically an ordered list with randomly distributed short-cuts in order to improve search times, see Figure 2. The maximum height (i.e. the maxi-

```
structure Node
    key,level,validLevel <,timeInsert> : integer
    value : pointer to word
    next[level],prev : pointer to Node
```

Figure 3. The Node structure.

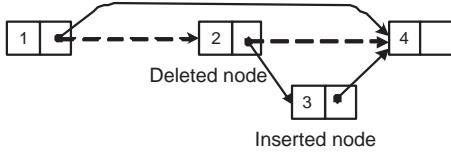


Figure 4. Concurrent insert and delete operation can delete both nodes.

```

function TAS(value:pointer to word):boolean
  atomic do
    if *value=0 then
      *value:=1;
      return true;
    else return false;

procedure FAA(address:pointer to word, number:integer)
  atomic do
    *address := *address + number;

function CAS(address:pointer to word, oldvalue:word,
  newvalue:word):boolean
  atomic do
    if *address = oldvalue then
      *address := newvalue;
      return true;
    else return false;

```

Figure 5. The Test-And-Set (TAS), Fetch-And-Add (FAA) and Compare-And-Swap (CAS) atomic primitives.

mum number of next pointers) of the data structure is $\log N$. The height of each inserted node is randomized geometrically in the way that 50% of the nodes should have height 1, 25% of the nodes should have height 2 and so on. To use the data structure as a priority queue, the nodes are ordered in respect of priority (which has to be unique for each node), the nodes with highest priority are located first in the list. The fields of each node item are described in Figure 3 as it is used in this implementation. For all code examples in this paper, code that is between the “⟨” and “⟩” symbols are only used for the special implementation that involves timestamps (see Section 6), and are thus not included in the standard version of the implementation.

In order to make the Skiplist construction concurrent and non-blocking, we are using three of the standard atomic synchronization primitives, Test-And-Set (TAS), Fetch-And-Add (FAA) and Compare-And-Swap (CAS). Figure 5 describes the specification of these primitives which are available in most modern platforms.

As we are concurrently (with possible preemptions) traversing nodes that will be continuously allocated and reclaimed, we have to consider several aspects of memory management. No node should be reclaimed and then later re-allocated while some other process is traversing this node. This can be solved for example by careful reference counting. We have selected to use the lock-free memory management scheme invented by Valois [19] and corrected by Michael and Scott [10], which makes use of the FAA and CAS atomic synchronization primitives.

To insert or delete a node from the list we have to change the respective set of next pointers. These have to be changed consistently, but not necessary all at once. Our solution is to have additional information on each node about its deletion (or insertion) status. This additional information will guide the concurrent processes that might traverse into one partial deleted or inserted node. When we have changed all necessary next pointers, the node is fully deleted or inserted.

One problem, that is general for non-blocking implementations that are based on the linked-list structure, arises when inserting a new node into the list. Because of the linked-list structure one has to make sure that the previous node is not about to be deleted. If we are changing the next pointer of this previous node atomically with CAS, to point to the new node, and then immediately afterwards the previous node is deleted - then the new node will be deleted as well, as illustrated in Figure 4. There are several solutions to this problem. One solution is to use the CAS2 operation as it can change two pointers atomically, but this operation is not available in any existing multiprocessor system. A second solution is to insert auxiliary nodes [19] between each two normal nodes, and the latest method introduced by Harris [4] is to use one bit of the pointer values as a deletion mark. On most modern 32-bit systems, 32-bit values can only be located at addresses that are evenly dividable by 4, therefore bits 0 and 1 of the address are always set to zero. The method is then to use the previously unused bit 0 of the next pointer to mark that this node is about to be deleted, using CAS. Any concurrent *Insert* operation will then be notified about the deletion, when its CAS operation will fail.

One memory management issue is how to de-reference pointers safely. If we simply de-reference the pointer, it might be that the corresponding node has been reclaimed before we could access it. It can also be that bit 0 of the pointer was set, thus marking that the node is deleted, and therefore the pointer is not valid. The following functions are defined for safe handling of the memory management:

```

function READ_NODE(address:pointer to pointer to
  Node):pointer to Node /* De-reference the pointer and in-
    crease the reference counter for the corresponding node. In
    case the pointer is marked, NULL is returned */

```

```

// Global variables
head,tail : pointer to Node
// Local variables
node2 : pointer to Node

function ReadNext(node1:pointer to pointer to Node,
level:integer):pointer to Node
R1  if IS_MARKED((*node1).value) then
R2    *node1:=HelpDelete(*node1,level);
R3  node2:=READ_NODE((*node1).next[level]);
R4  while node2=NULL do
R5    *node1:=HelpDelete(*node1,level);
R6    node2:=READ_NODE((*node1).next[level]);
R7  return node2;

function ScanKey(node1:pointer to pointer to Node,
level:integer, key:integer):pointer to Node
S1  node2:=ReadNext(node1,level);
S2  while node2.key < key do
S3    RELEASE_NODE(*node1);
S4    *node1:=node2;
S5    node2:=ReadNext(node1,level);
S6  return node2;

```

Figure 6. Functions for traversing the nodes in the SkipList data structure.

```

function COPY_NODE(node:pointer to Node):pointer to Node /* Increase the reference counter for the corresponding given node */
procedure RELEASE_NODE(node:pointer to Node)
/* Decrement the reference counter on the corresponding given node. If the reference count reaches zero, then call RELEASE_NODE on the nodes that this node has owned pointers to, then reclaim the node */

```

While traversing the nodes, processes will eventually reach nodes that are marked to be deleted. As the process that invoked the corresponding *Delete* operation might be pre-empted, this *Delete* operation has to be helped to finish before the traversing process can continue. However, it is only necessary to help the part of the *Delete* operation on the current level in order to be able to traverse to the next node. The function *ReadNext*, see Figure 6, traverses to the next node of *node1* on the given level while helping (and then sets *node1* to the previous node of the helped one) any marked nodes in between to finish the deletion. The function *ScanKey*, see Figure 6, traverses in several steps through the next pointers (starting from *node1*) at the current level until it finds a node that has the same or higher key (priority) value than the given key. It also sets *node1* to be the previous node of the returned node.

The implementation of the *Insert* operation, see Fig-

```

// Local variables
node1,node2,newNode,
savedNodes[maxlevel]: pointer to Node

function Insert(key:integer, value:pointer to word):boolean
I1  {TraverseTimeStamps();}
I2  level:=randomLevel();
I3  newNode:=CreateNode(level,key,value);
I4  COPY_NODE(newNode);
I5  node1:=COPY_NODE(head);
I6  for i:=maxLevel-1 to 1 step -1 do
I7    node2:=ScanKey(&node1,i,key);
I8    RELEASE_NODE(node2);
I9    if i<level then savedNodes[i]:=COPY_NODE(node1);
I10 while true do
I11   node2:=ScanKey(&node1,0,key);
I12   value2:=node2.value;
I13   if not IS_MARKED(value2) and node2.key=key then
I14     if CAS(&node2.value,value2,value) then
I15       RELEASE_NODE(node1);
I16       RELEASE_NODE(node2);
I17       for i:=1 to level-1 do
I18         RELEASE_NODE(savedNodes[i]);
I19       RELEASE_NODE(newNode);
I20       RELEASE_NODE(newNode);
I21   return true2;
I22   else
I23     RELEASE_NODE(node2);
I24     continue;
I25   newNode.next[0]:=node2;
I26   RELEASE_NODE(node2);
I27   if CAS(&node1.next[0],node2,newNode) then
I28     RELEASE_NODE(node1);
I29     break;
I30   Back-Off
I31   for i:=1 to level-1 do
I32     newNode.validLevel:=i;
I33     node1:=savedNodes[i];
I34   while true do
I35     node2:=ScanKey(&node1,i,key);
I36     newNode.next[i]:=node2;
I37     RELEASE_NODE(node2);
I38     if IS_MARKED(newNode.value) or
I39       CAS(&node1.next[i],node2,newNode) then
I40         RELEASE_NODE(node1);
I41         break;
I42       Back-Off
I43     newNode.validLevel:=level;
I44   if IS_MARKED(newNode.value) then
I45     newNode:=HelpDelete(newNode,0);
I46   RELEASE_NODE(newNode);
I47   return true;

```

Figure 7. Insert

```

// Local variables
prev,last,node1,node2 : pointer to Node

function DeleteMin():pointer to Node
D1  {TraverseTimeStamps();}
D2  {time:=getNextTimeStamp();}
D3  prev:=COPY_NODE(head);
D4  while true do
D5    node1:=ReadNext(&prev,0);
D6    if node1=tail then
D7      RELEASE_NODE(prev);
D8      RELEASE_NODE(node1);
D9      return NULL;
    retry:
D10   value:=node1.value;
D11   if not IS_MARKED(value) <and
        compareTimeStamp(time,node1.timeInsert)>0 then
D12     if CAS(&node1.value,value,
            GET_MARKED(value)) then
D13       node1.prev:=prev;
D14       break;
D15     else goto retry;
D16   else if IS_MARKED(value) then
D17     node1:=HelpDelete(node1,0);
D18     RELEASE_NODE(prev);
D19     prev:=node1;
D20   for i:=0 to node1.level-1 do
D21     repeat
D22       node2:=node1.next[i];
D23     until IS_MARKED(node2) or CAS(
            &node1.next[i],node2,GET_MARKED(node2));
D24   prev:=COPY_NODE(head);
D25   for i:=node1.level-1 to 0 step -1 do
D26     while true do
D27       if node1.next[i]=1 then break;
D28       last:=ScanKey(&prev,i,node1.key);
D29       RELEASE_NODE(last);
D30       if last≠node1 or node1.next[i]=1 then break;
D31       if CAS(&prev.next[i],node1,
            GET_UNMARKED(node1.next[i])) then
D32         node1.next[i]:=1;
D33         break;
D34       if node1.next[i]=1 then break;
D35       Back-Off
D36     RELEASE_NODE(prev);
D37     RELEASE_NODE(node1);
D38     RELEASE_NODE(node1); /* Delete the node */
D39   return value;

```

Figure 8. DeleteMin

```

// Local variables
prev,last,node2 : pointer to Node

function HelpDelete(node:pointer to Node,
level:integer):pointer to Node
H1  for i:=level to node.level-1 do
H2    repeat
H3      node2:=node.next[i];
H4    until IS_MARKED(node2) or CAS(
        &node.next[i],node2,GET_MARKED(node2));
H5  prev:=node.prev;
H6  if not prev or level ≥ prev.validLevel then
H7    prev:=COPY_NODE(head);
H8    for i:=maxLevel-1 to level step -1 do
H9      node2:=ScanKey(&prev,i,node.key);
H10     RELEASE_NODE(node2);
H11   else COPY_NODE(prev);
H12   while true do
H13     if node.next[level]=1 then break;
H14     last:=ScanKey(&prev,level,node.key);
H15     RELEASE_NODE(last);
H16     if last≠node or node.next[level]=1 then break;
H17     if CAS(&prev.next[level],node,
            GET_UNMARKED(node.next[level])) then
H18       node.next[level]:=1;
H19       break;
H20     if node.next[level]=1 then break;
H21     Back-Off
H22   RELEASE_NODE(node);
H23   return prev;

```

Figure 9. HelpDelete

ure 7, starts in lines I5-I11 with a search phase to find the node after which the new node (*newNode*) should be inserted. This search phase starts from the head node at the highest level and traverses down to the lowest level until the correct node is found (*node1*). When going down one level, the last node traversed on that level is remembered (*savedNodes*) for later use (this is where we should insert the new node at that level). Now it is possible that there already exists a node with the same priority as of the new node, this is checked in lines I12-I24, the value of the old node (*node2*) is changed atomically with a CAS. Otherwise, in lines I25-I42 it starts trying to insert the new node starting with the lowest level and increasing up to the level of the new node. The next pointers of the nodes (to become previous) are changed atomically with a CAS. After the new node has been inserted at the lowest level, it is possible that it is deleted by a concurrent *DeleteMin* operation before it has been inserted at all levels, and this is checked in lines I38 and I44.

The *DeleteMin* operation, see Figure 8, starts from the head node and finds the first node (*node1*) in the list that

does not have its deletion mark on the value set, see lines D3-D12. It tries to set this deletion mark in line D12 using the CAS primitive, and if it succeeds it also writes a valid pointer to the prev field of the node. This prev field is necessary in order to increase the performance of concurrent *HelpDelete* operations, these operations otherwise would have to search for the previous node in order to complete the deletion. The next step is to mark the deletion bits of the next pointers in the node, starting with the lowest level and going upwards, using the CAS primitive in each step, see lines D20-D23. Afterwards in lines D24-D35 it starts the actual deletion by changing the next pointers of the previous node (*prev*), starting at the highest level and continuing downwards. The reason for doing the deletion in decreasing order of levels, is that concurrent search operations also start at the highest level and proceed downwards, in this way the concurrent search operations will sooner avoid traversing this node. The procedure performed by the *DeleteMin* operation in order to change each next pointer of the previous node, is to first search for the previous node and then perform the CAS primitive until it succeeds.

The algorithm has been designed for pre-emptive as well as fully concurrent systems. In order to achieve the lock-free property (that at least one thread is doing progress) on pre-emptive systems, whenever a search operation finds a node that is about to be deleted, it calls the *HelpDelete* operation and then proceeds searching from the previous node of the deleted. The *HelpDelete* operation, see Figure 9, tries to fulfill the deletion on the current level and returns when it is completed. It starts in lines H1-H4 with setting the deletion mark on all next pointers in case they have not been set. In lines H5-H6 it checks if the node given in the prev field is valid for deletion on the current level, otherwise it searches for the correct node (*prev*) in lines H7-H10. The actual deletion of this node on the current level takes place in lines H12-H21. This operation might execute concurrently with the corresponding *DeleteMin* operation, and therefore both operations synchronize with each other in lines D27, D30, D32, D34, H13, H16, H18 and H20 in order to avoid executing sub-operations that have already been performed.

In fully concurrent systems though, the helping strategy can downgrade the performance significantly. Therefore the algorithm, after a number of consecutive failed attempts to help concurrent *DeleteMin* operations that hinders the progress of the current operation, puts the operation into back-off mode. When in back-off mode, the thread does nothing for a while, and in this way avoids disturbing the concurrent operations that might otherwise progress slower. The duration of the back-off is proportional to the number of threads, and for each consecutive entering of back-off mode during one operation invocation, the duration is increased exponentially.

4 Correctness

In this section we present the proof of our algorithm. We first prove that our algorithm is a linearizable one [5] and then we prove that it is lock-free. A set of definitions that will help us to structure and shorten the proof is first explained in this section. We start by defining the sequential semantics of our operations and then introduce two definitions concerning concurrency aspects in general.

Definition 1 *We denote with L_t the abstract internal state of a priority queue at the time t . L_t is viewed as a set of pairs $\langle p, v \rangle$ consisting of a unique priority p and a corresponding value v . The operations that can be performed on the priority queue are Insert (I) and DeleteMin (DM). The time t_1 is defined as the time just before the atomic execution of the operation that we are looking at, and the time t_2 is defined as the time just after the atomic execution of the same operation. The return value of true_2 is returned by an Insert operation that has succeeded to update an existing node, the return value of true is returned by an Insert operation that succeeds to insert a new node. In the following expressions that defines the sequential semantics of our operations, the syntax is $S_1 : O_1, S_2$, where S_1 is the conditional state before the operation O_1 , and S_2 is the resulting state after performing the corresponding operation:*

$$\langle p_1, _ \rangle \notin L_{t_1} : I_1(\langle p_1, v_1 \rangle) = \text{true}, \\ L_{t_2} = L_{t_1} \cup \{\langle p_1, v_1 \rangle\} \quad (1)$$

$$\langle p_1, v_1 \rangle \in L_{t_1} : I_1(\langle p_1, v_1 \rangle) = \text{true}_2, \\ L_{t_2} = L_{t_1} \setminus \{\langle p_1, v_1 \rangle\} \cup \{\langle p_1, v_1 \rangle\} \quad (2)$$

$$\langle p_1, v_1 \rangle = \{\langle \min p, v \rangle | \langle p, v \rangle \in L_{t_1}\} \\ : DM_1() = \langle p_1, v_1 \rangle, L_{t_2} = L_{t_1} \setminus \{\langle p_1, v_1 \rangle\} \quad (3)$$

$$L_{t_1} = \emptyset : DM_1() = \perp \quad (4)$$

Definition 2 *In a global time model each concurrent operation Op “occupies” a time interval $[b_{Op}, f_{Op}]$ on the linear time axis ($b_{Op} < f_{Op}$). The precedence relation (denoted by ‘ \rightarrow ’) is a relation that relates operations of a possible execution, $Op_1 \rightarrow Op_2$ means that Op_1 ends before Op_2 starts. The precedence relation is a strict partial order. Operations incomparable under \rightarrow are called overlapping. The overlapping relation is denoted by \parallel and is commutative, i.e. $Op_1 \parallel Op_2$ and $Op_2 \parallel Op_1$. The precedence relation is extended to relate sub-operations of operations. Consequently, if $Op_1 \rightarrow Op_2$, then for any sub-operations op_1 and op_2 of Op_1 and Op_2 , respectively, it holds that $op_1 \rightarrow op_2$. We also define the direct precedence relation \rightarrow_d , such that if $Op_1 \rightarrow_d Op_2$, then $Op_1 \rightarrow Op_2$ and moreover there exists no operation Op_3 such that $Op_1 \rightarrow Op_3 \rightarrow Op_2$.*

Definition 3 In order for an implementation of a shared concurrent data object to be linearizable [5], for every concurrent execution there should exist an equal (in the sense of the effect) and valid (i.e. it should respect the semantics of the shared data object) sequential execution that respects the partial order of the operations in the concurrent execution.

Next we are going to study the possible concurrent executions of our implementation. First we need to define the interpretation of the abstract internal state of our implementation.

Definition 4 The pair $\langle p, v \rangle$ is present ($\langle p, v \rangle \in L$) in the abstract internal state L of our implementation, when there is a next pointer from a present node on the lowest level of the Skiplist that points to a node that contains the pair $\langle p, v \rangle$, and this node is not marked as deleted with the mark on the value.

Lemma 1 The definition of the abstract internal state for our implementation is consistent with all concurrent operations examining the state of the priority queue.

Proof: As the next and value pointers are changed using the CAS operation, we are sure that all threads see the same state of the Skiplist, and therefore all changes of the abstract internal state seems to be atomic. \square

Definition 5 The decision point of an operation is defined as the atomic statement where the result of the operation is finitely decided, i.e. independent of the result of any sub-operations proceeding the decision point, the operation will have the same result. We define the state-read point of an operation to be the atomic statement where a sub-state of the priority queue is read, and this sub-state is the state on which the decision point depends. We also define the state-change point as the atomic statement where the operation changes the abstract internal state of the priority queue after it has passed the corresponding decision point.

We will now show that all of these points conform to the very same statement, i.e. the linearizability point.

Lemma 2 An Insert operation which succeeds ($I(\langle p, v \rangle) = \text{true}$), takes effect atomically at one statement.

Proof: The decision point for an Insert operation which succeeds ($I(\langle p, v \rangle) = \text{true}$), is when the CAS sub-operation in line I27 (see Figure 7) succeeds, all following CAS sub-operations will eventually succeed, and the Insert operation will finally return true. The state of the list (L_{t_1}) directly before the passing of the decision point must have been $\langle p, _ \rangle \notin L_{t_1}$, otherwise the CAS would

have failed. The state of the list directly after passing the decision point will be $\langle p, v \rangle \in L_{t_2}$. \square

Lemma 3 An Insert operation which updates ($I(\langle p, v \rangle) = \text{true}_2$), takes effect atomically at one statement.

Proof: The decision point for an Insert operation which updates ($I(\langle p, v \rangle) = \text{true}_2$), is when the CAS will succeed in line I14. The state of the list (L_{t_1}) directly before passing the decision point must have been $\langle p, _ \rangle \in L_{t_1}$, otherwise the CAS would have failed. The state of the list directly after passing the decision point will be $\langle p, v \rangle \in L_{t_3}$. \square

Lemma 4 A DeleteMin operation which succeeds ($D() = \langle p, v \rangle$), takes effect atomically at one statement.

Proof: The decision point for an DeleteMin operation which succeeds ($D() = \langle p, v \rangle$) is when the CAS sub-operation in line D12 (see Figure 8) succeeds. The state of the list (L_t) directly before passing of the decision point must have been $\langle p, v \rangle \in L_t$, otherwise the CAS would have failed. The state of the list directly after passing the decision point will be $\langle p, _ \rangle \notin L_t$. \square

Lemma 5 A DeleteMin operations which fails ($D() = \perp$), takes effect atomically at one statement.

Proof: The decision point and also the state-read point for an DeleteMin operations which fails ($D() = \perp$), is when the hidden read sub-operation of the ReadNext sub-operation in line D5 successfully reads the next pointer on lowest level that equals the tail node. The state of the list (L_t) directly before the passing of the state-read point must have been $L_t = \emptyset$. \square

Definition 6 We define the relation \Rightarrow as the total order and the relation \Rightarrow_d as the direct total order between all operations in the concurrent execution. In the following formulas, $E_1 \implies E_2$ means that if E_1 holds then E_2 holds as well, and \oplus stands for exclusive or (i.e. $a \oplus b$ means $(a \vee b) \wedge \neg(a \wedge b)$):

$$\begin{aligned} \text{Op}_1 \Rightarrow_d \text{Op}_2, \exists \text{Op}_3. \text{Op}_1 \Rightarrow_d \text{Op}_3, \\ \exists \text{Op}_4. \text{Op}_4 \Rightarrow_d \text{Op}_2 \implies \text{Op}_1 \Rightarrow_d \text{Op}_2 \end{aligned} \quad (5)$$

$$\text{Op}_1 \parallel \text{Op}_2 \implies \text{Op}_1 \Rightarrow_d \text{Op}_2 \oplus \text{Op}_2 \Rightarrow_d \text{Op}_1 \quad (6)$$

$$\text{Op}_1 \Rightarrow_d \text{Op}_2 \implies \text{Op}_1 \Rightarrow \text{Op}_2 \quad (7)$$

$$\text{Op}_1 \Rightarrow \text{Op}_2, \text{Op}_2 \Rightarrow \text{Op}_3 \implies \text{Op}_1 \Rightarrow \text{Op}_3 \quad (8)$$

Lemma 6 The operations that are directly totally ordered using formula 5, form an equivalent valid sequential execution.

Proof: If the operations are assigned their direct total order ($Op_1 \Rightarrow_d Op_2$) by formula 5 then also the decision, state-read and the state-change points of Op_1 is executed before the respective points of Op_2 . In this case the operations semantics behave the same as in the sequential case, and therefore all possible executions will then be equivalent to one of the possible sequential executions. \square

Lemma 7 *The operations that are directly totally ordered using formula 6 can be ordered unique and consistent, and form an equivalent valid sequential execution.*

Proof: Assume we order the overlapping operations according to their decision points. As the state before as well as after the decision points is identical to the corresponding state defined in the semantics of the respective sequential operations in formulas 1 to 4, we can view the operations as occurring at the decision point. As the decision points consist of atomic operations and are therefore ordered in time, no decision point can occur at the very same time as any other decision point, therefore giving a unique and consistent ordering of the overlapping operations. \square

Lemma 8 *With respect to the retries caused by synchronization, one operation will always do progress regardless of the actions by the other concurrent operations.*

Proof: We now examine the possible execution paths of our implementation. There are several potentially unbounded loops that can delay the termination of the operations. We call these loops retry-loops. If we omit the conditions that are because of the operations semantics (i.e. searching for the correct position etc.), the retry-loops take place when sub-operations detect that a shared variable has changed value. This is detected either by a subsequent read sub-operation or a failed CAS. These shared variables are only changed concurrently by other CAS sub-operations. According to the definition of CAS, for any number of concurrent CAS sub-operations, exactly one will succeed. This means that for any subsequent retry, there must be one CAS that succeeded. As this succeeding CAS will cause its retry loop to exit, and our implementation does not contain any cyclic dependencies between retry-loops that exit with CAS, this means that the corresponding *Insert* or *DeleteMin* operation will progress. Consequently, independent of any number of concurrent operations, one operation will always progress. \square

Theorem 1 *The algorithm implements a lock-free and linearizable priority queue.*

Proof: Following from Lemmas 6 and 7 and using the direct total order we can create an identical (with the same

semantics) sequential execution that preserves the partial order of the operations in a concurrent execution. Following from Definition 3, the implementation is therefore linearizable. As the semantics of the operations are basically the same as in the Skiplist [11], we could use the corresponding proof of termination. This together with Lemma 8 and that the state is only changed at one atomic statement (Lemmas 1,2,3,4,5), gives that our implementation is lock-free. \square

5 Experiments

In our experiments each concurrent thread performs 10000 sequential operations, whereof the first 100 or 1000 operations are *Insert* operations, and the remaining operations are randomly chosen with a distribution of 50% *Insert* operations versus 50% *DeleteMin* operations. The key values of the inserted nodes are randomly chosen between 0 and $1000000 * n$, where n is the number of threads. Each experiment is repeated 50 times, and an average execution time for each experiment is estimated. Exactly the same sequential operations are performed for all different implementations compared. Besides our implementation, we also performed the same experiment with two lock-based implementations. These are; 1) the implementation using multiple locks and Skiplists by Lotan *et al.* [9] which is the most recently claimed to be one of the most efficient concurrent priority queues existing, and 2) the heap-based implementation using multiple locks by Hunt *et al.* [7]. All lock-based implementations are based on simple spin-locks using the TAS atomic primitive. A clean-cache operation is performed just before each sub-experiment. All implementations are written in C and compiled with the highest optimization level, except from the atomic primitives, which are written in assembler.

The experiments were performed using different number of threads, varying from 1 to 30. To get a highly pre-emptive environment, we performed our experiments on a Compaq dual-processor Pentium II 450 MHz PC running Linux. A set of experiments was also performed on a Sun Solaris system with 4 processors. In order to evaluate our algorithm with full concurrency we also used a SGI Origin 2000 195 MHz system running Irix with 64 processors. The results from these experiments are shown in Figure 10 together with a close-up view of the Sun experiment. The average execution time is drawn as a function of the number of threads.

From the results we can conclude that all of the implementations scale similarly with respect to the average size of the queue. The implementation by Lotan and Shavit [9] scales linearly with respect to increasing number of threads when having full concurrency, although when exposed to pre-emption its performance decreases very rapidly; already with 4 threads the performance decreased with over 20

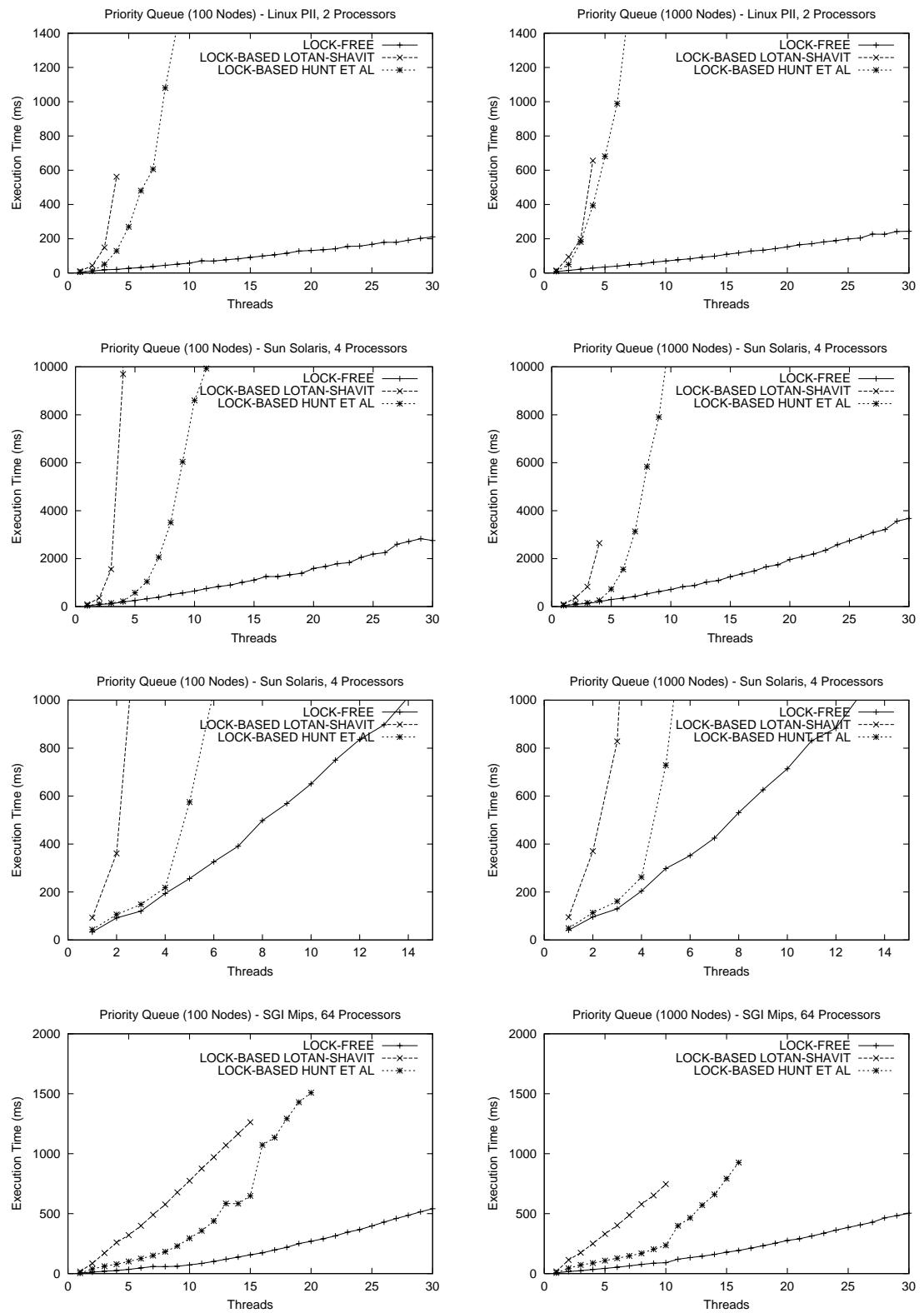


Figure 10. Experiment with priority queues and high contention, initialized with 100 respective 1000 nodes

times. We must point out here that the implementation by Lotan and Shavit is designed for a large (i.e. 256) number of processors. The implementation by Hunt *et al.* [7] shows better but similar behavior. Because of this behavior we decided to run the experiments for these two implementations only up to a certain number of threads to avoid getting time-outs. Our lock-free implementation scales best compared to all other involved implementations, having best performance already with 3 threads, independently if the system is fully concurrent or involves pre-emptions.

6 Extended Algorithm

When we have concurrent *Insert* and *DeleteMin* operations we might want to have certain real-time properties of the semantics of the *DeleteMin* operation, as expressed in [9]. The *DeleteMin* operation should only return items that have been inserted by an *Insert* operation that finished before the *DeleteMin* operation started. To ensure this we are adding timestamps to each node. When the node is fully inserted its timestamp is set to the current time. Whenever the *DeleteMin* operation is invoked it first checks the current time, and then discards all nodes that have a timestamp that is after this time. In the code of the implementation (see Figures 6,7,8 and 9), the additional statements that involve timestamps are marked within the “⟨” and “⟩” symbols. The function *getNextTimeStamp*, see Figure 14, creates a new timestamp. The function *compareTimeStamp*, see Figure 14, compares if the first timestamp is less, equal or higher than the second one and returns the values -1,0 or 1, respectively.

As we are only using the timestamps for relative comparisons, we do not need real absolute time, only that the timestamps are monotonically increasing. Therefore we can implement the time functionality with a shared counter, the synchronization of the counter is handled using CAS. However, the shared counter usually has a limited size (i.e. 32 bits) and will eventually overflow. Therefore the values of the timestamps have to be recycled. We will do this by exploiting information that are available in real-time systems, with a similar approach as in [15].

We assume that we have n periodic tasks in the system, indexed $\tau_1 \dots \tau_n$. For each task τ_i we will use the standard notations T_i , C_i , R_i and D_i to denote the period (i.e. min period for sporadic tasks), worst case execution time, worst case response time and deadline, respectively. The deadline of a task is less or equal to its period.

For a system to be safe, no task should miss its deadlines, i.e. $\forall i | R_i \leq D_i$.

For a system scheduled with fixed priority, the response time for a task in the initial system can be calculated using the standard response time analysis techniques [1]. If we with B_i denote the blocking time (the time the task can be

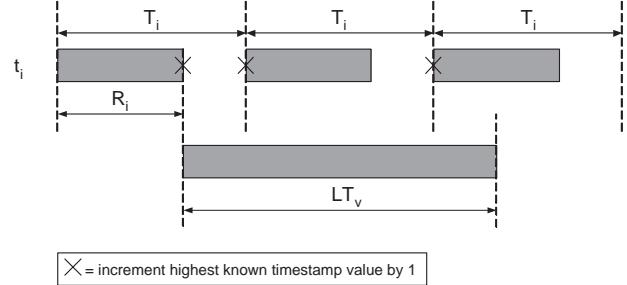


Figure 11. Maximum timestamp increase estimation - worst case scenario

delayed by lower priority tasks) and with $hp(i)$ denote the set of tasks with higher priority than task τ_i , the response time R_i for task τ_i can be formulated as:

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \quad (9)$$

The summand in the above formula gives the time that task τ_i may be delayed by higher priority tasks. For systems scheduled with dynamic priorities, there are other ways to calculate the response times [1].

Now we examine some properties of the timestamps that can exist in the system. Assume that all tasks call either the *Insert* or *DeleteMin* operation only once per iteration. As each call to *getNextTimeStamp* will introduce a new timestamp in the system, we can assume that every task invocation will introduce one new timestamp. This new timestamp has a value that is the previously highest known value plus one. We assume that the tasks always execute within their response times R with arbitrary many interruptions, and that the execution time C is comparably small. This means that the increment of highest timestamp respective the write to a node with the current timestamp can occur anytime within the interval for the response time. The maximum time for an *Insert* operation to finish is the same as the response time R_i for its task τ_i . The minimum time between two index increments is when the first increment is executed at the end of the first interval and the next increment is executed at the very beginning of the second interval, i.e. $T_i - R_i$. The minimum time between the subsequent increments will then be the period T_i . If we denote with LT_v the maximum life-time that the timestamp with value v exists in the system, the worst case scenario in respect of growth of timestamps is shown in Figure 11.

The formula for estimating the maximum difference in value between two existing timestamps in any execution becomes as follows:

$$MaxTag = \sum_{i=0}^n \left(\left\lceil \frac{\max_{v \in \{0.. \infty\}} LT_v}{T_i} \right\rceil + 1 \right) \quad (10)$$

Now we have to bound the value of $\max_{v \in \{0.. \infty\}} LT_v$. When comparing timestamps, the absolute value of these are not important, only the relative values. Our method is that we continuously traverse the nodes and replace outdated timestamps with a newer timestamp that has the same comparison result. We traverse and check the nodes at the rate of one step to the right for every invocation of an *Insert* or *DeleteMin* operation. With outdated timestamps we define timestamps that are older (i.e. lower) than any timestamp value that is in use by any running *DeleteMin* operation. We denote with *AncientVal* the maximum difference that we allow between the highest known timestamp value and the timestamp value of a node, before we call this timestamp outdated.

$$AncientVal = \sum_{i=0}^n \left\lceil \frac{\max_j R_j}{T_i} \right\rceil \quad (11)$$

If we denote with t_{ancient} the maximum time it takes for a timestamp value to be outdated counted from its first occurrence in the system, we get the following relation:

$$AncientVal = \sum_{i=0}^n \left\lfloor \frac{t_{\text{ancient}}}{T_i} \right\rfloor > \sum_{i=0}^n \left(\frac{t_{\text{ancient}}}{T_i} \right) - n \quad (12)$$

$$t_{\text{ancient}} < \frac{AncientVal + n}{\sum_{i=0}^n \frac{1}{T_i}} \quad (13)$$

Now we denote with t_{traverse} the maximum time it takes to traverse through the whole list from one position and getting back, assuming the list has the maximum size N .

$$N = \sum_{i=0}^n \left\lfloor \frac{t_{\text{traverse}}}{T_i} \right\rfloor > \sum_{i=0}^n \left(\frac{t_{\text{traverse}}}{T_i} \right) - n \quad (14)$$

$$t_{\text{traverse}} < \frac{N + n}{\sum_{i=0}^n \frac{1}{T_i}} \quad (15)$$

The worst-case scenario is that directly after the timestamp of one node gets traversed, it gets outdated. Therefore we get:

$$\max_{v \in \{0.. \infty\}} LT_v = t_{\text{ancient}} + t_{\text{traverse}} \quad (16)$$

Putting all together we get:

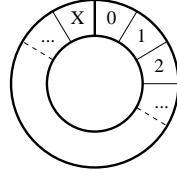


Figure 12. Timestamp value recycling

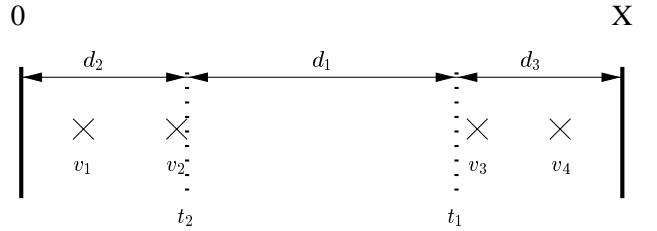


Figure 13. Deciding the relative order between reused timestamps

$$MaxTag < \sum_{i=0}^n \left(\left\lceil \frac{N + 2n + \sum_{k=0}^n \left\lceil \frac{\max_j R_j}{T_k} \right\rceil}{T_i \sum_{l=0}^n \frac{1}{T_l}} \right\rceil + 1 \right) \quad (17)$$

The above equation gives us a bound on the length of the "window" of active timestamps for any task in any possible execution. In the unbounded construction the tasks, by producing larger timestamps every time they slide this window on the $[0, \dots, \infty]$ axis, always to the right. The approach now is instead of sliding this window on the set $[0, \dots, \infty]$ from left to right, to cyclically slide it on a $[0, \dots, X]$ set of consecutive natural numbers, see figure 12. Now at the same time we have to give a way to the tasks to identify the order of the different timestamps because the order of the physical numbers is not enough since we are re-using timestamps. The idea is to use the bound that we have calculated for the span of different active timestamps. Let us then take a task that has observed v_i as the lowest timestamp at some invocation τ . When this task runs again as τ' , it can conclude that the active timestamps are going to be between v_i and $(v_i + MaxTag) \bmod X$. On the other hand we should make sure that in this interval $[v_i, \dots, (v_i + MaxTag) \bmod X]$ there are no old timestamps. By looking closer to equation 10 we can conclude that all the other tasks have written values to their registers with timestamps that are at most *MaxTag* less than v_i at the time that τ wrote the value v_i . Consequently if we use an interval that has double the

```

// Global variables
timeCurrent: integer
checked: pointer to Node
// Local variables
time,newtime,safeTime: integer
current,node,next: pointer to Node

function compareTimeStamp(time1:integer,
time2:integer):integer
C1 if time1=time2 then return 0;
C2 if time2=MAX_TIME then return -1;
C3 if time1>time2 and (time1-time2)≤MAX_TAG or
time1<time2 and (time1-time2+MAX_TIME)
≤MAX_TAG then return 1;
C4 else return -1;

function getNextTimeStamp():integer
G1 repeat
G2     time:=timeCurrent;
G3     if (time+1)≠MAX_TIME then newtime:=time+1;
G4     else newtime:=0;
G5     until CAS(&timeCurrent,time,newtime);
G6 return newtime;

procedure TraverseTimeStamps()
T1 safeTime:=timeCurrent;
T2 if safeTime≥ANCIENT_VAL then
T3     safeTime:=safeTime-ANCIENT_VAL;
T4 else safeTime:=safeTime+MAX_TIME-ANCIENT_VAL;
T5 while true do
T6     node:=READ_NODE(checked);
T7     current:=node;
T8     next:=ReadNext(&node,0);
T9     RELEASE_NODE(node);
T10    if compareTimeStamp(safeTime,next.timeInsert)>0 then
T11        next.timeInsert:=safeTime;
T12        if CAS(&checked,current,next) then
T13            RELEASE_NODE(current);
T14            break;
T15        RELEASE_NODE(next);

```

Figure 14. Creation, comparison, traversing and updating of bounded timestamps.

size of $MaxTag$, τ' can conclude that old timestamps are all on the interval $[(v_i - MaxTag) \bmod X, \dots, v_i]$.

Therefore we can use a timestamp field with double the size of the maximum possible value of the timestamp.

$$TagFieldSize = MaxTag * 2$$

$$TagFieldBits = \lceil \log_2 TagFieldSize \rceil$$

In this way τ' will be able to identify that v_1, v_2, v_3, v_4 (see figure 13) are all new values if $d_2 + d_3 < MaxTag$ and can also conclude that:

$$v_3 < v_4 < v_1 < v_2$$

The mechanism that will generate new timestamps in a cyclical order and also compare timestamps is presented in Figure 14 together with the code for traversing the nodes. Note that the extra properties of the priority queue that are achieved by using timestamps are not complete with respect to the *Insert* operations that finishes with an update. These update operations will behave the same as for the standard version of the implementation.

Besides from real-time systems, the presented technique can also be useful in non real-time systems as well. For example, consider a system of $n = 10$ threads, where the minimum time between two invocations would be $T = 10$ ns, and the maximum response time $R = 1000000000$ ns (i.e. after 1 s we would expect the thread to have crashed). Assuming a maximum size of the list $N = 10000$, we will have a maximum timestamp difference $MaxTag < 1000010030$, thus needing 31 bits. Given that most systems have 32-bit integers and that many modern systems handle 64 bits as well, it implies that this technique is practical for also non real-time systems.

7 Conclusions

We have presented a lock-free algorithmic implementation of a concurrent priority queue. The implementation is based on the sequential Skiplist data structure and builds on top of it to support concurrency and lock-freedom in an efficient and practical way. Compared to the previous attempts to use Skiplists for building concurrent priority queues our algorithm is lock-free and avoids the performance penalties that come with the use of locks. Compared to the previous lock-free/wait-free concurrent priority queue algorithms, our algorithm inherits and carefully retains the basic design characteristic that makes Skiplists practical: simplicity. Previous lock-free/wait-free algorithms did not perform well because of their complexity, furthermore they were often based on atomic primitives that are not available in today's systems.

We compared our algorithm with some of the most efficient implementations of priority queues known. Experiments show that our implementation scales well, and with 3 threads or more our implementation outperforms the corresponding lock-based implementations, for all cases on both fully concurrent systems as well as with pre-emption.

We believe that our implementation is of highly practical interest for multi-threaded applications. We are currently incorporating it into the NOBLE [16] library.

References

- [1] N.C. AUDSLEY, A. BURNS, R.I. DAVIS, K.W. TINDELL, A.J. WELLINGS. Fixed Priority Pre-emptive Scheduling: An Historical Perspective. *Real-Time Systems*, Vol. 8, Num. 2/3, pp. 129-154, 1995.
- [2] G. BARNES. Wait-Free Algorithms for Heaps. *Technical Report*, Computer Science and Engineering, University of Washington, Feb. 1992.
- [3] M. GRAMMATIKAKIS, S. LIESCHE. Priority queues and sorting for parallel simulation. *IEEE Transactions on Software Engineering*, SE-26 (5), pp. 401-422, 2000.
- [4] T. L. HARRIS. A Pragmatic Implementation of Non-Blocking Linked Lists. *Proceedings of the 15th International Symposium of Distributed Computing*, Oct. 2001.
- [5] M. HERLIHY, J. WING. Linearizability: a Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463-492, 1990.
- [6] M. HERLIHY. Wait-Free Synchronization. *ACM TOPLAS*, Vol. 11, No. 1, pp. 124-149, Jan. 1991.
- [7] G. HUNT, M. MICHAEL, S. PARTHASARATHY, M. SCOTT. An Efficient Algorithm for Concurrent Priority Queue Heaps. *Information Processing Letters*, 60(3), pp. 151-157, Nov. 1996.
- [8] A. ISRAELI, L. RAPPAPORT. Efficient wait-free implementation of a concurrent priority queue. *7th Intl Workshop on Distributed Algorithms '93*, Lecture Notes in Computer Science 725, Springer Verlag, pp 1-17, Sept. 1993.
- [9] I. LOTAN, N. SHAVIT. SkipList-Based Concurrent Priority Queues. *International Parallel and Distributed Processing Symposium*, 2000.
- [10] M. MICHAEL, M. SCOTT. Correction of a Memory Management Method for Lock-Free Data Structures.
- Computer Science Dept., University of Rochester*, 1995.
- [11] W. PUGH. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, vol.33, no.6, pp. 668-76, June 1990.
- [12] R. RAJKUMAR. Real-Time Synchronization Protocols for Shared Memory Multiprocessors. *10th International Conference on Distributed Computing Systems*, pp. 116-123, 1990.
- [13] L. SHA AND R. RAJKUMAR, J. LEHOCZKY. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers* Vol. 39, 9, pp. 1175-1185, Sep. 1990.
- [14] A. SILBERSCHATZ, P. GALVIN. Operating System Concepts. *Addison Wesley*, 1994.
- [15] H. SUNDELL, P. TSIGAS. Space Efficient Wait-Free Buffer Sharing in Multiprocessor Real-Time Systems Based on Timing Information. *Proceedings of the 7th International Conference on Real-Time Computing Systems and Applications (RTCSA 2000)*, pp. 433-440, IEEE press, 2000.
- [16] H. SUNDELL, P. TSIGAS. NOBLE: A Non-Blocking Inter-Process Communication Library. *Proceedings of the 6th Workshop on Languages, Compilers and Runtime Systems for Scalable Computers (LCR'02)*, Lecture Notes in Computer Science, Springer Verlag, 2002.
- [17] P. TSIGAS, Y. ZHANG. Evaluating the performance of non-blocking synchronization on shared-memory multiprocessors. *Proceedings of the international conference on Measurement and modeling of computer systems (SIGMETRICS 2001)*, pp. 320-321, ACM Press, 2001.
- [18] P. TSIGAS, Y. ZHANG. Integrating Non-blocking Synchronisation in Parallel Applications: Performance Advantages and Methodologies. *Proceedings of the 3rd ACM Workshop on Software and Performance (WOSP '02)*, ACM Press, 2002.
- [19] J. D. VALOIS. Lock-Free Data Structures. *PhD Thesis, Rensselaer Polytechnic Institute, Troy, New York*, 1995.

An Optimistic Approach to Lock-Free FIFO Queues

Edya Ladan-Mozes¹ and Nir Shavit²

¹ Department of Computer Science, Tel-Aviv University, Israel

² Sun Microsystems Laboratories and Tel-Aviv University

Abstract. First-in-first-out (FIFO) queues are among the most fundamental and highly studied concurrent data structures. The most effective and practical dynamic-memory concurrent queue implementation in the literature is the lock-free FIFO queue algorithm of Michael and Scott, included in the standard *JavaTM Concurrency Package*.

This paper presents a new dynamic-memory lock-free FIFO queue algorithm that performs consistently better than the Michael and Scott queue. The key idea behind our new algorithm is a novel way of replacing the singly-linked list of Michael and Scott, whose pointers are inserted using a costly compare-and-swap (CAS) operation, by an “optimistic” doubly-linked list whose pointers are updated using a simple store, yet can be “fixed” if a bad ordering of events causes them to be inconsistent. We believe it is the first example of such an “optimistic” approach being applied to a real world data structure.

1 Introduction

First-in-first-out (FIFO) queues are among the most fundamental and highly studied concurrent data structures [1–12], and are an essential building block of concurrent data structure libraries such as JSR-166, the JavaTM Concurrency Package [13]. A concurrent queue is a linearizable structure that supports enqueue and dequeue operations with the usual FIFO semantics. This paper focuses on queues with dynamic memory allocation.

The most effective and practical dynamic-memory concurrent FIFO queue implementation is the lock-free FIFO queue algorithm of Michael and Scott [14] (Henceforth the *MS-queue*). On shared-memory multiprocessors, this compare-and-swap (CAS) based algorithm is superior to all former dynamic-memory queue implementations including lock-based queues [14], and has been included as part of the JavaTM Concurrency Package [13]. Its key feature is that it allows uninterrupted parallel access to the head and tail of the queue.

This paper presents a new dynamic-memory lock-free FIFO queue algorithm that performs consistently better than the MS-queue. It is a practical example of an “optimistic” approach to reduction of synchronization overhead in concurrent data structures. At the core of this approach is the ability to use simple stores instead of CAS operations in common executions, and *fix* the data structure in the uncommon cases when bad executions cause structural inconsistencies.

1.1 The New Queue Algorithm

As with many finely tuned high performance algorithms (see for example CLH [15, 16] vs. MCS [17] locks), the key to our new algorithm’s improved performance is in saving a few costly operations along the algorithm’s main execution paths.

Figure 1 describes the MS-queue algorithm which is based on concurrent manipulation of a singly-linked list. Its main source of inefficiency is that while its `dequeue` operation requires a single successful CAS in order to complete, the `enqueue` operation requires *two* such successful CASs. This may not seem important, until one realizes that it increases the chances of failed CAS operations, and that on modern multiprocessors [18, 19], even the successful CAS operations cost an order-of-magnitude longer to complete than a load or a store, since they require exclusive ownership and flushing of the processor’s write buffers.

The key idea in our new algorithm is to (literally) approach things from a different direction... by logically reversing the direction of `enqueues` and `dequeues` to/from the list. If `enqueues` were to add elements at the beginning of the list, they would require only a single CAS, since one could first direct the new node’s `next` pointer to the node at the beginning of the list using only a store operation, and then CAS the `tail` pointer to the new node to complete the insertion. However, this re-direction would leave us with a problem at the end of the list: `dequeues` would not be able to traverse the list “backwards” to perform a linked-list removal.

Our solution, depicted in Figure 2, is to maintain a doubly-linked list, but to construct the “backwards” direction, the path of `prev` pointers needed by `dequeues`, in an optimistic fashion using only stores (and no memory barriers). This doubly-linked list may seem counter-intuitive given the extensive and complex work of maintaining the doubly-linked lists of lock-free deque algorithms using double-compare-and-swap operations [20]. However, we are able to store and follow the optimistic `prev` pointers in a highly efficient manner.

If a `prev` pointer is found to be inconsistent, we run a `fixList` method along the chain of `next` pointers which is guaranteed to be consistent. Since `prev` pointers become inconsistent as a result of long delays, not as a result of contention, the frequency of calls to `fixList` is low. The result is a FIFO queue

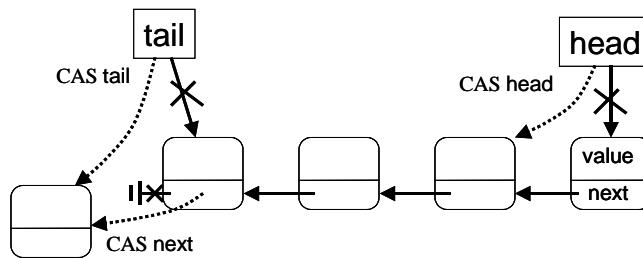


Fig. 1. The single CAS `dequeue` and costly two CAS `enqueue` of the MS-Queue algorithm

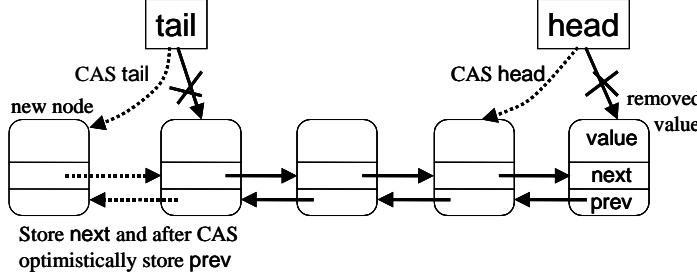


Fig. 2. The Single CAS enqueue and dequeue of the new algorithm

based on a doubly-linked list where pointers in both directions are set using simple stores, and both `enqueues` and `dequeues` require only a single successful CAS operation to complete.

1.2 Optimistic Synchronization

Optimistically replacing CAS with loads/stores was first suggested by Moir et al. [21] who show how one can replace the use of CAS with simple loads in good executions, using CAS only if a bad execution is incurred. However, while they show a general theoretical transformation, we show a practical example of a highly concurrent data structure whose actual performance is enhanced by using the optimistic approach.

Our optimistic approach joins several recent algorithms tailored to the good executions while dealing with the bad ones in a more costly fashion. Among these is the obstruction-freedom methodology of Herlihy et al. [22] and the lock-elision approach by Rajwar and Goodman [23] that use backoff and locking (respectively) to deal with bad cases resulting from contention. Our approach is different in that inconsistencies occur because of long delays, not as a result of contention. We use a special mechanism to fix these inconsistencies, and our resulting algorithm is lock-free.

1.3 Performance

We compared our new lock-free queue algorithm to the most efficient lock-based and lock-free dynamic memory queue implementations in the literature, the two-lock-queue and lock-free MS-queue of Michael and Scott [14]. Our empirical results, presented in Section 4, show that the new algorithm performs consistently better than the MS-queue. This improved performance is not surprising, as our `enqueues` require fewer costly CAS operations, and as our benchmarks show, generate significantly less failed CAS operations.

The new algorithm can use the same dynamic memory pool structure as the MS-queue. It fits with memory recycling methods such as ROP [24] or SMR

[25], and it can be written in the JavaTM programming language without the need for a memory pool or ABA-tags. We thus believe it can serve as a viable practical alternative to the MS-queue.

2 The Algorithm in Detail

The efficiency of our new algorithm rests on implementing a queue using a doubly-linked list, which, as we show, allows `enqueues` and `dequeues` to be performed with a single CAS per operation. Our algorithm guarantees that this list is always connected and ordered by the `enqueue` order in one direction. The other direction is optimistic and may be inaccurate at various points of the execution, but can be reconstructed to an accurate state when needed.

Our shared queue data structure (see Figure 3) consists of a `head` pointer, a `tail` pointer, and nodes. Each node added to the queue contains a `value`, a `next` pointer and a `prev` pointer. Initially, a node with a predefined dummy value, hence forth called a `dummy` node, is created and both `head` and `tail` point to it. During the execution, the `tail` always points to the last (youngest) node inserted to the queue, and the `head` points to the first (oldest) node. When the queue becomes empty, both `head` and `tail` point to a `dummy` node. Since our algorithm uses CAS for synchronization, the ABA issue arises [14, 10]. In Section 2.1, we describe the `enqueue` and `dequeue` operations ignoring ABA issues. The tagging mechanism we added to overcome the ABA problem is explained in Section 3. The code in this section includes this tagging mechanism. Initially, the tags of the `tail` and `head` are zero. When a new node is created, the tags of the `next` and `prev` pointers are initiated to a predefined null value.

```

struct pointer_t {
    <ptr, tag>: <node_t *, unsigned integer>
};

struct node_t {
    data_type value;
    pointer_t next;
    pointer_t prev;
};
struct queue_t {
    pointer_t tail;
    pointer_t head;
};

```

Fig. 3. The queue data structures

2.1 The Queue Operations

A FIFO queue supports two operations (or methods): `enqueue` and `dequeue`. The `enqueue` operation inserts a value to the queue and the `dequeue` operation deletes the oldest value in the queue.

The code of the `enqueue` method appears in Figure 4, and the code of the `dequeue` method appears in Figure 5. To insert a value, the `enqueue` method creates a new node that contains the value, and then tries to insert this node to the queue. As seen in Figure 2, the `enqueue` reads the current `tail` of the queue, and sets the new node's `next` pointer to point to that same node. Then it tries to atomically modify the `tail` to point to its new node using a CAS operation. If the CAS succeeded, the new node was inserted into the queue. Otherwise the `enqueue` retries.

```

void enqueue(queue_t* q, data_type val)
E01: pointer_t tail
E02: node_t* nd = new_node()                                # Allocate a new node
E03: nd->value = val                                       # Set enqueued value
E04: while(TRUE){                                         # Do till success
E05:   tail = q->tail                                     # Read the tail
E06:   nd->next = <tail.ptr, tail.tag+1>                  # Set node's next ptr
E07:   if CAS(&(q->tail), tail, <nd, tail.tag+1>){ # Try to CAS the tail
E08:     (tail.ptr)->prev = <nd, tail.tag>                # Success, write prev
E09:     break                                                 # Enqueue done!
E10: }
E11: }
```

Fig. 4. The `enqueue` operation

To delete a node, a `dequeue` method reads the current `head` and `tail` of the queue, and the `prev` pointer of the node pointed by the `head`. It then tries to CAS the `head` to point to the node as that pointed by the `prev` pointer. If it succeeded, then the node previously pointed by the `head` was deleted. If it failed, it repeats the above steps. If the queue is empty then `NULL` is returned.

We now explain how we update the `prev` pointers of the nodes in a consistent and lock-free manner, assuming there is no ABA problem. The `prev` pointers are modified in two stages. The first stage is performed optimistically immediately after the successful insertion of a new node. An `enqueue` method that succeeded in atomically modifying the `tail` using a CAS, updates the `prev` pointer of the node previously pointed by the `tail` to point to the new node. This is done using a simple store operation. Once this write is completed, the `prev` pointer points to its preceding node in the list. Thus the order of operations to perform an `enqueue` is a write of the `next` in the new node, then a CAS of the `tail`, and finally a write of the `prev` pointer of the node pointed to by the `next` pointer. This ordering will prove crucial in guaranteeing the correctness of our algorithm.

Unfortunately, the storing of the `prev` pointer by an `enqueue` might be delayed for various reasons, and a dequeuing method might not see the necessary `prev` pointer. The second stage is intended to fix this situation. In order to fix the `prev` pointer, we use the fact that the `next` pointer of each node is set only by the `enqueue` method that inserted that node, and never changes until the node

```

data_type dequeue(queue_t* q)
D01: pointer_t tail, head, firstNodePrev
D02: node_t* nd_dummy
D03: data_type val
D04: while(TRUE){                                # Try till success or empty
D05:   head = q->head                          # Read the head
D06:   tail = q->tail                           # Read the tail
D07:   firstNodePrev = (head.ptr)->prev        # Read first node prev
D08:   val = (head.ptr)->value                  # Read first node val
D09:   if (head == q->head){                   # Check consistency
D10:     if (val != dummy_val){                # Head val is dummy?
D11:       if (tail != head){                 # More than 1 node?
D12:         if (firstNodePrev.tag != head.tag){# Tags not equal?
D13:           fixList(q, tail, head)          # Call fixList
D14:           continue                    # Re-iterate (D04)
D15:         }
D16:       }
D17:     else{                               # Last node in queue
D18:       nd_dummy = new_node()            # Create a new node
D19:       nd_dummy->value = dummy_val    # Set it's val to dummy
D20:       nd_dummy->next = <tail.ptr, tail.tag+1> # Set its next ptr
D21:       if CAS(&(q->tail), tail ,<nd_dummy, tail.tag+1>){# CAS tail
D22:         (head.ptr).prev = <nd_dummy, tail.tag>      # Write prev
D23:       }
D24:     else{                           # CAS failed
D25:       free(nd_dummy)               # free nd_dummy
D26:     }
D27:     continue;                      # Re-iterate (D04)
D28:   }
D29:   if CAS(&(q->head), head, <firstNodePrev.ptr,head.tag+1>){# CAS
D30:     free (head.ptr)                  # Free the dequeued node
D31:     return val                      # Dequeue done!
D32:   }
D33: }
D34: else {                                     # Head points to dummy
D35:   if (tail.ptr == head.ptr){              # Tail points to dummy?
D36:     return NULL;                         # Empty queue, done!
D37:   }
D38: else{                                      # Need to skip dummy
D39:   if (firstNodePrev.tag != head.tag){# Tags not equal?
D40:     fixList(q, tail, head);          # Call fixList
D41:     continue;                        # Re-iterate (D04)
D42:   }
D43:   CAS(&(q->head),head,<firstNodePrev.ptr,head.tag+1>)#Skip dummy
D44: }
D45: }
D46: }
D47: }

```

Fig. 5. The dequeue operation

is dequeued. Thus, if ABA problems resulting from node recycling are ignored, this order is invariant. The fixing mechanism walks through the entire list from the **tail** to the **head** along the chain of **next** pointers, and corrects the **prev** pointers accordingly. Figure 7 provides the code of the **fixList** procedure. As can be seen, the fixing mechanism requires only simple load and store operations.

There are two special cases we need to take care of: when the last node is being deleted and when the the **dummy** node needs to be skipped.

- The situation in which there is only one node in the queue is encountered when the **tail** and **head** point to the same node, which is not a **dummy** node. Deleting this node requires three steps and two CAS operations, as seen in Figure 6 Part A. First, a new node with a **dummy** value is created, and its next pointer is set to point to the last node. Second, the **tail** is atomically modified using a CAS to point to this **dummy** node, and then, the **head** is atomically modified using a CAS to also point to this **dummy** node. The intermediate state in which the **tail** points to a **dummy** node and the **head** points to another node is special, and occurs only in the above situation. This sequence of operations ensures that the algorithm is not blocked even if a **dequeue** method modified the **tail** to point to a **dummy** node and then stopped. We can detect the situation in which the **tail** points to a **dummy** node and the **head** does not, and continue the execution of the **dequeue** method. In addition, enqueueing methods can continue to insert new nodes to the queue, even in the intermediate state.
- In our algorithm, a **dummy** node is a special node with a **dummy** value. It is created and inserted to the queue when it becomes empty as explained above. Since a **dummy** node does not contain a real value, it must be skipped when nodes are deleted from the queue. The steps for skipping a **dummy** node are similar to those of a regular dequeue, except that no value is returned. When a **dequeue** method identifies that the **head** points to a **dummy** node and the **tail** does not, as in Figure 6 Part B, it modifies the **head** using a CAS to point to the node pointed by the **prev** pointer of this **dummy** node. Then it can continue to **dequeue** nodes.

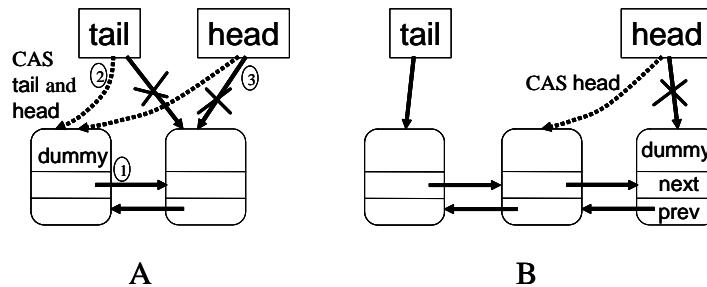


Fig. 6. A - A **dequeue** of the last node, B - Skipping the dummy node

```

F01: void fixList(queue_t* q, pointer_t tail, pointer_t head)
F02: pointer_t curNode , curNodeNext, nextNodePrev
F03: curNode = tail                                # Set curNode to tail
F04: while((head == q->head) && (curNode != head)){ # While not at head
F05:   curNodeNext = (curNode.ptr)->next           # Read curNode next
F06:   if (curNodeNext.tag != curNode.tag){          # Tags don't equal?
F07:     return;                                    # ABA, return!
F08:   }
F09:   nextNodePrev = (curNodeNext.ptr)->prev      # Read next node prev
F10:  if (nextNodePrev != <curNode.ptr, curNode.tag-1>){#Ptr don't equal?
F11:    (curNodeNext.ptr)->prev = <curNode.ptr, curNode.tag-1>; # Fix
F12:  }
F13:  curNode = <curNodeNext.ptr, curNode.tag-1>    # Advance curNode
F14: }

```

Fig. 7. The fixList procedure

3 Solving the ABA Problem

An ABA situation [10, 14] can occur when a process read some part of the shared memory in a given state and then was suspended for a while. When it wakes up the part it read could be in an identical state, however many insertions and deletions could have happened in the interim. The process may incorrectly succeed in performing a CAS operation, bringing the data structure to an inconsistent state. To identify such situation and eliminate ABA, we use the standard tagging-mechanism approach [26, 14].

In the tagging-mechanism, each pointer (`tail`, `head`, `next`, and `prev`) is added a `tag`. The `tags` of the `tail` and `head` are initiated to zero. When a new node is created, the `next` and `prev` tags are initiated to a predefined null value. The `tag` of each pointer is atomically modified with the pointer itself when a CAS operation is performed on the pointer.

Each time the `tail` or `head` is modified, its `tag` is incremented, also in the special cases of deleting the last node and skipping the dummy node. If the `head` and `tail` point to the same node, their `tags` must be equal. Assume that an `enqueue` method executed by a process P read that the `tail` points to node A and then was suspended. By the time it woke up, A was deleted, B was inserted and A was inserted again. The `tag` attached to the `tail` pointing to A will now be different (incremented twice) from the `tag` originally read by P . Hence P 's `enqueue` will fail when attempting to CAS the `tail`.

The ABA problem can also occur while modifying the `prev` pointers. The `tag` of the `next` pointer is set by the enqueueing process to equal the `tag` of the new `tail` it tries to CAS. The tag of the `prev` pointer is set to equal the `tag` of the `next` pointer in the same node. Thus consecutive nodes in the queue have consecutive `tags` in the `next` and `prev` pointers. Assume that an `enqueue` method executed by process P inserted a node to the queue, and stopped before

it modified the `prev` pointer of the consecutive node A (see Section 2.1). Then A was deleted and inserted again. When P woke up, it wrote its pointer and the `tag` to the `prev` pointer of A. Though the pointer is incorrect, the tag indicates this since it is smaller than the one expected. A `dequeue` method verifies that the `tag` of the `prev` pointer of the node it is deleting equals the `tag` of the `head` pointer it read. If the `tags` are different, it concludes that an ABA problem occurred, and calls a method to fix the `prev` pointer.

The fixing of the `prev` pointer after it was corrupted by an ABA situation is performed in the `fixList` procedure (Figure 7), in combination with the second stage of modifying the `prev` pointers, as explained in Section 2.1. In addition to using the fact that the `next` pointers are set locally by the `enqueue` method and never change, we use the fact that consecutive nodes must have consecutive `tags` attached to the `next` and `prev` pointers. The fixing mechanism walks through the entire list from the `tail` to the `head` along the `next` pointers of the nodes, correcting `prev` pointers if their `tags` are not consistent.

Finally we note that in garbage-collected languages such as the JavaTM programming language, ABA does not occur and the `tags` are not needed. When creating a new instance of a node, its `prev` pointer is set to NULL. Based on this, the fixing mechanism is invoked if the `prev` pointer points to NULL (instead of checking that the tags are equal). In this way we can detect the case in which an `enqueue` did not succeed in its optimistic update of the `prev` pointer of the consecutive node, and fix the list according to the `next` pointers.

4 Performance

We evaluated the performance of our FIFO queue algorithm relative to other known methods by running a collection of synthetic benchmarks on a 16 processor Sun EnterpriseTM 6500, an SMP machine formed from 8 boards of two 400MHz UltraSparc® processors, connected by a crossbar UPA switch, and running a SolarisTM 9 operating system. Our C code was compiled by a Sun `cc` compiler 5.3, with flags `-xO5 -xarch=v8plusa`.

4.1 The Benchmarks

We compare our algorithm to the two-lock queue and to MS-queue of Michael and Scott [14]. We believe these algorithms to be the most efficient known lock-based and lock-free dynamic-memory queue algorithms in the literature. We used Michael and Scott's code (referenced in [14]).

The original Michael and Scott paper [14] showed only an *enqueue-dequeue pairs* benchmark where a process repeatedly alternated between enqueueing and dequeuing. This tests a rather limited type of behavior. In order to simulate additional patterns, we implemented an internal memory management mechanism. As in Micheal and Scott's benchmark, we use an array of nodes that are allocated in advance. Each process has its own pool with an equal share of these nodes. Each process performs a series of `enqueues` on its pool of nodes and `dequeues`

from the queue. A dequeued node is placed in dequeuing process pool for reuse. If there are no more nodes in its local pool, a process must first `dequeue` at least one node, and can then continue to `enqueue`. Similarly, a process cannot `dequeue` nodes if its pool is full. To guarantee fairness, we used the same mechanism for all the algorithms. We tested several benchmarks of which two are presented here:

- enqueue-dequeue pairs: each process alternately performed `enqueue` or `dequeue` operation.
- 50% enqueues: each process chooses uniformly at random whether to perform an enqueue or a dequeue, creating a random pattern of 50% `enqueue` and 50% `dequeue` operations.

4.2 The Experiments

We repeated the above benchmarks delaying each process a random amount of time between operations to mimic local work usually performed by processes (in the range of 0 to 1000 increment operations in a loop).

We measured latency (in milliseconds) as a function of the number of processes: the amount of time that elapsed until the completion of a total of a million operations divided equally among processes. To counteract transient startup effects, we synchronized the start of the processes (i.e., no process can start before all others finished their initialization phase).

We pre-tested the algorithms on the given benchmarks by running hundreds of combinations of exponential backoff delays. The results we present were taken from the best combination of backoff values for each algorithm in each benchmark (although we found, similarly to Michael and Scott, that the exact choice of backoff did not cause a significant change in performance). Each of the presented data points in our graphs is the average of eight runs.

4.3 Empirical Results

As can be seen in Figure 8, the new algorithm consistently outperforms the MS-queue in both the 50% and the enqueue-dequeue pairs benchmarks when there are more than two processes. From the enqueue-dequeue pairs benchmark one can also see that the lock-based algorithm is consistently worst than the lock-free algorithms, and deteriorates when there is multiprogramming, that is, when there are 32 processes on 16 processors. Hence, in the rest of this section, we concentrate on the performance of the MS-queue and our new algorithm.

Figure 8 shows that the results in both enqueue-dequeue pairs and 50% enqueues benchmarks were very similar, except in the case of one or two processes. To explain this, let us consider the overhead of an empty queue, the number of calls to the `fixList` procedure as it appears in the left side of Figure 9, and the number failed CAS operations as it appears in the right side of Figure 9.

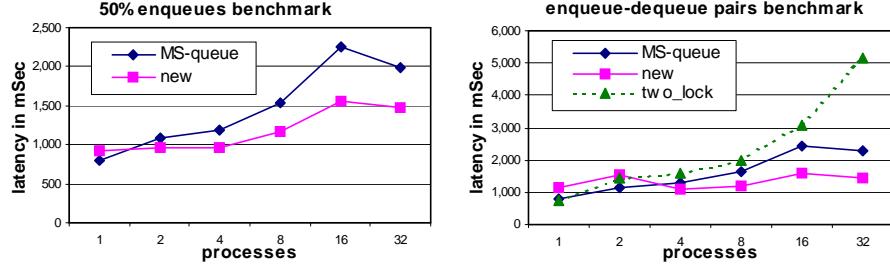


Fig. 8. Results of enqueue-dequeue pairs and 50% benchmarks

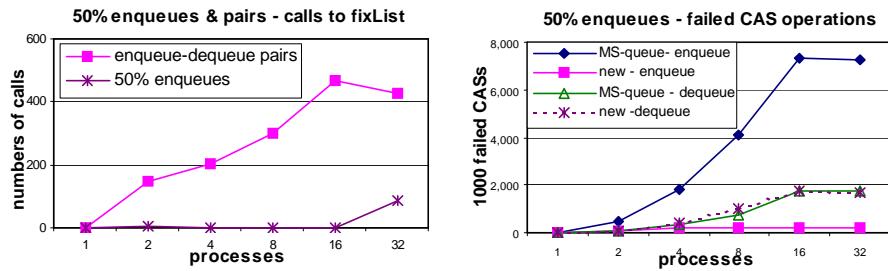


Fig. 9. The number of failed CASs and calls to fixList

- As described in Section 2.1 (see also Figure 6), additional successful CASs are required by the new algorithm when the queue becomes empty. As the number of concurrent processes increases, their scheduling causes the queue to become empty less frequently, thus incurring less of the overhead of an empty queue. A benchmark which we do not present here shows that this phenomena can be eliminated if the enqueue-dequeue pairs benchmark is initiated with a non-empty queue. In the 50% enqueues benchmark, due to its random characteristics, the overhead of an empty queue is eliminated even in low concurrency levels.
- Overall, there were a negligible number of calls to `fixlist` in both benchmarks, no more than 450 calls for a million operations. This makes a strong argument in favor of the optimistic approach.

Recall that the `fixList` procedure is called when a process tries to `dequeue` a node before the `enqueueing` process completed the optimistic update of the `prev` pointer of the consecutive node. This happens more frequently in the enqueue-dequeue pairs benchmark due to its alternating nature. In the 50% enqueues benchmark, due to its more random patterns, there are almost no calls to `fixList` when the concurrency level is low, and about 85 when there are 32 processes.

- The righthanded side of Figure 9 shows the number of failed CAS operations in the `enqueue` and `dequeue` methods. These numbers expose one of the key performance benefits of the new algorithm. Though the number of failed CASs in the `dequeue` operations in both algorithms is approximately the same, the number of failed CASes in the `enqueue` of MS-queue is about 20 to 40 times greater than in our new algorithm. This is a result of the additional CAS operation required by MS-queue’s `enqueue` method, and is the main advantage allowed by our new optimistic doubly-linked list structure.

We conclude that in our tested benchmarks, our new algorithm outperforms the MS-queue. The MS-queue’s latency is increased by the failed CASs in the `enqueue` operation, while the latency of our new algorithm is influenced by the additional CASs when the queue is empty. We note again that in our presented benchmarks we did not add initial nodes to soften the effect of encountering an empty queue.

5 Correctness Proof

This section contains a sketch of the formal proof that our algorithm has the desired properties of a concurrent FIFO queue. A sequential FIFO queue as defined in [27] is a data structure that supports two operations: `enqueue` and `dequeue`. The `enqueue` operation takes a value as an argument, inserts it to the queue, and does not return a value. The `dequeue` operation does not take an argument, deletes and returns the oldest value from the queue.

We prove that our concurrent queue is linearizable to the sequential FIFO queue, and that it is lock-free. We treat basic read/write (load/store) and CAS operations as atomic actions, and can thus take the standard approach of viewing them as if they occurred one after the other in sequence [28].

In the following we explain the FIFO queue semantics and define the linearization points for each `enqueue` and `dequeue` operation. We then define the insertion order of elements to the queue. The correctness proof and the lock freedom property proof are only briefly described out of space limitations.

5.1 Correct FIFO Queue Semantics

The queue in our implementation is represented by a `head` and a `tail` pointers, and uses a `dummy` node. Each node in the queue contains a value, a `next` pointer and a `prev` pointer. All pointers, `head`, `tail`, `next` and `prev`, are attached with a `tag`. Initially, all tags are zero and the `head` and `tail` point to the `dummy` node.

The Compare-And-Swap (CAS) operation used in our algorithm takes a register, an *old* value, and a *new* value. If the register’s current content equals *old*, then it is replaced by *new*, otherwise the register remains unchanged [29]. A successful CAS operation is an operation that modified the register.

The successfulness of the `enqueue` and `dequeue` operations depends on the successfulness of CAS operations performed in the execution. For any process,

the `enqueue` operation is always successful. The operation ends when a process successfully performed the CAS operation in line E07. A successful `dequeue` operation is one that successfully performed the CAS in D22. If the queue is empty, the `dequeue` operation is considered unsuccessful and it returns null.

Definition 1. *The linearization points of `enqueue` and `dequeue` operations are:*

- *Enqueue operations are linearized at the successful CAS in line E07.*
- *Successful dequeue operations are linearized at the successful CAS in line D29.*
- *Unsuccessful dequeue operations are linearized in line D06.*

Definition 2. *In any state of the queue, the insertion order of nodes to the queue is the reverse order of the nodes starting from the tail, linked by the `next` pointers, to the head.*

If the `dummy` node is linked before the `head` is reached, then the insertion order is the same from the `tail` to the `dummy` node, the `dummy` node is excluded, and the node pointed by the `head` is attached instead of the `dummy` node. If the `head` points to the `dummy` node then the `dummy` node is excluded.

5.2 The Proof Structure

In the full paper we show that the insertion order is consistent with the linearization order on the `enqueue` operations. We do that by showing that the `next` pointer of a linearized `enqueue` operation always points to the node inserted by the previous linearized `enqueue` operation, and that the `next` pointers never change during the execution. We then show that the correctness of the `prev` pointers can be verified using the `tags`, and fixed if needed by the `fixList` procedure. We also prove that in any state of the queue there is at most one node with a `dummy` value in the queue, and that the queue is empty if both `head` and `tail` point to a `dummy` node.

To finish the proof we show that the deletion order of nodes from the queue is consistent with the insertion order. This is done by proving that we can detect the case in which the optimistic update of the `prev` pointer did not occur (and also the case of an ABA situation) and fix it using the `tags` and the `fixList` procedure. We then show that when a `dequeue` operation takes place, the `prev` pointer of the node pointed by the `head`, always point to the consecutive node as dictated by the `next` pointers.

From the above we can conclude that our concurrent implementation implements a FIFO queue.

5.3 Lock Freedom

In order to prove that our algorithm is lock-free we need to show that if one process fails in executing an `enqueue` or `dequeue` operation, then another process have modified the `tail` or the `head`, and thus the system as whole made progress. We also need to show that the `fixList` procedure eventually ends. These properties are fairly easy to conclude from the code.

5.4 Complexity

It can be seen from the code that each `enqueue` and `dequeue` operation takes a constant number of steps in the uncontended case. The `fixList` procedure, in a specific state of the queue, requires all the running `dequeue` processes to go over all the nodes in the queue in order to fix the list. However, once this part of the queue is fixed, when ABA does not occur, all the nodes in this part can be dequeued without the need to fix the list again.

6 Conclusion

In this paper we presented a new dynamic-memory lock-free FIFO queue. Our queue is based on an optimistic assumption of good ordering of operations in the common case, and on the ability to fix the data structure if needed. It requires only one CAS operation for each enqueue and dequeue operation and performs constantly better than the MS-queue. We believe that our new algorithm can serve as a viable alternative to the MS-queue for implementing linearizable FIFO queues.

References

1. Gottlieb, A., Lubachevsky, B.D., Rudolph, L.: Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Trans. Program. Lang. Syst.* **5** (1983) 164–189
2. Herlihy, M., Wing, J.: Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems* **12** (1990) 463–492
3. Hwang, K., Briggs, F.A.: Computer Architecture and Parallel Processing. McGraw-Hill, Inc. (1990)
4. Lamport, L.: Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems* **5** (1983) 190–222
5. Mellor-Crummey, J.M.: Concurrent queues: Practical fetch-and- ϕ algorithms. Technical Report Technical Report 229, University of Rochester (1987)
6. Prakash, S., Lee, Y.H., Johnson, T.: Non-blocking algorithms for concurrent data structures. Technical Report 91–002, Department of Information Sciences, University of Florida (1991)
7. Prakash, S., Lee, Y.H., Johnson, T.: A non-blocking algorithm for shared queues using compare-and-swap. *IEEE Transactions on Computers* **43** (1994) 548–559
8. Stone, H.S.: High-performance computer architecture. Addison-Wesley Longman Publishing Co., Inc. (1987)
9. Stone, J.: A simple and correct shared-queue algorithm using compare-and-swap. In: Proceedings of the 1990 conference on Supercomputing, IEEE Computer Society Press (1990) 495–504
10. Treiber, R.K.: Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center (1986)
11. Tsigas, P., Zhang, Y.: A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems. In: Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures, ACM Press (2001) 134–143

12. Valois, J.: Implementing lock-free queues. In: Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems. (1994) 64–69
13. Lea, D.: (The java concurrency package (JSR-166))
<http://gee.cs.oswego.edu/d1/concurrency-interest/index.html>.
14. Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC '96), New York, USA, ACM (1996) 267–275
15. Craig, T.: Building FIFO and priority-queueing spin locks from atomic swap. Technical Report TR 93-02-02, University of Washington, Department of Computer Science (1993)
16. Magnussen, P., Landin, A., Hagersten, E.: Queue locks on cache coherent multiprocessors. In: Proceedings of the 8th International Symposium on Parallel Processing (IPPS), IEEE Computer Society (1994) 165–171
17. Mellor-Crummey, J., Scott, M.: Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Transactions on Computer Systems **9** (1991) 21–65
18. Weaver, D., (Editors), T.G.: The SPARC Architecture Manual (Version 9). PTR Prentice Hall, Englewood Cliffs, NJ) (1994)
19. Intel: Pentium Processor Family User's Manual: Vol 3, Architecture and Programming Manual. (1994)
20. Agesen, O., Detlefs, D., Flood, C., Garthwaite, A., Martin, P., Moir, M., Shavit, N., Steele, G.: DCAS-based concurrent deques. Theory of Computing Systems **35** (2002) 349–386
21. Luchangco, V., Moir, M., Shavit, N.: On the uncontended complexity of consensus. In: Proceedings of Distributed Computing. (2003)
22. Herlihy, M., Luchangco, V., Moir, M.: Obstruction-free synchronization: Double-ended queues as an example. In: Proceedings of the 23rd International Conference on Distributed Computing Systems, IEEE (2003) 522–529
23. Rajwar, R., Goodman, J.: Speculative lock elision: Enabling highly concurrent multithreaded execution. In: Proceedings of the 34th Annual International Symposium on Microarchitecture. (2001) 294–305
24. Herlihy, M., Luchangco, V., Moir, M.: The repeat offender problem: A mechanism for supporting lock-free dynamic-sized data structures. In: Proceedings of the 16th International Symposium on DIStributed Computing. Volume 2508., Springer-Verlag Heidelberg (2002) 339–353 A improved version of this paper is in preparation for journal submission; please contact authors.
25. Michael, M.: Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In: The 21st Annual ACM Symposium on Principles of Distributed Computing, ACM Press (2002) 21–30
26. Moir, M.: Practical implementations of non-blocking synchronization primitives. In: Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing. (1997) 219–228
27. Cormen, T., Leiserson, C., Rivest, R., Stein, C.: Introduction to Algorithms. Second edition edn. MIT Press, Cambridge, MA (2001)
28. Afek, Y., Attiya, H., Dolev, D., Gafni, E., Merritt, M., Shavit, N.: Atomic snapshots of shared memory. In Dwork, C., ed.: Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing, Québec City, Québec, Canada, ACM Press (1990) 1–14
29. Herlihy, M.: Wait-free synchronization. ACM Transactions on Programming Languages and Systems (TOPLAS) **13** (1991) 124–149

Efficient Almost Wait-free Parallel Accessible Dynamic Hashtables

Gao, H.¹, Groote, J.F.², Hesselink, W.H.¹

¹ Department of Mathematics and Computing Science, University of Groningen, P.O. Box 800, 9700 AV Groningen, The Netherlands (Email: {hui,wim}@cs.rug.nl)

² Department of Mathematics and Computing Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands and CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands (Email: jfg@win.tue.nl)

Abstract

In multiprogrammed systems, synchronization often turns out to be a performance bottleneck and the source of poor fault-tolerance. Wait-free and lock-free algorithms can do without locking mechanisms, and therefore do not suffer from these problems. We present an efficient almost wait-free algorithm for parallel accessible hashtables, which promises more robust performance and reliability than conventional lock-based implementations. Our solution is as efficient as sequential hashtables. It can easily be implemented using C-like languages and requires on average only constant time for insertion, deletion or accessing of elements. Apart from that, our new algorithm allows the hashtables to grow and shrink dynamically when needed.

A true problem of lock-free algorithms is that they are hard to design correctly, even when apparently straightforward. Ensuring the correctness of the design at the earliest possible stage is a major challenge in any responsible system development. Our algorithm contains 81 atomic statements. In view of the complexity of the algorithm and its correctness properties, we turned to the interactive theorem prover PVS for mechanical support. We employ standard deductive verification techniques to prove around 200 invariance properties of our almost wait-free algorithm, and describe how this is achieved using the theorem prover PVS.

CR Subject Classification (1991): D.1 Programming techniques

AMS Subject Classification (1991): 68Q22 Distributed algorithms, 68P20 Information storage and retrieval

Keywords & Phrases: Hashtables, Distributed algorithms, Lock-free, Wait-free

1 Introduction

We are interested in efficient, reliable, parallel algorithms. The classical synchronization paradigms are not most suited for this, because synchronization often turns out a performance bottleneck, and failure of a single process can force all other processes to come to a halt. Therefore, wait-free, lock-free, or synchronization-free algorithms are of interest [11, 19, 13].

An algorithm is *wait-free* when each process can accomplish its task in a finite number of steps, independently of the activity and speed of other processes. An algorithm is *lock-free* when it guarantees that within a finite number of steps always some process will complete its tasks, even if other processes halt. An algorithm is *synchronization-free* when it does not contain synchronization primitives. The difference between wait-free and lock-free is that a lock-free process can be arbitrarily delayed by other processes that repeatedly start and accomplish tasks. The difference between synchronization-free and lock-free is that in a synchronization-free algorithm processes may delay each other arbitrarily, without getting closer to accomplishing their respective tasks. As we present a lock-free algorithm, we only speak about lock-freedom below, but most applies to wait-freedom or synchronization-freedom as well.

Since the processes in a lock-free algorithm run rather independently of each other, lock-free algorithms scale up well when there are more processes. Processors can finish their tasks on their own, without being blocked, and generally even without being delayed by other processes. So, there is no need to wait for slow or overloaded processors. In fact, when there are processors of

differing speeds, or under different loads, a lock-free algorithm will generally distribute common tasks over all processors, such that it is finished as quickly as possible.

As argued in [13], another strong argument for lock-free algorithms is reliability. A lock-free algorithm will carry out its task even when all but one processor stops working. Without problem it can stand any pattern of processors being switched off and on again. The only noticeable effect of failing processors is that common tasks will be carried out somewhat slower, and the failing processor may have claimed resources, such as memory, that it can not relinquish anymore.

For many algorithms the penalty to be paid is minor; setting some extra control variables, or using a few extra pointer indirections suffices. Sometimes, however, the time and space complexities of a lock-free algorithm is substantially higher than its sequential, or ‘synchronized’ counterpart [7]. Furthermore, some machine architectures are not very capable of handling shared variables, and do not offer *compare-and-swap* or *test-and-set* instructions necessary to implement lock-free algorithms.

Hashtables are very commonly in use to efficiently store huge but sparsely filled tables. As far as we know, no wait- or lock-free algorithm for hashtables has ever been proposed. There are very general solutions for wait-free addresses in general [1, 2, 6, 9, 10], but these are not efficient. Furthermore, there exist wait-free algorithms for different domains, such as linked lists [19], queues [20] and memory management [8, 11]. In this paper we present an almost wait-free algorithm for hashtables. Strictly speaking, the algorithm is only lock-free, but wait-freedom is only violated when a hashtable is resized, which is a relatively rare operation. We allow fully parallel *insertion*, *deletion* and *finding* of elements. As a correctness notion, we take that the operations behave the same as for ‘ordinary’ hashtables, under some arbitrary serialization of these operations. So, if a *find* is carried out strictly after an *insert*, the inserted element is found. If *insert* and *find* are carried out at the same time, it may be that *find* takes place before *insertion*, and it is not determined whether an element will be returned.

An important feature of our hashtable is that it can dynamically grow and shrink when needed. This requires subtle provisions, which can be best understood by considering the following scenarios. Suppose that process *A* is about to (slowly) insert an element in a hashtable H_1 . Before this happens, however, a fast process *B* has resized the hashtable by making a new hashtable H_2 , and has copied the content from H_1 to H_2 . If (and only if) process *B* did not copy the insertion of *A*, *A* must be informed to move to the new hashtable, and carry out the insertion there. Suppose a process *C* comes into play also copying the content from H_1 to H_2 . This must be possible, since otherwise *B* can stop copying, blocking all operations of other processes on the hashtable, and thus violating the lock-free nature of the algorithm. Now the value inserted by *A* can but need not be copied by both *B* and/or *C*. This can be made more complex by a process *D* that attempts to replace H_2 by H_3 . Still, the value inserted by *A* should show up exactly once in the hashtable, and it is clear that processes should carefully keep each other informed about their activities on the tables.

A true problem of lock-free algorithms is that they are hard to design correctly, which even holds for apparently straightforward algorithms. Whereas human imagination generally suffices to deal with all possibilities of sequential processes or synchronized parallel processes, this appears impossible (at least to us) for lock-free algorithms. The only technique that we see fit for any but the simplest lock-free algorithms is to prove the correctness of the algorithm very precisely, and to double check this using a proof checker or theorem prover.

Our algorithm contains 81 atomic statements. The structure of our algorithm and its correctness properties, as well as the complexity of reasoning about them, makes neither automatic nor manual verification feasible. We have therefore chosen the higher-order interactive theorem prover PVS [3, 18] for mechanical support. PVS has a convenient specification language and contains a proof checker which allows users to construct proofs interactively, to automatically execute trivial proofs, and to check these proofs mechanically.

Our solution is as efficient as sequential hashtables. It requires on average only constant time for insertion, deletion or accessing of elements.

Overview of the paper

Section 2 contains the description of the hashtable interface offered to the users. The algorithm is presented in Section 3. Section 4 contains a description of the proof of the safety properties of the algorithm: functional correctness, atomicity, and absence of memory loss. This proof is based on a list of around 200 invariants, presented in Appendix A, while the relationships between the invariants are given by a dependency graph in Appendix B. Progress of the algorithm is proved informally in Section 5. Conclusions are drawn in Section 6.

2 The interface

The aim is to construct a hashtable that can be accessed simultaneously by different processes in such a way that no process can passively block another process' access to the table.

A hashtable is an implementation of (partial) functions between two domains, here called *Address* and *Value*. The hashtable thus implements a modifiable shared variable $\mathbf{X} \in \text{Address} \rightarrow \text{Value}$. The domains *Address* and *Value* both contain special default elements $0 \in \text{Address}$ and $\mathbf{null} \in \text{Value}$. An equality $\mathbf{X}(a) = \mathbf{null}$ means that no value is currently associated with the address a . In particular, since we never store a value for the address 0, we impose the invariant

$$\mathbf{X}(0) = \mathbf{null} .$$

We use open addressing to keep all elements within the table. For the implementation of the hashtables we require that from every value the address it corresponds to is derivable. We therefore assume that some function $ADR \in \text{Value} \rightarrow \text{Address}$ is given with the property that

$$\text{Ax1: } v = \mathbf{null} \equiv ADR(v) = 0$$

Indeed, we need **null** as the value corresponding to the undefined addresses and use address 0 as the (only) address associated with the value **null**. We thus require the hashtable to satisfy the invariant

$$\mathbf{X}(a) \neq \mathbf{null} \Rightarrow ADR(\mathbf{X}(a)) = a .$$

Note that the existence of ADR is not a real restriction since one can choose to store the pair (a, v) instead of v . When a can be derived from v , it is preferable to store v , since that saves memory.

There are four principle operations: *find*, *delete*, *insert* and *assign*. The first one is to *find* the value currently associated with a given address. This operation yields **null** if the address has no associated value. The second operation is to *delete* the value currently associated with a given address. It fails if the address was empty, i.e. $\mathbf{X}(a) = \mathbf{null}$. The third operation is to *insert* a new value for a given address, provided the address was empty. So, note that at least one out of two consecutive *inserts* for address a must fail, except when there is a *delete* for address a in between them. The operation *assign* does the same as *insert*, except that it rewrites the value even if the associated address is not empty. Moreover, *assign* never fails.

We assume that there is a bounded number of processes that may need to interact with the hashtable. Each process is characterized by the sequence of operations

$$(\text{getAccess} ; (\text{find} + \text{delete} + \text{insert} + \text{assign})^* ; \text{releaseAccess})^\omega$$

A process that needs to access the table, first calls the procedure *getAccess* to get the current hashtable pointer. It may then invoke the procedures *find*, *delete*, *insert*, and *assign* repeatedly, in an arbitrary, serial manner. A process that has access to the table can call *releaseAccess* to log out. The processes may call these procedures concurrently. The only restriction is that every process can do at most one invocation at a time.

The basic correctness conditions for concurrent systems are functional correctness and atomicity, say in the sense of [16], Chapter 13. Functional correctness is expressed by prescribing how the procedures *find*, *insert*, *delete*, *assign* affect the value of the abstract mapping \mathbf{X} . Atomicity is expressed by the condition that the modification of \mathbf{X} is executed atomically at some time between

the invocation of the routine and its response. Each of these procedures has the precondition that the calling process has access to the table. In this specification, we use auxiliary private variables declared locally in the usual way. We give them the suffix S to indicate that the routines below are the specifications of the procedures. We use angular brackets $\langle \dots \rangle$ to indicate atomic execution of the enclosed command.

```

proc findS(a : Address \ {0}) : Value =
  local rS : Value;
  (fS)   ⟨ rS := X(a) ⟩;
  return rS.

proc deleteS(a : Address \ {0}) : Bool =
  local sucS : Bool;
  (dS)   ⟨ sucS := (X[a] ≠ null) ;
         if sucS then X[a] := null end ⟩ ;
  return sucS.

proc insertS(v : Value \ {null}) : Bool =
  local sucS : Bool ; a : Address := ADR(v) ;
  (iS)   ⟨ sucS := (X[a] = null) ;
         if sucS then X[a] := v end ⟩ ;
  return sucS.

proc assignS(v : Value \ {null}) =
  local a : Address := ADR(v) ;
  (aS)   ⟨ X[a] := v ⟩ ;
  end.

```

Note that, in all cases, we require that the body of the procedure is executed atomically at some moment between the beginning and the end of the call, but that this moment need not coincide with the beginning or end of the call. This is the reason that we do not (e.g.) specify *find* by the single line **return** X(*a*).

Due to the parallel nature of our system we cannot use pre and postconditions to specify it. For example, it may happen that *insert*(*v*) returns *true* while X(ADR(*v*)) = null since another process deletes ADR(*v*) between the execution of (iS) and the response of *insert*.

We prove partial correctness by extending the implementation with the auxiliary variables and commands used in the specification. So, we regard X as a shared auxiliary variable and *rS* and *sucS* as private auxiliary variables; we augment the implementations of *find*, *delete*, *insert*, *assign* with the atomic commands (fS), (dS), (iS), (aS), respectively. We prove that the implementation of the procedure below executes its atomic specification command always precisely once and that the resulting value *r* or *suc* of the implementation equals the resulting value *rS* or *sucS* in the specification above. It follows that, by removing the implementation variables from the combined program, we obtain the specification. This removal may eliminate many atomic steps of the implementation. This is known as removal of stutterings in TLA [14] or abstraction from τ steps in process algebras.

3 The algorithm

An implementation consists of P processes along with a set of variables, for $P \geq 1$. Each process, numbered from 1 up to P , is a sequential program comprised of atomic statements. Actions on private variables can be added to an atomic statement, but all actions on shared variables must be separated into atomic accesses. Since auxiliary variables are only used to facilitate the proof of correctness, they can be assumed to be touched instantaneously without violation of the atomicity restriction.

3.1 Hashing

We implement function `X` via hashing with open addressing, cf. [15, 21]. We do not use direct chaining, where colliding entries are stored in a secondary list, because maintaining these lists in a lock-free manner is tedious [19], and expensive when done wait-free. A disadvantage of open addressing with deletion of elements is that the contents of the hashtable must regularly be refreshed by copying the non-deleted elements to a new hashtable. As we wanted to be able to resize the hashtables anyhow, we consider this less of a burden.

In principle, hashing is a way to store address-value pairs in an array (hashtable) with a length much smaller than the number of potential addresses. The indices of the array are determined by a hash function. In case the hash function maps two addresses to the same index in the array there must be some method to determine an alternative index. The question how to choose a good hash function and how to find alternative locations in the case of open addressing is treated extensively elsewhere, e.g. [15].

For our purposes it is convenient to combine these two roles in one abstract function `key` given by:

$$\text{key}(a : \text{Address}, l : \text{Nat}, n : \text{Nat}) : \text{Nat} ,$$

where l is the length of the array (hashtable), that satisfies

$$\text{Ax2: } 0 \leq \text{key}(a, l, n) < l$$

for all a , l , and n . The number n serves to obtain alternative locations in case of collisions: when there is a collision, we re-hash until an empty “slot” (i.e. `null`) or the same address in the table is found. The approach with a third argument n is unusual but very general. It is more usual to have a function `Key` dependent on a and l , and use a second function `Inc`, which may depend on a and l , to use in case of collisions. Then our function `key` is obtained recursively by

$$\text{key}(a, l, 0) = \text{Key}(a, l) \text{ and } \text{key}(a, l, n + 1) = \text{Inc}(a, l, \text{key}(a, l, n)) .$$

We require that, for any address a and any number l , the first l keys are all different, as expressed in

$$\text{Ax3: } 0 \leq k < m < l \Rightarrow \text{key}(a, l, k) \neq \text{key}(a, l, m) .$$

3.2 Tagging of values

In hashtables with open addressing a deleted value cannot be replaced by `null` since `null` signals the end of the search. Therefore, such a replacement would invalidate searches for other values. Instead, we introduce an additional “value” `del` to replace deleted values.

Since we want the values in the hashtable to migrate to a bigger table when the first table becomes full, we need to tag values that are being migrated. We cannot simply remove such a value from the old table, since the migrating process may stop functioning during the migration. Therefore, a value being copied must be tagged in such a way that it is still recognizable. This is done by the function `old`. We thus introduce an extended domain of values to be called `EValue`, which is defined as follows:

$$\text{EValue} = \{\text{del}\} \cup \text{Value} \cup \{\text{old}(v) \mid v \in \text{Value}\}$$

We furthermore assume the existence of functions $\text{val} : \text{EValue} \rightarrow \text{Value}$ and $\text{oldp} : \text{EValue} \rightarrow \text{Bool}$ that satisfy, for all $v \in \text{Value}$:

$$\begin{aligned} \text{val}(v) &= v \\ \text{val}(\text{del}) &= \text{null} \\ \text{val}(\text{old}(v)) &= v \\ \text{oldp}(v) &= \text{false} \\ \text{oldp}(\text{del}) &= \text{false} \\ \text{oldp}(\text{old}(v)) &= \text{true} \end{aligned}$$

Note that the *old* tag can easily be implemented by designating one special bit in the representation of *Value*. In the sequel we write **done** for *old(null)*. Moreover, we extend the function *ADR* to domain *EValue* by $ADR(v) = ADR(val(v))$.

3.3 Data structure

A *Hashtable* is either \perp , indicating the absence of a hashtable, or it has the following structure:

```
size : Nat;
occ : Nat;
dels : Nat;
bound : Nat;
table : array 0 .. size-1 of EValue.
```

The field **size** indicates the size of the hashtable, **bound** the maximal number of places that can be occupied before refreshing the table. Both are set when creating the table and remain constant. The variable **occ** gives the number of occupied positions in the table, while the variable **dels** gives the number of deleted positions. If h is a pointer to a hashtable, we write $h.\text{size}$, $h.\text{occ}$, $h.\text{dels}$ and $h.\text{bound}$ to access these fields of the hashtable. We write $h.\text{table}[i]$ to access the i^{th} *EValue* in the table.

Apart from the *current* hashtable, which is the main representative of the variable **X**, we have to deal with *old* hashtables, which were in use before the current one, and *new* hashtables, which can be created after the current one.

We now introduce data structures that are used by the processes to find and operate on the hashtable and allow to delete hashtables that are not used anymore. The basic idea is to count the number of processes that are using a hashtable, by means of a counter **busy**. The hashtable can be thrown away when **busy** is set to 0. An important observation is that **busy** cannot be stored as part of the hashtable, in the same way as the variables **size**, **occ** and **bound** above. The reason for this is that a process can attempt to access the current hashtable by increasing its **busy** counter. However, just before it wants to write the new value for **busy** it falls asleep. When the process wakes up the hashtable might have been deleted and the process would be writing at a random place in memory.

This forces us to use separate arrays **H** and **busy** to store the pointers to hashtables and the **busy** counters. There can be $2P$ hashtables around, because each process can simultaneously be accessing one hashtable and attempting to create a second one. The arrays below are shared variables.

```
H : array 1 .. 2P of pointer to Hashtable ;
busy : array 1 .. 2P of Nat ;
prot : array 1 .. 2P of Nat ;
next : array 1 .. 2P of 0 .. 2P .
```

As indicated, we also need arrays **prot** and **next**. The variable **next**[i] points to the next hashtable to which the contents of hashtable **H**[i] is being copied. If **next**[i] equals 0, this means that there is no next hashtable. The variable **prot**[i] is used to guard the variables **busy**[i], **next**[i] and **H**[i] against being reused for a new table, before all processes have discarded these.

We use a shared variable **currInd** to hold the index of the currently valid hashtable:

```
currInd : 1 .. 2P .
```

Note however that after a process copies **currInd** to its local memory, other processes may create a new hashtable and change **currInd** to point to that one.

3.4 Primary procedures

We first provide the code for the primary procedures, which match directly with the procedures in the interface. Every process has a private variable

```
index : 1 .. 2P;
```

containing what it regards as the currently active hashtable. At entry of each primary procedure, it must be the case that the variable $H[index]$ contains valid information. In section 3.5, we provide procedure `getAccess` with the main purpose to guarantee this property. When `getAccess` has been called, the system is obliged to keep the hashtable at `index` stored in memory, even if there are no accesses to the hashtable using any of the primary procedures. A procedure `releaseAccess` is provided to release resources, and it should be called whenever the process will not access the hashtable for some time.

3.4.1 Syntax

We use a syntax analogous to Modula-3 [5]. We use `:=` for the assignment. We use the C-operations `++` and `--` for atomic increments and decrements. The semicolon is a separator, not a terminator. The basic control mechanisms are

```
loop .. end  is an infinite loop, terminated by exit or return
while .. do .. end  and repeat .. until ..  are ordinary repetitions
if .. then .. {elsif ..} [else ..] end  is the conditional
case .. end  is a case statement.
```

Types are slanted and start with a capital. Shared variables and shared data elements are in typewriter font. Private variables are slanted or in math italic.

3.4.2 The main loop

We model the clients of the hashtable in the following loop. Note that this is not an essential part of the algorithm, but it is needed in the PVS description, and therefore provided here.

```
loop
0:      getAccess() ;
loop
1:      choose call; case call of
        (f, a) with a ≠ 0 → find(a)
        (d, a) with a ≠ 0 → delete(a)
        (i, v) with v ≠ null → insert(v)
        (a, v) with v ≠ null → assign(v)
        (r) → releaseAccess(index); exit
      end
    end
  end
```

The main loop shows that each process repeatedly invokes its four principle operations with correct arguments in an arbitrary, serial manner. Procedure `getAccess` has to provide the client with a protected value for `index`. Procedure `releaseAccess` releases this value and its protection. Note that `exit` means a jump out of the inner loop.

3.4.3 Find

Finding an address in a hashtable with open addressing requires a linear search over the possible hash keys until the address or an empty slot is found. The kernel of procedure `find` is therefore:

```

n := 0 ;
repeat r := h.table[key(a,l,n)] ; n++ ;
until r = null ∨ a = ADR(r) ;

```

The main complication is that the process has to join the migration activity by calling *refresh* when it encounters an entry **done** (i.e. *old(null)*).

Apart from a number of special commands, we group statements such that at most one shared variable is accessed and label these ‘atomic’ statements with a number. The labels are chosen identical to the labels in the PVS code, and therefore not completely consecutive.

In every execution step, one of the processes proceeds from one label to a next one. The steps are thus treated as atomic. The atomicity of steps that refer to shared variables more than once is emphasized by enclosing them in angular brackets. Since procedure calls only modify private control data, procedure headers are not always numbered themselves, but their bodies usually have numbered atomic statements.

```

proc find(a : Address \ {0}) : Value =
    local r : EValue ; n, l : Nat ; h : pointer to Hashtable ;
5:    h := H[index] ; n := 0 ; {cnt := 0} ;
6:    l := h.size ;
    repeat
7:        { r := h.table[key(a,l,n)] ;
          { if r = null ∨ a = ADR(r) then cnt++ ; (fS) end } } ;
8:        if r = done then
            refresh() ;
10:       h := H[index] ; n := 0 ;
11:       l := h.size ;
            else n++ end ;
13:    until r = null ∨ a = ADR(r) ;
14:    return val(r) .

```

In order to prove correctness, we add between braces instructions that only modify auxiliary variables, like the specification variables **X** and **rS** and other auxiliary variables to be introduced later. The part between braces is comment for the implementation, it only serves in the proof of correctness. The private auxiliary variable *cnt* of type *Nat* counts the number of times (fS) is executed and serves to prove that (fS) is executed precisely once in every call of *find*.

This procedure matches the code of an ordinary find in a hashtable with open addressing, except for the code at the condition *r = done*. This code is needed for the case that the value *r* is being copied, in which case the new table must be located. Locating the new table is carried out by the procedure *refresh*, which is discussed in Section 3.5. In line 7, the accessed hashtable should be valid (see invariants *fi4* and *He4* in Appendix A). After *refresh* the local variables *n*, *h* and *l* must be reset, to restart the search in the new hashtable. If the procedure terminates, the specifying atomic command (fS) has been executed precisely once (see invariant *Cn1*) and the return values of the specification and the implementation are equal (see invariant *Co1*). If the operation succeeds, the return value must be a valid entry currently associated with the given address in the current hashtable. It is not evident but it has been proved that the linear search of the process executing *find* cannot be violated by other processes, i.e. no other process can *delete*, *insert*, or *rewrite* an entry associated with the same address (as what the process is looking for) in the region where the process has already searched.

We require that there exist at least one **null** entry or **done** entry in any valid hashtable, hence the local variable *n* in the procedure *find* will never go beyond the size of the hashtable (see invariants *Cu1*, *fi4*, *fi5* and axiom *Ax2*). When the **bound** of the new hashtable is tuned properly before use (see invariants *Ne7*, *Ne8*), the hashtable will not be updated too frequently, and termination of the procedure *find* can be guaranteed.

3.4.4 Delete

Deletion is similar to finding. Since r is a local variable to the procedure *delete*, we regard 18a and 18b as two parts of atomic instruction 18. If the entry is found in the table, then at line 18b this entry is overwritten with the designated element **del**.

```

proc delete( $a : \text{Address} \setminus \{\text{null}\}$ ) :  $\text{Bool} =$ 
    local  $r : E\text{Value}$  ;  $k, l, n : \text{Nat}$  ;  $h : \text{pointer to Hashtable}$  ;  $suc : \text{Bool}$  ;
15:     $h := H[\text{index}]$  ;  $suc := \text{false}$  ;  $\{cnt := 0\}$  ;
16:     $l := h.\text{size}$  ;  $n := 0$  ;
        repeat
17:         $k := \text{key}(a, l, n)$  ;
             $\langle r := h.\text{table}[k]$  ;
                 $\{ \text{if } r = \text{null} \text{ then } cnt++ ; (\text{dS}) \text{ end } \} \rangle$  ;
18a:       if  $oldp(r)$  then
             $\text{refresh}()$  ;
20:            $h := H[\text{index}]$  ;
21:            $l := h.\text{size}$  ;  $n := 0$  ;
            elsif  $a = ADR(r)$  then
18b:                $\langle \text{if } r = h.\text{table}[k] \text{ then}$ 
                     $suc := \text{true}$  ;  $h.\text{table}[k] := \text{del}$  ;
                     $\{ cnt++ ; (\text{dS}) ; Y[k] := \text{del} \}$ 
                 $\text{end} \rangle$  ;
                else  $n++$  end ;
            until  $suc \vee r = \text{null}$  ;
25:       if  $suc$  then  $h.\text{dels}++$  end ;
26:       return  $suc$  .

```

In this procedure, there are two possibilities if r is not outdated in each loop: either deletion fails with $r = \text{null}$ in 17 or deletion succeeds with $r = h.\text{table}[k]$ in 18b. In the latter case, we have in one atomic statement a double access of the shared variable $h.\text{table}[k]$. This is a so-called compare&swap instruction. Atomicity is needed here to preclude interference. The specifying command (dS) is executed either in 17 or in 18, and it is executed precisely once (see invariant *Cn2*), since in 18 the guard $a = ADR(r)$ implies $r \neq \text{null}$ (see invariant *de1* and axiom *Ax1*).

In order to remember the address from the value rewritten to **done** after the value is being copied in the procedure *moveContents*, in 18, we introduce a new auxiliary shared variable Y of type array of $E\text{Value}$, whose contents equals the corresponding contents of the current hashtable almost everywhere except that the values it contains are not tagged to be *old* or rewritten to be **done** (see invariants *Cu9*, *Cu10*).

Since we postpone the increment of $h.\text{dels}$ until line 25, the field **dels** is a lower bound of the number of positions deleted in the hashtable (see invariant *Cu4*).

3.4.5 Insert

The procedure for insertion in the table is given below. Basically, it is the standard algorithm for insertion in a hashtable with open addressing. Notable is line 28 where the current process finds the current hashtable too full, and orders a new table to be made. We assume that $h.\text{bound}$ is a number less than $h.\text{size}$ (see invariant *Cu3*), which is tuned for optimal performance. Furthermore, in line 35, it can be detected that values in the hashtable have been marked *old*, which is a sign that hashtable h is outdated, and the new hashtable must be located to perform the insertion.

```

proc insert( $v : \text{Value} \setminus \{\text{null}\}$ ) :  $\text{Bool} =$ 
    local  $r : E\text{Value}$  ;  $k, l, n : \text{Nat}$  ;  $h : \text{pointer to Hashtable}$  ;
         $suc : \text{Bool}$  ;  $a : \text{Address} := ADR(v)$  ;
27:     $h := H[\text{index}]$  ;  $\{cnt := 0\}$  ;
28:    if  $h.\text{occ} > h.\text{bound}$  then

```

```

            newTable() ;
30:        h := H[index] end ;
31:        n := 0 ; l := h.size ; suc := false ;
repeat
32:        k := key(a, l, n) ;
33:        ⟨ r := h.table[k] ;
            { if a = ADR(r) then cnt++ ; (iS) end } ⟩ ;
35a:        if oldp(r) then
            refresh() ;
36:        h := H[index] ;
37:        n := 0 ; l := h.size ;
        elseif r = null then
35b:        { if h.table[k] = null then
            suc := true ; h.table[k] := v ;
            { cnt++ ; (iS) ; Y[k] := v }
            end } ;
            else n++ end ;
        until suc ∨ a = ADR(r) ;
41:        if suc then h.occ++ end ;
42:        return suc .

```

Instruction 35b is a test&set instruction, a simpler version of compare&swap. Procedure *insert* terminates successfully when the insertion to an empty slot is completed, or it fails when there already exists an entry with the given address currently in the hashtable (see invariant Co3 and the specification of *insert*).

3.4.6 Assign

Procedure *assign* is almost the same as *insert* except that it rewrites an entry with a give value even when the associated address is not empty. We provide it without further comments.

```

proc assign(v : Value \ {null}) =
    local r : EValue ; k, l, n : Nat ; h : pointer to Hashtable ;
        suc : Bool ; a : Address := ADR(v) ;
43:        h := H[index] ; cnt := 0;
44:        if h.occ > h.bound then
            newTable() ;
46:        h := H[index] end ;
47:        n := 0 ; l := h.size ; suc := false ;
repeat
48:        k := key(a, l, n) ;
49:        r := h.table[k] ;
50a:        if oldp(r) then
            refresh() ;
51:        h := H[index] ;
52:        n := 0 ; l := h.size ;
        elseif r = null ∨ a = ADR(r) then
50b:        { if h.table[k] = r then
            suc := true ; h.table[k] := v ;
            { cnt++ ; (aS) ; Y[k] := v }
            end }
            else n++ end ;
        until suc ;
57:        if r = null then h.occ++ end ;
end.

```

3.5 Memory management and concurrent migration

In this section, we provide the public procedures `getAccess` and `releaseAccess` and the auxiliary procedures `refresh` and `newTable`. Since `newTable` and `releaseAccess` have the responsibilities for allocations and deallocations, we begin with the treatment of memory by providing a model of the heap.

3.5.1 The model of the heap

We *model* the `Heap` as an infinite array of hashtables, declared and initialized in the following way:

```
Heap : array Nat of Hashtable := ([Nat]⊥) ;
H_index : Nat := 1 .
```

So, initially, $\text{Heap}[i] = \perp$ for all indices i . The indices of array `Heap` are the pointers to hashtables. We thus simply regard **pointer to Hashtable** as a synonym of `Nat`. Therefore, the notation `h.table` used elsewhere in the paper stands for `Heap[h].table`. Since we reserve 0 (to be distinguished from the absent hashtable \perp and the absent value `null`) for the null pointer (i.e. $\text{Heap}[0] = \perp$, see invariant `He1`), we initialize `H_index`, which is the index of the next hashtable, to be 1 instead of 0. Allocation of memory is modeled in

```
proc allocate(s, b : Nat) : Nat =
  ⟨ Heap[H_index] := blank hashtable with size = s, bound = b, occ = dels = 0 ;
    H_index++ ⟩ ;
  return H_index ;
```

We assume that `allocate` sets all values in the hashtable `Heap[H_index]` to `null`, and also sets its fields `size`, `bound`, `occ` and `dels` appropriately. Deallocation of hashtables is modeled by

```
proc deAlloc(h : Nat) =
  ⟨ assert Heap[h] ≠ ⊥ ; Heap[h] := ⊥ ⟩
end .
```

The **assert** in `deAlloc` indicates the obligation to prove that `deAlloc` is called only for allocated memory.

3.5.2 GetAccess

The procedure `getAccess` is defined as follows.

```
proc getAccess() =
  loop
    59:   index := currInd;
    60:   prot[index]++;
    61:   if index = currInd then
    62:     busy[index]++;
    63:     if index = currInd then return;
        else releaseAccess(index) end;
    65:     else prot[index]--;
  end
end.
```

This procedure is a bit tricky. When the process reaches line 62, the `index` has been protected not to be used for creating a new hashtable in the procedure `newTable` (see invariants `pr2`, `pr3` and `nT12`).

The hashtable pointer `H[index]` must contain the valid contents after the procedure `getAccess` returns (see invariants `Ot3`, `He4`). So, in line 62, `busy` is increased, guaranteeing that the hashtable will not inadvertently be destroyed (see invariant `bu1` and line 69). Line 63 needs to check the

index again in case that instruction 62 has the precondition that the hashtable is not valid. Once some process gets hold of one hashtable after calling *getAccess*, no process can throw it away until the process releases it (see invariant *rA7*). Note that this is using *releaseAccess* implicitly done in *refresh*.

3.5.3 ReleaseAccess

The procedure *releaseAccess* is given by

```

proc releaseAccess(i : 1 . . 2P) =
    local h : pointer to Hashtable ;
77:    h := H[i] ;
68:    busy[i]-- ;
69:    if h ≠ 0 ∧ busy[i] = 0 then
70:        ⟨ if H[i] = h then H[i] := 0 ; ⟩
71:        deAlloc(h) ;
            end ;
        end ;
72:    prot[i]-- ;
end.
```

Since *deAlloc* in line 71 accesses a shared variable, we have separated its call from 70. The counter *busy[i]* is used to protect the hashtable from premature deallocation. Only if *busy[i]=0*, *H[i]* can be released. The main problem of the design at this point is that it can happen that several processes concurrently execute *releaseAccess* for the same value of *i*, with interleaving just after the decrement of *busy[i]*. Then they all may find *busy[i] = 0*. Therefore, a bigger atomic command is needed to ensure that precisely one of them sets *H[i]* to 0 (line 70) and calls *deAlloc*. Indeed, in line 71, *deAlloc* is called only for allocated memory (see invariant *rA3*). The counter *prot[i]* can be decreased since position *i* is no longer used by this process.

3.5.4 NewTable

When the current hashtable has been used for some time, some actions of the processes may require replacement of this hashtable. Procedure *newTable* is called when the number of occupied positions in the current hashtable exceeds the bound (see lines 28, 44). Procedure *newTable* tries to allocate a new hashtable as the successor of the current one (i.e. the next current hashtable). If several processes call *newTable* concurrently, they need to reach consensus on the choice of an index for the next hashtable (line 78). A newly allocated hashtable that will not be used must be deallocated again.

```

proc newTable() =
    local i : 1 . . 2P ; b, bb : Bool ;
77:    while next[index] = 0 do
78:        choose i ∈ 1 . . 2P ;
            ⟨ b := (prot[i] = 0) ;
                if b then prot[i] := 1 end ⟩ ;
            if b then
                busy[i] := 1 ;
                choose bound > H[index].bound - H[index].dels + 2P ;
                choose size > bound + 2P ;
                H[i] := allocate(size, bound) ;
81:                next[i] := 0 ;
82:                choose bb := (next[index] = 0) ;
                    ⟨ bb := (next[index] = 0) ;
                        if bb then next[index] := i end ⟩ ;
                        if bb then releaseAccess(i) end ;
```

```

    end end ;
    refresh() ;
end .

```

In command 82, we allocate a new blank hashtable (see invariant $nT8$), of which the **bound** is set greater than $H[index].bound - H[index].dels + 2P$ in order to avoid creating a too small hashtable (see invariants $nT6$, $nT7$). The variables **occ** and **dels** are initially 0 because the hashtable is completely filled with the value **null** at this moment.

We require the **size** of a hashtable to be more than **bound**+ $2P$ because of the following scenario: P processes find “ $h.occ > h.bound$ ” at line 28 and call *newtable*, *refresh*, *migrate*, *moveContents* and *moveElement* one after the other. After moving some elements, all processes but process p sleep at line 126 with $b_{mE} = \text{true}$ (b_{mE} is the local variable b of procedure *moveElement*). Process p continues the migration and updates the new current index when the migration completes. Then, process p does several insertions to let the **occ** of the current hashtable reach one more than its **bound**. Just at that moment, $P - 1$ processes wake up, increase the **occ** of the current hashtable to be $P - 1$ more, and return to line 30. Since $P - 1$ processes insert different values in the hashtable, after $P - 1$ processes finish their insertions, the **occ** of the current hashtable reaches $2P - 1$ more than its **bound**.

It may be useful to make **size** larger than **bound**+ $2P$ to avoid too many collisions, e.g. with a constraint $\text{size} \geq \alpha \cdot \text{bound}$ for some $\alpha > 1$. If we did not introduce **dels**, every migration would force the sizes to grow, so that our hashtable would require unbounded space for unbounded life time. We introduced **dels** to avoid this.

Strictly speaking, instruction 82 inspects one shared variable, $H[index]$, and modifies three other shared variables, viz. $H[i]$, $\text{Heap}[H_index]$, and H_index . In general, we split such multiple shared variable accesses in separate atomic commands. Here the accumulation is harmless, since the only possible interferences are with other allocations at line 82 and deallocations at line 71. In view of the invariant $Ha2$, all deallocations are at pointers $h < H_index$. Allocations do not interfere because they contain the increment $H_index++$ (see procedure *allocate*).

The procedure *newTable* first searches for a free index i , say by round robin. We use a nondeterministic choice. Once a free index has been found, a hashtable is allocated and the index gets an indirection to the allocated address. Then the current index gets a **next** pointer to the new index, unless this pointer has been set already.

The variables $\text{prot}[i]$ are used primarily as counters with atomic increments and decrements. In 78, however, we use an atomic test-and-set instruction. Indeed, separation of this instruction in two atomic instructions is incorrect, since that would allow two processes to grab the same index i concurrently.

3.5.5 Migrate

After the choice of the next current hashtable, the procedure *migrate* has the task to transfer the contents in the current hashtable to the next current hashtable by calling a procedure *moveContents* and update the current hashtable pointer afterwards. Migration is complete when at least one of the (parallel) calls to *migrate* has terminated.

```

proc migrate() =
    local i : 0 .. 2P; h : pointer to Hashtable ; b : Bool ;
94:    i := next[index];
95:    prot[i]++;
97:    if index ≠ currInd then
98:        prot[i]--;
else
99:        busy[i]++;
100:       h := H[i];
101:       if index = currInd then
               moveContents(H[index], h);

```

```

103:      ⟨ b := (currInd = index) ;
104:          if b then currInd := i ;
105:              {Y := H[i].table }
           end ) ;
           if b then
               busy[index]-- ;
               prot[index]-- ;
               end ;
           end ;
           releaseAccess(i) ;
       end end .

```

According to invariants *mi4* and *mi5*, it is an invariant that $i = \text{next}(index) \neq 0$ holds after instruction 94.

Line 103 contains a compare&swap instruction to update the current hashtable pointer when some process finds that the migration is finished while *currInd* is still identical to its *index*, which means that *i* is still used for the next current hashtable (see invariant *mi5*). The increments of *prot[i]* and *busy[i]* here are needed to protect the next hashtable. The decrements serve to avoid memory loss.

3.5.6 Refresh

In order to avoid that a process starts migration of an old hashtable, we encapsulate *migrate* in *refresh* in the following way.

```

proc refresh() =
90:    if index ≠ currInd then
        releaseAccess(index) ;
        getAccess() ;
    else migrate() end ;
end.

```

When *index* is outdated, the process needs to call *releaseAccess* to abandon its hashtable and *getAccess* to acquire the present pointer to the current hashtable. Otherwise, the process can join the migration.

3.5.7 MoveContents

Procedure *moveContents* has to move the contents of the current table to the next current table. All processes that have access to the table, may also participate in this migration. Indeed, they cannot yet use the new table (see invariants *Ne1* and *Ne3*). We have to take care that delayed actions on the current table and the new table are carried out or aborted correctly (see invariants *Cu1* and *mE10*). Migration requires that every value in the current table be moved to a unique position in the new table (see invariant *Ne19*).

Procedure *moveContents* uses a private variable *toBeMoved* that ranges over sets of locations. The procedure is given by

```

proc moveContents(from, to : pointer to Hashtable) =
    local i : Nat ; b : Bool ; v : EValue} ; toBeMoved : set of Nat ;
    toBeMoved := {0, …, from.size – 1} ;
110:   while currInd = index ∧ toBeMoved ≠ ∅ do
111:       choose i ∈ toBeMoved ;
           v := from.table[i] ;
           if from.table[i] = done then
               toBeMoved := toBeMoved – {i} ;
           else

```

```

114:   ⟨  $b := (v = \text{from.table}[i])$  ;
      if  $b$  then  $\text{from.table}[i] := \text{old}(\text{val}(v))$  end ⟩ ;
      if  $b$  then
        if  $\text{val}(v) \neq \text{null}$  then  $\text{moveElement}(\text{val}(v), \text{to})$  end ;
         $\text{from.table}[i] := \text{done}$  ;
         $\text{toBeMoved} := \text{toBeMoved} - \{i\}$  ;
      end end end ;
    end .

```

Note that the value is tagged as outdated before being duplicated (see invariant $mC11$). After tagging, the value cannot be deleted or assigned until the migration has been completed. Tagging must be done atomically, since otherwise an interleaving deletion may be lost. When indeed the value has been copied to the new hashtable, in line 117 that value becomes **done** in the hashtable. This has the effect that other processes need not wait for this process to complete procedure moveElement , but can help with the migration of this value if needed.

Since the address is lost after being rewritten to **done**, we had to introduce the shared auxiliary hashtable Y to remember its value for the proof of correctness. This could have been avoided by introducing a second tagging bit, say for “very old”.

The processes involved in the same migration should not use the same strategy for choosing i in line 111, since it is advantageous that moveElement is called often with different values. They may exchange information: any of them may replace its set toBeMoved by the intersection of that set with the set toBeMoved of another one. We do not give a preferred strategy here, one can refer to algorithms for the *write-all* problem [4, 13].

3.5.8 MoveElement

The procedure moveElement moves a value to the new hashtable. Note that the value is tagged as outdated in moveContents before moveElement is called.

```

proc  $\text{moveElement}(v : \text{Value} \setminus \{\text{null}\}, \text{to} : \text{pointer to Hashtable}) =$ 
  local  $a : \text{Address}$  ;  $k, m, n : \text{Nat}$  ;  $w : E\text{Value}$  ;  $b : \text{Bool}$  ;
120:   $n := 0$  ;  $b := \text{false}$  ;  $a := \text{ADR}(v)$  ;  $m := \text{to.size}$  ;
  repeat
121:   $k := \text{key}(a, m, n)$  ;  $w := \text{to.table}[k]$  ;
    if  $w = \text{null}$  then
123:    ⟨  $b := (\text{to.table}[k] = \text{null})$  ;
        if  $b$  then  $\text{to.table}[k] := v$  end ⟩ ;
        else  $n++$  end ;
125:    until  $b \vee a = \text{ADR}(w) \vee \text{currInd} \neq \text{index}$  ;
126:    if  $b$  then  $\text{to.occ}++$  end
  end .

```

The value is only allowed to be inserted once in the new hashtable (see invariant $Ne19$), otherwise it will violate the main property of open addressing. In total, four situations can occur in the procedure moveElement :

- the current location k contains a value with an other address, the process will increase n and inspect the next location.
- the current location k contains a value with the same address, which means the value has been copied to the new hashtable already. The process therefore terminates.
- the current location k is an empty slot. The process inserts v and returns. If insertion fails, as an other process did fill the empty slot, the search is continued.
- when index happens to differ from currInd , the whole migration has been completed.

While the current hashtable pointer is not updated yet, there exists at least one **null** entry in the new hashtable (see invariants Ne8, Ne22 and Ne23), hence the local variable n in the procedure *moveElement* never goes beyond the size of the hashtable (see invariants *mE3* and *mE8*), and the termination is thus guaranteed.

4 Correctness (Safety)

In this section, we describe the proof of safety of the algorithm. The main aspects of safety are functional correctness, atomicity, and absence of memory loss. These aspects are formalized in eight invariants described in section 4.1. To prove these invariants, we need many other invariants. These are listed in Appendix A. In section 4.2, we sketch the verification of some of the invariants by informal means. In section 4.3, we describe how the theorem prover PVS is used in the verification. As exemplified in 4.2, Appendix B gives the dependencies between the invariants.

Notational Conventions. Recall that there are at most P processes with process identifiers ranging from 1 up to P . We use p, q, r to range over process identifiers, with a preference for p . Since the same program is executed by all processes, every private variable name of a process $\neq p$ is extended with the suffix “.” + “process identifier”. We do not do this for process p . So, e.g., the value of a private variable x of process q is denoted by $x.q$, but the value of x of process p is just denoted by x . In particular, $pc.q$ is the program location of process q . It ranges over all integer labels used in the implementation.

When local variables in different procedures have the same names, we add an abbreviation of the procedure name as a subscript to the name. We use the following abbreviations: *fi* for *find*, *del* for *delete*, *ins* for *insert*, *ass* for *assign*, *gA* for *getAccess*, *rA* for *releaseAccess*, *nT* for *newTable*, *mig* for *migrate*, *ref* for *refresh*, *mC* for *moveContents*, *mE* for *moveElement*.

In the implementation, there are several places where the same procedure is called, say *getAccess*, *releaseAccess*, etc. We introduce auxiliary private variables *return*, local to such a procedure, to hold the return location. We add a procedure subscript to distinguish these variables according to the above convention.

If V is a set, $\#V$ denotes the number of elements of V . If b is a boolean, then $\#b = 0$ when b is false, and $\#b = 1$ when b is true. Unless explicitly defined otherwise, we always (implicitly) universally quantify over addresses a , values v , non-negative integer numbers k, m , and n , natural number l , processes p, q and r . Indices i and j range over $[1, 2P]$. We abbreviate $H(\text{currInd}).\text{size}$ as *curSize*.

In order to avoid using too many parentheses, we use the usual binding order for the operators. We give “ \wedge ” higher priority than “ \vee ”. We use parentheses whenever necessary.

4.1 Main properties

We have proved the following three safety properties of the algorithm. Firstly, the access procedures *find*, *delete*, *insert*, *assign*, are functionally correct. Secondly they are executed atomically. The third safety property is absence of memory loss.

Functional correctness of *find*, *delete*, *insert* is the condition that the result of the implementation is the same as the result of the specification (fS), (dS), (iS). This is expressed by the required invariants:

$$\begin{aligned} Co1: \quad & pc = 14 \Rightarrow \text{val}(r_{fi}) = rS_{fi} \\ Co2: \quad & pc \in \{25, 26\} \Rightarrow \text{suc}_{del} = \text{suc}S_{del} \\ Co3: \quad & pc \in \{41, 42\} \Rightarrow \text{suc}_{ins} = \text{suc}S_{ins} \end{aligned}$$

Note that functional correctness of *assign* holds trivially since it does not return a result.

According to the definition of atomicity in chapter 13 of [16], atomicity means that each execution of one of the access procedures contains precisely one execution of the corresponding specifying action (fS), (dS), (iS), (aS). We introduced the private auxiliary variables *cnt* to count

the number of times the specifying action is executed. Therefore, atomicity is expressed by the invariants:

- Cn1: $pc = 14 \Rightarrow cnt_{fi} = 1$
- Cn2: $pc \in \{25, 26\} \Rightarrow cnt_{del} = 1$
- Cn3: $pc \in \{41, 42\} \Rightarrow cnt_{ins} = 1$
- Cn4: $pc = 57 \Rightarrow cnt_{ass} = 1$

We interpret absence of memory loss to mean that the number of valid hashtables is bounded. More precisely, we prove that this number is bounded by $2P$. This is formalized in the invariant:

$$No1: \# \{k \mid k < H_index \wedge \text{Heap}(k) \neq \perp\} \leq 2P$$

4.2 Intuitive proof

The eight correctness properties (invariants) mentioned above have been completely proved with the interactive proof checker of PVS. The use of PVS did not only take care of the delicate bookkeeping involved in the proof, it could also deal with many trivial cases automatically. At several occasions where PVS refused to let a proof be finished, we actually found a mistake and had to correct previous versions of this algorithm.

In order to give some feeling for the proof, we describe some proofs. For the complete mechanical proof, we refer the reader to [12]. Note that, for simplicity, we assume that all non-specific private variables in the proposed assertions belong to the general process p , and general process q is an active process that tries to threaten some assertion (p may equal q).

Proof of invariant *Co1* (as claimed in 4.1). According to Appendix B, the stability of *Co1* follows from the invariants *Ot3*, *fi1*, *fi10*, which are given in Appendix A. Indeed, *Ot3* implies that no procedure returns to location 14. Therefore all return statements falsify the antecedent of *Co1* and thus preserve *Co1*. Since r_{fi} and rS_{fi} are private variables to process p , *Co1* can only be violated by process p itself (establishing pc at 14) when p executes 13 with $r_{fi} = \text{null} \vee a_{fi} = ADR(r_{fi})$. This condition is abbreviated as *Find*(r_{fi}, a_{fi}). Invariant *fi10* then implies that action 13 has the precondition $\text{val}(r_{fi}) = rS_{fi}$, so then it does not violate *Co1*. In PVS, we used a slightly different definition of *Find*, and we applied invariant *fi1* to exclude that r_{fi} is **done** or **del**, though invariant *fi1* is superfluous in this intuitive proof. \square

Proof of invariant *Ot3*. Since the procedures *getAccess*, *releaseAccess*, *refresh*, *newTable* are called only at specific locations in the algorithm, it is easy to list the potential return addresses. Since the variables *return* are private to process p , they are not modified by other processes. Stability of *Ot3* follows from this. As we saw in the previous proof, *Ot3* is used to guarantee that no unexpected jumps occur. \square

Proof of invariant *fi10*. According to Appendix B, we only need to use *fi9* and *Ot3*. Let us use the abbreviation $k = \text{key}(a_{fi}, l_{fi}, n_{fi})$. Since r_{fi} and rS_{fi} are both private variables, they can only be modified by process p when p is executing statement 7. We split this situation into two cases

1. with precondition *Find*($h_{fi}.\text{table}[k], a_{fi}$)
After execution of statement 7, r_{fi} becomes $h_{fi}.\text{table}[k]$, and rS_{fi} becomes $X(a_{fi})$. By *fi9*, we get $\text{val}(r_{fi}) = rS_{fi}$. Therefore the validity of *fi10* is preserved.
2. otherwise.
After execution of statement 7, r_{fi} becomes $h_{fi}.\text{table}[k]$, which then falsifies the antecedent of *fi10*. \square

Proof of invariant *fi9*. According to Appendix B, we proved that *fi9* follows from *Ax2*, *fi1*, *fi3*, *fi4*, *fi5*, *fi8*, *Ha4*, *He4*, *Cu1*, *Cu9*, *Cu10*, and *Cu11*. We abbreviate $\text{key}(a_{fi}, l_{fi}, n_{fi})$ as k . We

deduce $h_{fi} = H(index)$ from $fi4$, $H(index)$ is not \perp from $He4$, and k is below $H(index).size$ from $Ax2$, $fi4$ and $fi3$. We split the proof into two cases:

1. $index \neq currInd$: By $Ha4$, it follows that $H(index) \neq H(currInd)$. Hence from $Cu1$, we obtain $h_{fi}.table[k] = done$, which falsifies the antecedent of $fi9$.
2. $index = currInd$: By premise $Find(h_{fi}.table[k], a_{fi})$, we know that $h_{fi}.table[k] \neq done$ because of $fi1$. By $Cu9$ and $Cu10$, we obtain $val(h_{fi}.table[k]) = val(Y[k])$. Hence it follows that $Find(Y[k], a_{fi})$. Using $fi8$, we obtain

$$\forall m < n_{fi} : \neg Find(Y[key(a_{fi}, curSize, m)], a_{fi})$$

We get n_{fi} is below $curSize$ because of $fi5$. By $Cu11$, we conclude

$$X(a_{fi}) = val(h_{fi}.table[k])$$

□

4.3 The model in PVS

Our proof architecture (for one property) can be described as a dynamically growing tree in which each node is associated with an assertion. We start from a tree containing only one node, the proof goal, which characterizes some property of the system. We expand the tree by adding some new children via proper analysis of an unproved node (top-down approach, which requires a good understanding of the system). The validity of that unproved node is then reduced to the validity of its children and the validity of some less or equally deep nodes.

Normally, simple properties of the system are proved with appropriate precedence, and then used to help establish more complex ones. It is not a bad thing that some property that was taken for granted turns out to be not valid. Indeed, it may uncover a defect of the algorithm, but in any case it leads to new insights in it.

We model the algorithm as a transition system [17], which is described in the language of PVS in the following way. As usual in PVS, states are represented by a record with a number of fields:

```

State : TYPE = [#%
  % global variables
  ...
  busy : [ range(2*P) → nat ],
  prot : [ range(2*P) → nat ],
  ...
  % private variables:
  index : [ range(P) → range(2*P) ],
  ...
  pc : [ range(P) → nat ], % private program counters
  ...
  % local variables of procedures, also private to each process:
  % find
  a_find : [ range(P) → Address ],
  r_find : [ range(P) → EValue ],
  ...
  % getAccess
  return_getAccess : [ range(P) → nat ],
  ...
#]

```

where $range(P)$ stands for the range of integers from 1 to P .

Note that private variables are given with as argument a process identifier. Local variables are distinguished by adding their procedure's names as suffixes.

An action is a binary relation on states: it relates the state prior to the action to the state following the action. The system performed by a particular process is then specified by defining the precondition of each action as a predicate on the state and also the effect of each action in terms of a state transition. For example, line 5 of the algorithm is described in PVS as follows:

```
% corresponding to statement find5: h := H[index]; n := 0;
find5(i,s1,s2) : bool =
  pc(s1)(i)=5 AND
  s2 = s1 WITH [ (pc)(i) := 6,
    (n_find)(i) := 0,
    (h_find)(i) := H(s1)(index(s1)(i))
  ]
...
```

where i is a process identifier, $s1$ is a pre-state, $s2$ is a post-state.

Since our algorithm is concurrent, the global transition relation is defined as the disjunction of all atomic actions.

```
% transition steps
step(i,s1,s2) : bool =
  find5(i,s1,s2) or find6(i,s1,s2) or ...
  delete15(i,s1,s2) or delete16(i,s1,s2) or ...
...
```

Stability for each invariant has been proved by a *Theorem* in PVS of the form:

```
% Theorem about the stability of invariant fi10
IV_fi10: THEOREM
  forall (u,v : state, q : range(P)) :
    step(q,u,v) AND fi10(u) AND fi9(u) AND ot3(u)
    => fi10(v)
```

To ensure that all proposed invariants are stable, there is a global invariant **INV**, which is the conjunction of all proposed invariants.

```
% global invariant
INV(s:state) : bool =
  He3(s) and He4(s) and Cu1(s) and ...
...
% Theorem about the stability of the global invariant INV
IV_INV: THEOREM
  forall (u,v : state, q : range(P)) :
    step(q,u,v) AND INV(u) => INV(v)
```

We define **Init** as all possible initial states, for which all invariants must be valid.

```
% initial state
Init: { s : state |
  (forall (p: range(P)):
    pc(s)(p)=0 and ...
    ...) and
  (forall (a: Address):
    X(s)(a)=null) and
  ...
}
```

```
% The initial condition can be satisfied by the global invariant INV
IV_Init: THEOREM
  INV(Init)
```

The PVS code contains preconditions to imply well-definedness: e.g. in *find7*, the hashtable must be non-NIL and ℓ must be its size.

```
% corresponding to statement find7
find7(i,s1,s2) : bool =
  i?(Heap(s1)(h_find(s1)(i))) and
  l_find(s1)(i)=size(i_(Heap(s1)(h_find(s1)(i)))) and
  pc(s1)(i)=7 and
  ...
```

All preconditions are allowed, since we can prove lock-freedom in the following form. In every state $s1$ that satisfies the global invariant, every process q can perform a step, i.e., there is a state $s2$ with $(s1, s2) \in \text{step}$ and $pc(s1, q) \neq pc(s2, q)$. This is expressed in PVS by

```
% theorem for lock-freedom
IV_prog: THEOREM
forall (u: state, q: range(P)) :
  INV(u) => exists (v: state): pc(u)(q) /= pc(v)(q) and step(q,u,v)
```

5 Correctness (Progress)

In this section, we prove that our algorithm is lock-free and almost wait-free. Recall that an algorithm is called *lock-free* if some non-faulty process will finish its task in a finite number of steps, regardless of delays or failures by other processes. This means that no process can block the applications of further operations to the data structure, although any particular operation need not terminate since a slow process can be passed infinitely often by faster processes. An algorithm is called *wait-free* if every process is guaranteed to complete any operation in a finite number of its own steps, regardless of the schedule.

5.1 The easy part of progress

It is clear that *releaseAccess* is wait-free. It follows that the wait-freedom of *migrate* depends on wait-freedom of *moveContents*. If we assume that the choice of i in line 111 is fair, say by round robin, the loop of *moveContents* is bounded. So, wait-freedom of *moveContents* depends on wait-freedom of *moveElement*. It has been proved that n is bounded by m in *moveElement* (see invariants *mE3* and *mE8*). Since, moreover, $\text{toTable}[k] \neq \text{null}$ is stable, the loop of *moveElement* is also bounded. This concludes the sketch that *migrate* is wait-free.

5.2 Progress of *newTable*

The main part of procedure *newTable* is wait-free. This can be shown informally, as follows. Since we can prove the condition $\text{next}(\text{index}) \neq 0$ is stable while process p stays in the region [77, 84], once the condition $\text{next}(\text{index}) \neq 0$ holds, process p will exit *newTable* in a few rounds.

Otherwise, we may assume that p has precondition $\text{next}(\text{index}) = 0$ before executing line 78. By the invariant

$$\text{Ne5: } pc \in [1, 58] \vee pc \geq 62 \wedge pc \neq 65 \wedge \text{next}(\text{index}) = 0 \Rightarrow \text{index} = \text{currInd}$$

we get that $\text{index} = \text{currInd}$ holds and $\text{next}(\text{currInd}) = 0$ from the precondition. We define two sets of integers:

$$\begin{aligned} \text{prSet1}(i) &= \{r \mid \text{index}.r = i \wedge pc.r \notin \{0, 59, 60\}\} \\ \text{prSet2}(i) &= \{r \mid \text{index}.r = i \wedge pc.r \in \{104, 105\} \\ &\quad \vee i_{rA}.r = i \wedge \text{index}.r \neq i \wedge pc.r \in [67, 72] \\ &\quad \vee i_{nT}.r = i \wedge pc.r \in [81, 84] \\ &\quad \vee i_{mig}.r = i \wedge pc.r \geq 97\} \end{aligned}$$

and consider the sum $\sum_{i=1}^{2P} (\#(prSet1(i)) + \#(prSet2(i)))$. While process p is at line 78, the sum cannot exceed $2P - 1$ because there are only P processes around and process p contributes only once to the sum. It then follows from the pigeon hole principle that there exists $j \in [1, 2P]$ such that $\#(prSet1(j)) + \#(prSet2(j)) = 0$ and $j \neq index.p$. By the invariant

$$pr1: \quad prot[j] = \#(prSet1(j)) + \#(prSet2(j)) + \#(currInd = j) + \#(next(currInd) = j)$$

we can get that $prot[j] = 0$ because of $j \neq index.p = currInd$.

While `currInd` is constant, no process can modify `prot[j]` for $j \neq currInd$ infinitely often. Therefore, if process p acts infinitely often and chooses its value i in 78 by round robin, process p exits the loop of `newTable` eventually. This shows that the main part of `newTable` is wait-free.

5.3 The failure of wait-freedom

Procedure `getAccess` is not wait-free. When the active clients keep changing the current index faster than the new client can observe it, the accessing client is doomed to starvation.

It may be possible to make a queue for the accessing clients which is emptied by a process in `newTable`. The accessing clients must however also be able to enter autonomously. This would at least add another layer of complications. We therefore prefer to treat this failure of wait-freedom as a performance issue that can be dealt with in practice by tuning the sizes of the hashtables.

Of course, if the other processes are inactive, `getAccess` only requires constant time. Therefore, `getAccess` is lock-free. It follows that `refresh` and `newTable` are lock-free.

According to the invariants `f5`, `de8`, `in8` and `as6`, the primary procedures `find`, `delete`, `insert`, `assign` are loops bounded by $n \leq h.size$, so they are wait-free unless n is infinitely often reset to 0. This reset only occurs during migration.

Therefore, if we assume that `occ` is not increased too often beyond `bound` in `insert` and `assign`, the primary procedures are wait-free. Under these circumstances, `getAccess` is also wait-free, and then everything is wait-free.

6 Conclusions

Wait-free shared data objects are implemented without any unbounded busy-waiting loops or idle-waiting primitives. They are inherently resilient to halting failures and permit maximum parallelism. We have presented a new practical algorithm, which is almost wait-free, for concurrently accessible hashtables, which promises more robust performance and reliability than a conventional lock-based implementation. Moreover, the new algorithm is dynamic in the sense that it allows the hashtable to grow and shrink as needed.

The algorithm scales up linearly with the number of processes, provided the function `key` and the selection of i in line 111 are defined well. This is confirmed by some experiments where random values were stored, retrieved and deleted from the hashtable. These experiments indicated that 10^6 insertions, deletions and finds per second and per processor are possible on an SGI powerchallenge with 250Mhz R12000 processors. This figure should be taken as a rough indicator, as the performance of parallel processing is very much influenced by the machine architecture, the relative sizes of data structures compared to sizes of caches, and even the scheduling of processes on processors.

The correctness proof for our algorithm is noteworthy because of the extreme effort it took to finish it. Formal deduction by human-guided theorem proving can, in principle, verify any correct design, but doing so may require unreasonable amounts of effort, time, or skill. Though PVS provided great help for managing and reusing the proofs, we have to admit that the verification for our algorithm was very complicated due to the complexity of our algorithm. The total verification effort can roughly be estimated to consist of two man year excluding the effort in determining the algorithm and writing the documentation. The whole proof contains around 200 invariants. It takes an 1Ghz Pentium IV computer around two days to re-run an individual proof for one of the

biggest invariants. Without suitable tool support like PVS, we even doubt if it would be possible to complete a reliable proof of such size and complexity.

Probably, it is possible to simplify the proof and reduce the number of invariants a little bit, but we did not work on this. The complete version of the PVS specifications and the whole proof scripts can be found at [12]. Note that we simplified some definitions in the paper for the sake of presentation.

A Invariants

We present here all invariants whose validity has been verified by the theorem prover PVS.

Conventions. We abbreviate

$$\begin{aligned} \text{Find}(\mathbf{r}, \mathbf{a}) &= \mathbf{r} = \mathbf{null} \vee \mathbf{a} = \text{ADR}(\mathbf{r}) \\ \text{LeastFind}(a, n) &= (\forall m < n : \neg \text{Find}(\mathbf{Y}[\text{key}(a, \text{curSize}, m)], a)) \\ &\quad \wedge \text{Find}(\mathbf{Y}[\text{key}(a, \text{curSize}, n)], a)) \\ \text{LeastFind}(h, a, n) &= (\forall m < n : \neg \text{Find}(h.\mathbf{table}[\text{key}(a, h.\mathbf{size}, m)], a)) \\ &\quad \wedge \text{Find}(h.\mathbf{table}[\text{key}(a, h.\mathbf{size}, n)], a)) \end{aligned}$$

Axioms on functions *key* and *ADR*

$$\begin{aligned} \text{Ax1: } v = \mathbf{null} &\equiv \text{ADR}(v) = \mathbf{0} \\ \text{Ax2: } 0 \leq \text{key}(a, l, k) &< l \\ \text{Ax3: } 0 \leq k < m < l &\Rightarrow \text{key}(a, l, k) \neq \text{key}(a, l, m) \end{aligned}$$

Main correctness properties

$$\begin{aligned} \text{Co1: } pc = 14 &\Rightarrow \text{val}(r_{fi}) = rS_{fi} \\ \text{Co2: } pc \in \{25, 26\} &\Rightarrow \text{suc}_{del} = \text{suc}_{S_{del}} \\ \text{Co3: } pc \in \{41, 42\} &\Rightarrow \text{suc}_{ins} = \text{suc}_{S_{ins}} \\ \text{Cn1: } pc = 14 &\Rightarrow \text{cnt}_{fi} = 1 \\ \text{Cn2: } pc \in \{25, 26\} &\Rightarrow \text{cnt}_{del} = 1 \\ \text{Cn3: } pc \in \{41, 42\} &\Rightarrow \text{cnt}_{ins} = 1 \\ \text{Cn4: } pc = 57 &\Rightarrow \text{cnt}_{ass} = 1 \end{aligned}$$

The absence of memory loss is shown by

$$\begin{aligned} \text{No1: } \sharp(nbSet1) &\leq 2 * P \\ \text{No2: } \sharp(nbSet1) &= \sharp(nbSet2) \end{aligned}$$

where *nbSet1* and *nbSet2* are sets of integers, characterized by

$$\begin{aligned} \text{nbSet1} &= \{k \mid k < \text{H_index} \wedge \text{Heap}(k) \neq \perp\} \\ \text{nbSet2} &= \{i \mid \text{H}(i) \neq 0 \vee (\exists r : pc.r = 71 \wedge i_{rA}.r = i)\} \end{aligned}$$

Further, we have the following definitions of sets of integers:

$$\begin{aligned} \text{deSet1} &= \{k \mid k < \text{curSize} \wedge \mathbf{Y}[k] = \mathbf{del}\} \\ \text{deSet2} &= \{r \mid \text{index}.r = \text{currInd} \wedge pc.r = 25 \wedge \text{suc}_{del}.r\} \\ \text{deSet3} &= \{k \mid k < \text{H}(\text{next}(\text{currInd}).\text{size}) \wedge \text{H}(\text{next}(\text{currInd}).\text{table}[k]) = \mathbf{del}\} \end{aligned}$$

$$\begin{aligned}
ocSet1 &= \{r \mid index.r \neq currInd \\
&\quad \vee pc.r \in [30, 41] \vee pc.r \in [46, 57] \\
&\quad \vee pc.r \in [59, 65] \wedge return_{gA}.r \geq 30 \\
&\quad \vee pc.r \in [67, 72] \\
&\quad \wedge (return_{rA}.r = 59 \wedge return_{gA}.r \geq 30 \\
&\quad \quad \vee return_{rA}.r = 90 \wedge return_{ref}.r \geq 30) \\
&\quad \vee (pc.r = 90 \vee pc.r \in [104, 105]) \wedge return_{ref}.r \geq 30\} \\
ocSet2 &= \{r \mid pc.r \geq 125 \wedge b_{mE}.r \wedge to.r = H(currInd)\} \\
ocSet3 &= \{r \mid index.r = currInd \wedge pc.r = 41 \wedge suc_{ins}.r \\
&\quad \vee index.r = currInd \wedge pc.r = 57 \wedge r_{ass}.r = \text{null}\} \\
ocSet4 &= \{k \mid k < curSize \wedge \text{val}(Y[k]) \neq \text{null}\} \\
ocSet5 &= \{k \mid k < H(next(currInd)).size \\
&\quad \wedge \text{val}(H(next(currInd)).table[k]) \neq \text{null}\} \\
ocSet6 &= \{k \mid k < H(next(currInd)).size \\
&\quad \wedge H(next(currInd)).table[k] \neq \text{null}\} \\
ocSet7 &= \{r \mid pc.r \geq 125 \wedge b_{mE}.r \wedge to.r = H(next(currInd))\}
\end{aligned}$$

$$\begin{aligned}
prSet1(i) &= \{r \mid index.r = i \wedge pc.r \notin \{0, 59, 60\}\} \\
prSet2(i) &= \{r \mid index.r = i \wedge pc.r \in \{104, 105\} \\
&\quad \vee i_{rA}.r = i \wedge index.r \neq i \wedge pc.r \in [67, 72] \\
&\quad \vee i_{nT}.r = i \wedge pc.r \in [81, 84] \\
&\quad \vee i_{mig}.r = i \wedge pc.r \geq 97\} \\
prSet3(i) &= \{r \mid index.r = i \wedge pc.r \in [61, 65] \cup [104, 105] \\
&\quad \vee i_{rA}.r = i \wedge pc.r = 72 \\
&\quad \vee i_{nT}.r = i \wedge pc.r \in [81, 82] \\
&\quad \vee i_{mig}.r = i \wedge pc.r \in [97, 98]\} \\
prSet4(i) &= \{r \mid index.r = i \wedge pc.r \in [61, 65] \\
&\quad \vee i_{mig}.r = i \wedge pc.r \in [97, 98]\} \\
buSet1(i) &= \{r \mid index.r = i \\
&\quad \wedge (pc.r \in [1, 58] \cup (62, 68] \wedge pc.r \neq 65 \\
&\quad \quad \vee pc.r \in [69, 72] \wedge return_{rA}.r > 59 \\
&\quad \quad \vee pc.r > 72)\} \\
buSet2(i) &= \{r \mid index.r = i \wedge pc.r = 104 \\
&\quad \vee i_{rA}.r = i \wedge index.r \neq i \wedge pc.r \in [67, 68] \\
&\quad \vee i_{nT}.r = i \wedge pc.r \in [82, 84] \\
&\quad \vee i_{mig}.r = i \wedge pc.r \geq 100\}
\end{aligned}$$

We have the following invariants concerning the `Heap`

$$\begin{aligned}
He1: & \quad \text{Heap}(0) = \perp \\
He2: & \quad H(i) \neq 0 \equiv \text{Heap}(H(i)) \neq \perp \\
He3: & \quad \text{Heap}(H(currInd)) \neq \perp \\
He4: & \quad pc \in [1, 58] \vee pc > 65 \wedge \neg(pc \in [67, 72] \wedge i_{rA} = index) \\
&\quad \Rightarrow \text{Heap}(H(index)) \neq \perp \\
He5: & \quad \text{Heap}(H(i)) \neq \perp \Rightarrow H(i).size \geq P \\
He6: & \quad \text{next}(currInd) \neq 0 \Rightarrow \text{Heap}(H(next(currInd))) \neq \perp
\end{aligned}$$

Invariants concerning hashtable pointers

$$\begin{aligned}
Ha1: & \quad H_index > 0 \\
Ha2: & \quad H(i) < H_index \\
Ha3: & \quad i \neq j \wedge \text{Heap}(H(i)) \neq \perp \Rightarrow H(i) \neq H(j)
\end{aligned}$$

$$Ha4: \quad index \neq currInd \Rightarrow H(index) \neq H(currInd)$$

Invariants about counters for calling the specification.

$$Cn5: \quad pc \in [6, 7] \Rightarrow cnt_{fi} = 0$$

$$Cn6: \quad pc \in [8, 13]$$

$$\vee pc \in [59, 65] \wedge return_{gA} = 10$$

$$\vee pc \in [67, 72] \wedge (return_{rA} = 59 \wedge return_{gA} = 10$$

$$\vee return_{rA} = 90 \wedge return_{ref} = 10$$

$$\vee pc \geq 90 \wedge return_{ref} = 10$$

$$\Rightarrow cnt_{fi} = \sharp(r_{fi} = \text{null} \vee a_{fi} = ADR(r_{fi}))$$

$$Cn7: \quad pc \in [16, 21] \wedge pc \neq 18$$

$$\vee pc \in [59, 65] \wedge return_{gA} = 20$$

$$\vee pc \in [67, 72] \wedge (return_{rA} = 59 \wedge return_{gA} = 20$$

$$\vee return_{rA} = 90 \wedge return_{ref} = 20$$

$$\vee pc \geq 90 \wedge return_{ref} = 20$$

$$\Rightarrow cnt_{del} = 0$$

$$Cn8: \quad pc = 18 \Rightarrow cnt_{del} = \sharp(r_{del} = \text{null})$$

$$Cn9: \quad pc \in [28, 33]$$

$$\vee pc \in [59, 65] \wedge return_{gA} = 30$$

$$\vee pc \in [67, 72] \wedge (return_{rA} = 59 \wedge return_{gA} = 30$$

$$\vee return_{rA} = 77 \wedge return_{nT} = 30$$

$$\vee return_{rA} = 90 \wedge return_{ref} = 30$$

$$\vee pc \in [77, 84] \wedge return_{nT} = 30$$

$$\vee pc \geq 90 \wedge return_{ref} = 30$$

$$\Rightarrow cnt_{ins} = 0$$

$$Cn10: \quad pc \in [35, 37]$$

$$\vee pc \in [59, 65] \wedge return_{gA} = 36$$

$$\vee pc \in [67, 72] \wedge (return_{rA} = 59 \wedge return_{gA} = 36$$

$$\vee return_{rA} = 90 \wedge return_{ref} = 36$$

$$\vee pc \geq 90 \wedge return_{ref} = 36$$

$$\Rightarrow cnt_{ins} = \sharp(a_{ins} = ADR(r_{ins}) \vee suc_{ins})$$

$$Cn11: \quad pc \in [44, 52]$$

$$\vee pc \in [59, 65] \wedge return_{gA} \in \{46, 51\}$$

$$\vee pc \in [67, 72] \wedge (return_{rA} = 59 \wedge return_{gA} \in \{46, 51\}$$

$$\vee return_{rA} = 77 \wedge return_{nT} = 46$$

$$\vee return_{rA} = 90 \wedge return_{ref} \in \{46, 51\}$$

$$\vee pc \in [77, 84] \wedge return_{nT} = 46$$

$$\vee pc \geq 90 \wedge return_{ref} \in \{46, 51\}$$

$$\Rightarrow cnt_{ass}sign = 0$$

Invariants about old hashtables, current hashtable and the auxiliary hashtable Y . Here, we universally quantify over all non-negative integers $n < curSize$.

$$Cu1: \quad H(index) \neq H(currInd) \wedge k < H(index).\text{size}$$

$$\wedge (pc \in [1, 58] \vee pc > 65 \wedge \neg(pc \in [67, 72] \wedge i_{rA} = index))$$

$$\Rightarrow H(index).\text{table}[k] = \text{done}$$

- Cu2: $\#\{k \mid k < \text{curSize} \wedge Y[k] \neq \text{null}\} < \text{curSize}$
 Cu3: $H(\text{currInd}).\text{bound} + 2 * P < \text{curSize}$
 Cu4: $H(\text{currInd}).\text{dels} + \#\text{(deSet2)} = \#\text{(deSet1)}$
 Cu5: Cu5 has been eliminated. The numbering has been kept, so as not to endanger the consistency with Appendix B and the PVS script.
 Cu6: $H(\text{currInd}).\text{occ} + \#\text{(ocSet1)} + \#\text{(ocSet2)} \leq H(\text{currInd}).\text{bound} + 2 * P$
 Cu7: $\#\{k \mid k < \text{curSize} \wedge Y[k] \neq \text{null}\} = H(\text{currInd}).\text{occ} + \#\text{(ocSet2)} + \#\text{(ocSet3)}$
 Cu8: $\text{next}(\text{currInd}) = 0 \Rightarrow \neg \text{oldp}(H(\text{currInd}).\text{table}[n])$
 Cu9: $\neg \text{oldp}(H(\text{currInd}).\text{table}[n]) \Rightarrow H(\text{currInd}).\text{table}[n] = Y[n]$
 Cu10: $\text{oldp}(H(\text{currInd}).\text{table}[n]) \wedge \text{val}(H(\text{currInd}).\text{table}[n]) \neq \text{null}$
 $\Rightarrow \text{val}(H(\text{currInd}).\text{table}[n]) = \text{val}(Y[n])$
 Cu11: $\text{LeastFind}(a, n) \Rightarrow X(a) = \text{val}(Y[\text{key}(a, \text{curSize}, n)])$
 Cu12: $X(a) = \text{val}(Y[\text{key}(a, \text{curSize}, n)]) \neq \text{null} \Rightarrow \text{LeastFind}(a, n)$
 Cu13: $X(a) = \text{val}(Y[\text{key}(a, \text{curSize}, n)]) \neq \text{null} \wedge n \neq m < \text{curSize}$
 $\Rightarrow \text{ADR}(Y[\text{key}(a, \text{curSize}, m)]) \neq a$
 Cu14: $X(a) = \text{null} \wedge \text{val}(Y[\text{key}(a, \text{curSize}, n)]) \neq \text{null}$
 $\Rightarrow \text{ADR}(Y[\text{key}(a, \text{curSize}, n)]) \neq a$
 Cu15: $X(a) \neq \text{null}$
 $\Rightarrow \exists m < \text{curSize} : X(a) = \text{val}(Y[\text{key}(a, \text{curSize}, m)])$
 Cu16: $\exists(f : \{m : 0 \leq m < \text{curSize} \wedge \text{val}(Y[m]) \neq \text{null}\} \rightarrow$
 $\{v : v \neq \text{null} \wedge (\exists k < \text{curSize} : v = \text{val}(Y[k]))\}) :$
 $f \text{ is bijective}$

Invariants about `next` and `next(currInd)`:

- Ne1: $\text{currInd} \neq \text{next}(\text{currInd})$
 Ne2: $\text{next}(\text{currInd}) \neq 0 \Rightarrow \text{next}(\text{next}(\text{currInd})) = 0$
 Ne3: $pc \in [1, 59] \vee pc \geq 62 \wedge pc \neq 65 \Rightarrow \text{index} \neq \text{next}(\text{currInd})$
 Ne4: $pc \in [1, 58] \vee pc \geq 62 \wedge pc \neq 65 \Rightarrow \text{index} \neq \text{next}(\text{index})$
 Ne5: $pc \in [1, 58] \vee pc \geq 62 \wedge pc \neq 65 \wedge \text{next}(\text{index}) = 0 \Rightarrow \text{index} = \text{currInd}$
 Ne6: $\text{next}(\text{currInd}) \neq 0$
 $\Rightarrow \#\text{(ocSet6)} \leq \#\{k \mid k < \text{curSize} \wedge Y[k] \neq \text{null}\} - H(\text{currInd}).\text{dels} - \#\text{(deSet2)}$
 Ne7: $\text{next}(\text{currInd}) \neq 0$
 $\Rightarrow H(\text{currInd}).\text{bound} - H(\text{currInd}).\text{dels} + 2 * P \leq H(\text{next}(\text{currInd})).\text{bound}$
 Ne8: $\text{next}(\text{currInd}) \neq 0$
 $\Rightarrow H(\text{next}(\text{currInd})).\text{bound} + 2 * P < H(\text{next}(\text{currInd})).\text{size}$
 Ne9: $\text{next}(\text{currInd}) \neq 0 \Rightarrow H(\text{next}(\text{currInd})).\text{dels} = \#\text{(deSet3)}$
 Ne9a: $\text{next}(\text{currInd}) \neq 0 \Rightarrow H(\text{next}(\text{currInd})).\text{dels} = 0$

 Ne10: $\text{next}(\text{currInd}) \neq 0 \wedge k < h.\text{size} \Rightarrow h.\text{table}[k] \notin \{\text{del}, \text{done}\},$
 where $h = H(\text{next}(\text{currInd}))$
 Ne11: $\text{next}(\text{currInd}) \neq 0 \wedge k < H(\text{next}(\text{currInd})).\text{size}$
 $\Rightarrow \neg \text{oldp}(H(\text{next}(\text{currInd})).\text{table}[k])$
 Ne12: $k < \text{curSize} \wedge H(\text{currInd}).\text{table}[k] = \text{done} \wedge m < h.\text{size} \wedge \text{LeastFind}(h, a, m)$
 $\Rightarrow X(a) = \text{val}(h.\text{table}[\text{key}(a, h.\text{size}, m)]),$
 where $a = \text{ADR}(Y[k])$ and $h = H(\text{next}(\text{currInd}))$
 Ne13: $k < \text{curSize} \wedge H(\text{currInd}).\text{table}[k] = \text{done} \wedge m < h.\text{size}$
 $\wedge X(a) = \text{val}(h.\text{table}[\text{key}(a, h.\text{size}, m)]) \neq \text{null}$
 $\Rightarrow \text{LeastFind}(h, a, m),$
 where $a = \text{ADR}(Y[k])$ and $h = H(\text{next}(\text{currInd}))$
 Ne14: $\text{next}(\text{currInd}) \neq 0 \wedge a \neq 0 \wedge k < h.\text{size}$

$\wedge \text{X}(a) = \text{val}(h.\text{table}[\text{key}(a, h.\text{size}, k)]) \neq \text{null}$
 $\Rightarrow \text{LeastFind}(h, a, k),$
 where $h = \text{H}(\text{next}(\text{currInd}))$
 Ne15: $k < \text{curSize} \wedge \text{H}(\text{currInd}).\text{table}[k] = \text{done} \wedge \text{X}(a) \neq \text{null} \wedge m < h.\text{size}$
 $\wedge \text{X}(a) = \text{val}(h.\text{table}[\text{key}(a, h.\text{size}, m)]) \wedge n < h.\text{size} \wedge m \neq n$
 $\Rightarrow \text{ADR}(h.\text{table}[\text{key}(a, h.\text{size}, n)]) \neq a,$
 where $a = \text{ADR}(Y[k])$ and $h = \text{H}(\text{next}(\text{currInd}))$
 Ne16: $k < \text{curSize} \wedge \text{H}(\text{currInd}).\text{table}[k] = \text{done} \wedge \text{X}(a) = \text{null} \wedge m < h.\text{size}$
 $\Rightarrow \text{val}(h.\text{table}[\text{key}(a, h.\text{size}, m)]) = \text{null}$
 $\vee \text{ADR}(h.\text{table}[\text{key}(a, h.\text{size}, m)]) \neq a,$
 where $a = \text{ADR}(Y[k])$ and $h = \text{H}(\text{next}(\text{currInd}))$
 Ne17: $\text{next}(\text{currInd}) \neq 0 \wedge m < h.\text{size} \wedge a = \text{ADR}(h.\text{table}[m]) \neq 0$
 $\Rightarrow \text{X}(a) = \text{val}(h.\text{table}[m]) \neq \text{null},$
 where $h = \text{H}(\text{next}(\text{currInd}))$
 Ne18: $\text{next}(\text{currInd}) \neq 0 \wedge m < h.\text{size} \wedge a = \text{ADR}(h.\text{table}[m]) \neq 0$
 $\Rightarrow \exists n < \text{curSize} : \text{val}(Y[n]) = \text{val}(h.\text{table}[m]) \wedge \text{oldp}(\text{H}(\text{currInd}).\text{table}[n]),$
 where $h = \text{H}(\text{next}(\text{currInd}))$
 Ne19: $\text{next}(\text{currInd}) \neq 0 \wedge m < h.\text{size} \wedge a = \text{ADR}(h.\text{table}[\text{key}(a, h.\text{size}, m)]) \neq 0$
 $\wedge m \neq n < h.\text{size}$
 $\Rightarrow \text{ADR}(h.\text{table}[\text{key}(a, h.\text{size}, n)]) \neq a,$
 where $h = \text{H}(\text{next}(\text{currInd}))$
 Ne20: $k < \text{curSize} \wedge \text{H}(\text{currInd}).\text{table}[k] = \text{done} \wedge \text{X}(a) \neq \text{null}$
 $\Rightarrow \exists m < h.\text{size} : \text{X}(a) = \text{val}(h.\text{table}[\text{key}(a, h.\text{size}, m)]),$
 where $a = \text{ADR}(Y[k])$ and $h = \text{H}(\text{next}(\text{currInd}))$
 Ne21: Ne21 has been eliminated.
 Ne22: $\text{next}(\text{currInd}) \neq 0 \Rightarrow \#\text{(ocSet6)} = \text{H}(\text{next}(\text{currInd})).\text{occ} + \#\text{(ocSet7)}$
 Ne23: $\text{next}(\text{currInd}) \neq 0 \Rightarrow \text{H}(\text{next}(\text{currInd})).\text{occ} \leq \text{H}(\text{next}(\text{currInd})).\text{bound}$
 Ne24: $\text{next}(\text{currInd}) \neq 0 \Rightarrow \#\text{(ocSet5)} \leq \#\text{(ocSet4)}$
 Ne25: $\text{next}(\text{currInd}) \neq 0$
 $\Rightarrow \exists(f : \{m : 0 \leq m < h.\text{size} \wedge \text{val}(h.\text{table}[m]) \neq \text{null}\} \rightarrow$
 $\{v : v \neq \text{null} \wedge (\exists k < h.\text{size} : v = \text{val}(h.\text{table}[k]))\}) :$
 $f \text{ is bijective},$
 where $h = \text{H}(\text{next}(\text{currInd}))$
 Ne26: $\text{next}(\text{currInd}) \neq 0$
 $\Rightarrow \exists(f : \{v : v \neq \text{null} \wedge (\exists m < h.\text{size} : v = \text{val}(h.\text{table}[m]))\} \rightarrow$
 $\{v : v \neq \text{null} \wedge (\exists k < \text{curSize} : v = \text{val}(Y[k]))\}) :$
 $f \text{ is injective},$
 where $h = \text{H}(\text{next}(\text{currInd}))$
 Ne27: $\text{next}(\text{currInd}) \neq 0 \wedge (\exists n < h.\text{size} : \text{val}(h.\text{table}[n]) \neq \text{null})$
 $\Rightarrow \exists(f : \{m : 0 \leq m < h.\text{size} \wedge \text{val}(h.\text{table}[m]) \neq \text{null}\} \rightarrow$
 $\{k : 0 \leq k < \text{curSize} \wedge \text{val}(Y[k]) \neq \text{null}\})$
 $f \text{ is injective},$
 where $h = \text{H}(\text{next}(\text{currInd}))$

Invariants concerning procedure *find* (5...14)

- f1: $a_{fi} \neq \mathbf{0}$
- f2: $pc \in \{6, 11\} \Rightarrow n_{fi} = 0$
- f3: $pc \in \{7, 8, 13\} \Rightarrow l_{fi} = h_{fi}.\text{size}$
- f4: $pc \in [6, 13] \wedge pc \neq 10 \Rightarrow h_{fi} = \text{H}(\text{index})$
- f5: $pc = 7 \wedge h_{fi} = \text{H}(\text{currInd}) \Rightarrow n_{fi} < \text{curSize}$
- f6: $pc = 8 \wedge h_{fi} = \text{H}(\text{currInd}) \wedge \neg \text{Find}(r_{fi}, a_{fi}) \wedge r_{fi} \neq \text{done}$
 $\Rightarrow \neg \text{Find}(Y[\text{key}(a_{fi}, \text{curSize}, n_{fi})], a_{fi})$
- f7: $pc = 13 \wedge h_{fi} = \text{H}(\text{currInd}) \wedge \neg \text{Find}(r_{fi}, a_{fi}) \wedge m < n_{fi}$

f8: $\Rightarrow \neg \text{Find}(\text{Y}[\text{key}(a_{fi}, \text{curSize}, m)], a_{fi})$
 $pc \in \{7, 8\} \wedge h_{fi} = \text{H}(\text{currInd}) \wedge m < n_{fi}$
 $\Rightarrow \neg \text{Find}(\text{Y}[\text{key}(a_{fi}, \text{curSize}, m)], a_{fi})$
 f9: $pc = 7 \wedge \text{Find}(t, a_{fi}) \Rightarrow \text{X}(a_{fi}) = \text{val}(t),$
 $\text{where } t = h_{fi}.\text{table}[\text{key}(a_{fi}, l_{fi}, n_{fi})]$
 f10: $pc \notin \{1, 7\} \wedge \text{Find}(r_{fi}, a_{fi}) \Rightarrow \text{val}(r_{fi}) = rS_{fi}$
 f11: $pc = 8 \wedge \text{oldp}(r_{fi}) \wedge \text{index} = \text{currInd}$
 $\Rightarrow \text{next}(\text{currInd}) \neq 0$

Invariants concerning procedure *delete* (15...26)

de1: $a_{del} \neq \mathbf{0}$
 de2: $pc \in \{17, 18\} \Rightarrow l_{del} = h_{del}.\text{size}$
 de3: $pc \in [16, 25] \wedge pc \neq 20 \Rightarrow h_{del} = \text{H}(\text{index})$
 de4: $pc = 18 \Rightarrow k_{del} = \text{key}(a_{del}, l_{del}, n_{del})$
 de5: $pc \in \{16, 17\} \vee \text{Deleting} \Rightarrow \neg \text{suc}_{del}$
 de6: $\text{Deleting} \wedge \text{sucS}_{del} \Rightarrow r_{del} \neq \mathbf{null}$
 de7: $pc = 18 \wedge \neg \text{oldp}(h_{del}.\text{table}[k_{del}]) \Rightarrow h_{del} = \text{H}(\text{currInd})$
 de8: $pc \in \{17, 18\} \wedge h_{del} = \text{H}(\text{currInd}) \Rightarrow n_{del} < \text{curSize}$
 de9: $pc = 18 \wedge h_{del} = \text{H}(\text{currInd})$
 $\wedge (\text{val}(r_{del}) \neq \mathbf{null} \vee r_{del} = \mathbf{del})$
 $\Rightarrow r \neq \mathbf{null} \wedge (r = \mathbf{del} \vee \text{ADR}(r) = \text{ADR}(r_{del})),$
 $\text{where } r = \text{Y}[\text{key}(a_{del}, h_{del}.\text{size}, n_{del})]$
 de10: $pc \in \{17, 18\} \wedge h_{del} = \text{H}(\text{currInd}) \wedge m < n_{del}$
 $\Rightarrow \neg \text{Find}(\text{Y}[\text{key}(a_{del}, \text{curSize}, m)], a_{del})$
 de11: $pc \in \{17, 18\} \wedge \text{Find}(t, a_{del}) \Rightarrow \text{X}(a_{del}) = \text{val}(t),$
 $\text{where } t = h_{del}.\text{table}[\text{key}(a_{del}, l_{del}, n_{del})]$
 de12: $pc = 18 \wedge \text{oldp}(r_{del}) \wedge \text{index} = \text{currInd}$
 $\Rightarrow \text{next}(\text{currInd}) \neq 0$
 de13: $pc = 18 \Rightarrow k_{del} < \text{H}(\text{index}).\text{size}$

where *Deleting* is characterized by

$\text{Deleting} \equiv$
 $pc \in [18, 21] \vee pc \in [59, 65] \wedge \text{return}_{gA} = 20$
 $\vee pc \in [67, 72] \wedge (\text{return}_{rA} = 59 \wedge \text{return}_{gA} = 20$
 $\quad \vee \text{return}_{rA} = 90 \wedge \text{return}_{ref} = 20)$
 $\vee pc \geq 90 \wedge \text{return}_{ref} = 20$

Invariants concerning procedure *insert* (27...52)

in1: $a_{ins} = \text{ADR}(v_{ins}) \wedge v_{ins} \neq \mathbf{null}$
 in2: $pc \in [32, 35] \Rightarrow l_{ins} = h_{ins}.\text{size}$
 in3: $pc \in [28, 41] \wedge pc \notin \{30, 36\} \Rightarrow h_{ins} = \text{H}(\text{index})$
 in4: $pc \in \{33, 35\} \Rightarrow k_{ins} = \text{key}(a_{ins}, l_{ins}, n_{ins})$
 in5: $pc \in [32, 33] \vee \text{Inserting} \Rightarrow \neg \text{suc}_{ins}$
 in6: $\text{Inserting} \wedge \text{sucS}_{ins} \Rightarrow \text{ADR}(r_{ins}) \neq a_{ins}$
 in7: $pc = 35 \wedge \neg \text{oldp}(h_{ins}.\text{table}[k_{ins}]) \Rightarrow h_{ins} = \text{H}(\text{currInd})$
 in8: $pc \in \{33, 35\} \wedge h_{ins} = \text{H}(\text{currInd}) \Rightarrow n_{ins} < \text{curSize}$
 in9: $pc = 35 \wedge h_{ins} = \text{H}(\text{currInd})$
 $\wedge (\text{val}(r_{ins}) \neq \mathbf{null} \vee r_{ins} = \mathbf{del})$
 $\Rightarrow r \neq \mathbf{null} \wedge (r = \mathbf{del} \vee \text{ADR}(r) = \text{ADR}(r_{ins})),$
 $\text{where } r = \text{Y}[\text{key}(a_{ins}, h_{ins}.\text{size}, n_{ins})]$

- in10: $pc \in \{32, 33, 35\} \wedge h_{ins} = H(\text{currInd}) \wedge m < n_{ins}$
 $\Rightarrow \neg \text{Find}(Y[\text{key}(a_{ins}, \text{curSize}, m)], a_{ins})$
- in11: $pc \in \{33, 35\} \wedge \text{Find}(t, a_{ins}) \Rightarrow X(a_{ins}) = \text{val}(t),$
 where $t = h_{ins}.\text{table}[\text{key}(a_{ins}, l_{ins}, n_{ins})]$
- in12: $pc = 35 \wedge \text{oldp}(r_{ins}) \wedge \text{index} = \text{currInd}$
 $\Rightarrow \text{next}(\text{currInd}) \neq 0$
- in13: $pc = 35 \Rightarrow k_{ins} < H(\text{index}).\text{size}$

where *Inserting* is characterized by

$$\begin{aligned} \text{Inserting} \equiv & \\ & pc \in [35, 37] \vee pc \in [59, 65] \wedge \text{return}_{gA} = 36 \\ & \vee pc \in [67, 72] \wedge (\text{return}_{rA} = 59 \wedge \text{return}_{gA} = 36) \\ & \quad \vee \text{return}_{rA} = 90 \wedge \text{return}_{ref} = 36) \\ & \vee pc \geq 90 \wedge \text{return}_{ref} = 36 \end{aligned}$$

Invariants concerning procedure *assign* (43...57)

- as1: $a_{ass} = ADR(v_{ass}) \wedge v_{ass} \neq \text{null}$
- as2: $pc \in [48, 50] \Rightarrow l_{ass} = h_{ass}.\text{size}$
- as3: $pc \in [44, 57] \wedge pc \notin \{46, 51\} \Rightarrow h_{ass} = H(\text{index})$
- as4: $pc \in \{49, 50\} \Rightarrow k_{ass} = \text{key}(a_{ass}, l_{ass}, n_{ass})$
- as5: $pc = 50 \wedge \neg \text{oldp}(h_{ass}.\text{table}[k_{ass}]) \Rightarrow h_{ass} = H(\text{currInd})$
- as6: $pc = 50 \wedge h_{ass} = H(\text{currInd}) \Rightarrow n_{ass} < \text{curSize}$
- as7: $pc = 50 \wedge h_{ass} = H(\text{currInd})$
 $\wedge (\text{val}(r_{ass}) \neq \text{null} \vee r_{ass} = \text{del})$
 $\Rightarrow r \neq \text{null} \wedge (r = \text{del} \vee ADR(r) = ADR(r_{ass})),$
 where $r = Y[\text{key}(a_{ass}, h_{ass}.\text{size}, n_{ass})]$
- as8: $pc \in \{48, 49, 50\} \wedge h_{ass} = H(\text{currInd}) \wedge m < n_{ass}$
 $\Rightarrow \neg \text{Find}(Y[\text{key}(a_{ass}, \text{curSize}, m)], a_{ass})$
- as9: $pc = 50 \wedge \text{Find}(t, a_{ass}) \Rightarrow X(a_{ass}) = \text{val}(t),$
 where $t = h_{ass}.\text{table}[\text{key}(a_{ass}, l_{ass}, n_{ass})]$
- as10: $pc = 50 \wedge \text{oldp}(r_{ass}.\text{sign}) \wedge \text{index} = \text{currInd}$
 $\Rightarrow \text{next}(\text{currInd}) \neq 0$
- as11: $pc = 50 \Rightarrow k_{ass} < H(\text{index}).\text{size}$

Invariants concerning procedure *releaseAccess* (67...72)

- rA1: $h_{rA} < H_index$
- rA2: $pc \in [70, 71] \Rightarrow h_{rA} \neq 0$
- rA3: $pc = 71 \Rightarrow \text{Heap}(h_{rA}) \neq \perp$
- rA4: $pc = 71 \Rightarrow H(i_{rA}) = 0$
- rA5: $pc = 71 \Rightarrow h_{rA} \neq H(i)$
- rA6: $pc = 70 \Rightarrow H(i_{rA}) \neq H(\text{currInd})$
- rA7: $pc = 70$
 $\wedge (pc.r \in [1, 58] \vee pc.r > 65 \wedge \neg (pc.r \in [67, 72] \wedge i_{rA}.r = \text{index}.r))$
 $\Rightarrow H(i_{rA}) \neq H(\text{index}.r)$
- rA8: $pc = 70 \Rightarrow i_{rA} \neq \text{next}(\text{currInd})$
- rA9: $pc \in [68, 72] \wedge (h_{rA} = 0 \vee h_{rA} \neq H(i_{rA}))$
 $\Rightarrow H(i_{rA}) = 0$
- rA10: $pc \in [67, 72] \wedge \text{return}_{rA} \in \{0, 59\} \Rightarrow i_{rA} = \text{index}$
- rA11: $pc \in [67, 72] \wedge \text{return}_{rA} \in \{77, 90\} \Rightarrow i_{rA} \neq \text{index}$
- rA12: $pc \in [67, 72] \wedge \text{return}_{rA} = 77 \Rightarrow \text{next}(\text{index}) \neq 0$

- rA13: $pc = 71 \wedge pc.r = 71 \wedge p \neq r \Rightarrow h_{rA} \neq h_{rA.r}$
 rA14: $pc = 71 \wedge pc.r = 71 \wedge p \neq r \Rightarrow i_{rA} \neq i_{rA.r}$

Invariants concerning procedure *newTable* (77...84)

- nT1: $pc \in [81, 82] \Rightarrow \text{Heap}(H(i_{nT})) = \perp$
 nT2: $pc \in [83, 84] \Rightarrow \text{Heap}(H(i_{nT})) \neq \perp$
 nT3: $pc = 84 \Rightarrow \text{next}(i_{nT}) = 0$
 nT4: $pc \in [83, 84] \Rightarrow H(i_{nT}).\text{dels} = 0$
 nT5: $pc \in [83, 84] \Rightarrow H(i_{nT}).\text{occ} = 0$
 nT6: $pc \in [83, 84] \Rightarrow H(i_{nT}).\text{bound} + 2 * P < H(i_{nT}).\text{size}$
 nT7: $pc \in [83, 84] \wedge \text{index} = \text{currInd}$
 $\Rightarrow H(\text{currInd}).\text{bound} - H(\text{currInd}).\text{dels} + 2 * P < H(i_{nT}).\text{bound}$
 nT8: $pc \in [83, 84] \wedge k < H(i_{nT}).\text{size} \Rightarrow H(i_{nT}).\text{table}[k] = \text{null}$
 nT9: $pc \in [81, 84] \Rightarrow i_{nT} \neq \text{currInd}$
 nT10: $pc \in [81, 84] \wedge (pc.r \in [1, 58] \vee pc.r \geq 62 \wedge pc.r \neq 65)$
 $\Rightarrow i_{nT} \neq \text{index.r}$
 nT11: $pc \in [81, 84] \Rightarrow i_{nT} \neq \text{next}(\text{currInd})$
 nT12: $pc \in [81, 84] \Rightarrow H(i_{nT}) \neq H(\text{currInd})$
 nT13: $pc \in [81, 84]$
 $\wedge (pc.r \in [1, 58] \vee pc.r > 65 \wedge \neg (pc.r \in [67, 72] \wedge i_{rA.r} = \text{index.r}))$
 $\Rightarrow H(i_{nT}) \neq H(\text{index.r})$
 nT14: $pc \in [81, 84] \wedge pc.r \in [67, 72] \Rightarrow i_{nT} \neq i_{rA.r}$
 nT15: $pc \in [83, 84] \wedge pc.r \in [67, 72] \Rightarrow H(i_{nT}) \neq H(i_{rA.r})$
 nT16: $pc \in [81, 84] \wedge pc.r \in [81, 84] \wedge p \neq r \Rightarrow i_{nT} \neq i_{nT.r}$
 nT17: $pc \in [81, 84] \wedge pc.r \in [95, 99] \wedge \text{index.r} = \text{currInd}$
 $\Rightarrow i_{nT} \neq i_{mig.r}$
 nT18: $pc \in [81, 84] \wedge pc.r \geq 99 \Rightarrow i_{nT} \neq i_{mig.r}$

Invariants concerning procedure *migrate* (94...105)

- mi1: $pc = 98 \vee pc \in \{104, 105\} \Rightarrow \text{index} \neq \text{currInd}$
 mi2: $pc \geq 95 \Rightarrow i_{mig} \neq \text{index}$
 mi3: $pc = 94 \Rightarrow \text{next}(\text{index}) > 0$
 mi4: $pc \geq 95 \Rightarrow i_{mig} \neq 0$
 mi5: $pc \geq 95 \Rightarrow i_{mig} = \text{next}(\text{index})$
 mi6: $pc.r = 70$
 $\wedge (pc \in [95, 102] \wedge \text{index} = \text{currInd} \vee pc \in [102, 103] \vee pc \geq 110)$
 $\Rightarrow i_{rA.r} \neq i_{mig}$
 mi7: $pc \in [95, 97] \wedge \text{index} = \text{currInd} \vee pc \geq 99$
 $\Rightarrow i_{mig} \neq \text{next}(i_{mig})$
 mi8: $(pc \in [95, 97] \vee pc \in [99, 103] \vee pc \geq 110) \wedge \text{index} = \text{currInd}$
 $\Rightarrow \text{next}(i_{mig}) = 0$
 mi9: $(pc \in [95, 103] \vee pc \geq 110) \wedge \text{index} = \text{currInd}$
 $\Rightarrow H(i_{mig}) \neq H(\text{currInd})$
 mi10: $(pc \in [95, 103] \vee pc \geq 110) \wedge \text{index} = \text{currInd}$
 $\wedge (pc.r \in [1, 58] \vee pc.r \geq 62 \wedge pc.r \neq 65)$
 $\Rightarrow H(i_{mig}) \neq H(\text{index.r})$
 mi11: $pc = 101 \wedge \text{index} = \text{currInd} \vee pc = 102$
 $\Rightarrow h_{mig} = H(i_{mig})$
 mi12: $pc \geq 95 \wedge \text{index} = \text{currInd} \vee pc \in \{102, 103\} \vee pc \geq 110$
 $\Rightarrow \text{Heap}(H(i_{mig})) \neq \perp$
 mi13: $pc = 103 \wedge \text{index} = \text{currInd} \wedge k < \text{curSize} \Rightarrow H(\text{index}).\text{table}[k] = \text{done}$

- $mi14:$ $pc = 103 \wedge index = currInd \wedge n < H(i_{mig}).size$
 $\wedge LeastFind(H(i_{mig}), a, n)$
 $\Rightarrow X(a) = val(H(i_{mig})[key(a, H(i_{mig}).size, n)])$
- $mi15:$ $pc = 103 \wedge index = currInd \wedge n < H(i_{mig}).size$
 $\wedge X(a) = val(H(i_{mig}).table[key(a, H(i_{mig}).size, n)]) \neq null$
 $\Rightarrow LeastFind(H(i_{mig}), a, n)$
- $mi16:$ $pc = 103 \wedge index = currInd \wedge k < H(i_{mig}).size$
 $\Rightarrow \neg oldp(H(i_{mig}).table[k])$
- $mi17:$ $pc = 103 \wedge index = currInd \wedge X(a) \neq null \wedge k < h.size$
 $\wedge X(a) = val(h.table[key(a, h.size, k)]) \wedge k \neq n < h.size$
 $\Rightarrow ADR(h.table[key(a, h.size, n)]) \neq a,$
 where $h = H(i_{mig})$
- $mi18:$ $pc = 103 \wedge index = currInd \wedge X(a) = null \wedge k < h.size$
 $\Rightarrow val(h.table[key(a, h.size, k)]) = null$
 $\vee ADR(h.table[key(a, h.size, k)]) \neq a,$
 where $h = H(i_{mig})$
- $mi19:$ $pc = 103 \wedge index = currInd \wedge X(a) \neq null$
 $\Rightarrow \exists m < h.size : X(a) = val(h.table[key(a, h.size, m)]),$
 where $h = H(i_{mig})$
- $mi20:$ $pc = 117 \wedge X(a) \neq null \wedge val(H(index).table[i_{mC}]) \neq null$
 $\vee pc \geq 126 \wedge X(a) \neq null \wedge index = currInd$
 $\vee pc = 125 \wedge X(a) \neq null \wedge index = currInd$
 $\wedge (b_{mE} \vee val(w_{mE}) \neq null)$
 $\wedge a_{mE} = ADR(w_{mE}))$
 $\Rightarrow \exists m < h.size : X(a) = val(h.table[key(a, h.size, m)]),$
 where $a = ADR(Y[i_{mC}])$ and $h = H(next(currInd))$

Invariants concerning procedure *moveContents* (110...118):

- $mC1:$ $pc = 103 \vee pc \geq 110 \Rightarrow to = H(i_{mig})$
- $mC2:$ $pc \geq 110 \Rightarrow from = H(index)$
- $mC3:$ $pc > 102 \wedge m \in toBeMoved \Rightarrow m < H(index).size$
- $mC4:$ $pc = 111 \Rightarrow \exists m < from.size : m \in toBeMoved$
- $mC5:$ $pc \geq 114 \wedge pc \neq 118 \Rightarrow v_{mC} \neq done$
- $mC6:$ $pc \geq 114 \Rightarrow i_{mC} < H(index).size$
- $mC7:$ $pc = 118 \Rightarrow H(index).table[i_{mC}] = done$
- $mC8:$ $pc \geq 110 \wedge k < H(index).size \wedge k \notin toBeMoved$
 $\Rightarrow H(index).table[k] = done$
- $mC9:$ $pc \geq 110 \wedge index = currInd \wedge toBeMoved = \emptyset \wedge k < H(index).size$
 $\Rightarrow H(index).table[k] = done$
- $mC10:$ $pc \geq 116 \wedge val(v_{mC}) \neq null$
 $\wedge H(index).table[i_{mC}] = done$
 $\Rightarrow H(i_{mig}).table[key(a, H(i_{mig}).size, 0)] \neq null,$
 where $a = ADR(v_{mC})$
- $mC11:$ $pc \geq 116 \wedge H(index).table[i_{mC}] \neq done$
 $\Rightarrow val(v_{mC}) = val(H(index).table[i_{mC}])$
 $\wedge \neg oldp(H(index).table[i_{mC}])$
- $mC12:$ $pc \geq 116 \wedge index = currInd \wedge val(v_{mC}) \neq null$
 $\Rightarrow val(v_{mC}) = val(Y[i_{mC}])$

Invariants concerning procedure *moveElement* (120...126):

- $mE1:$ $pc \geq 120 \Rightarrow val(v_{mC}) = v_{mE}$

mE2: $pc \geq 120 \Rightarrow v_{mE} \neq \mathbf{null}$
 mE3: $pc \geq 120 \Rightarrow to = H(i_{mig})$
 mE4: $pc \geq 121 \Rightarrow a_{mE} = ADR(v_{mC})$
 mE5: $pc \geq 121 \Rightarrow m_{mE} = to.size$
 mE6: $pc \in \{121, 123\} \Rightarrow \neg b_{mE}$
 mE7: $pc = 123 \Rightarrow k_{mE} = key(a_{mE}, to.size, n_{mE})$
 mE8: $pc \geq 123 \Rightarrow k_{mE} < H(i_{mig}).size$
 mE9: $pc = 120$
 $\wedge to.table[key(ADR(v_{mE}), to.size, 0)] = \mathbf{null}$
 $\Rightarrow index = currInd$
 mE10: $pc \in \{121, 123\}$
 $\wedge to.table[key(a_{mE}, to.size, n_{mE})] = \mathbf{null}$
 $\Rightarrow index = currInd$
 mE11: $pc \in \{121, 123\} \wedge pc.r = 103$
 $\wedge to.table[key(a_{mE}, to.size, n_{mE})] = \mathbf{null}$
 $\Rightarrow index.r \neq currInd$
 mE12: $pc \in \{121, 123\} \wedge next(currInd) \neq 0 \wedge to = H(next(currInd))$
 $\Rightarrow n_{mE} < H(next(currInd)).size$
 mE13: $pc \in \{123, 125\} \wedge w_{mE} \neq \mathbf{null}$
 $\Rightarrow ADR(w_{mE}) = ADR(to.table[k_{mE}])$
 $\vee to.table[k_{mE}] \in \{\mathbf{del}, \mathbf{done}\}$
 mE14: $pc \geq 123 \wedge w_{mE} \neq \mathbf{null}$
 $\Rightarrow H(i_{mig}).table[k_{mE}] \neq \mathbf{null}$
 mE15: $pc = 117 \wedge val(v_{mC}) \neq \mathbf{null}$
 $\vee pc \in \{121, 123\} \wedge n_{mE} > 0$
 $\vee pc = 125$
 $\Rightarrow h.table[key(ADR(v_{mC}), h.size, 0)] \neq \mathbf{null}$,
 where $h = H(i_{mig})$
 mE16: $pc \in \{121, 123\}$
 $\vee (pc = 125 \wedge \neg b_{mE}$
 $\wedge (val(w_{mE}) = \mathbf{null} \vee a_{mE} \neq ADR(w_{mE})))$
 $\Rightarrow \forall m < n_{mE} :$
 $\neg Find(to.table[key(a_{mE}, to.size, m)], a_{mE})$

Invariants about the integer array prot.

pr1: $prot[i] = \#(prSet1(i)) + \#(prSet2(i)) + \#(currInd = i) + \#(next(currInd) = i)$
 pr2: $prot[currInd] > 0$
 pr3: $pc \in [1, 58] \vee pc \geq 62 \wedge pc \neq 65 \Rightarrow prot[index] > 0$
 pr4: $next(currInd) \neq 0 \Rightarrow prot[next(currInd)] > 0$
 pr5: $prot[i] = 0 \Rightarrow Heap(H[i]) = \perp$
 pr6: $prot[i] \leq \#(prSet3(i)) \wedge busy[i] = 0 \Rightarrow Heap(H[i]) = \perp$
 pr7: $pc \in [67, 72] \Rightarrow prot[i_{rA}] > 0$
 pr8: $pc \in [81, 84] \Rightarrow prot[i_{nT}] > 0$
 pr9: $pc \geq 97 \Rightarrow prot[i_{mig}] > 0$
 pr10: $pc \in [81, 82] \Rightarrow prot[i_{nT}] = \#(prSet4(i_{nT})) + 1$

Invariants about the integer array busy.

bu1: $busy[i] = \#(buSet1(i)) + \#(buSet2(i)) + \#(currInd = i) + \#(next(currInd) = i)$
 bu2: $busy[currInd] > 0$
 bu3: $pc \in [1, 58]$
 $\vee pc > 65 \wedge \neg(i_{rA} = index \wedge pc \in [67, 72])$

- $\Rightarrow \text{busy}[index] > 0$
- bu4: $\text{next}(\text{currInd}) \neq 0 \Rightarrow \text{busy}[\text{next}(\text{currInd})] > 0$
- bu5: $pc = 81 \Rightarrow \text{busy}[i_{nT}] = 0$
- bu6: $pc \geq 100 \Rightarrow \text{busy}[i_{mig}] > 0$

Some other invariants we have postulated:

- Ot1: $\text{X}(0) = \text{null}$
- Ot2: $\text{X}(a) \neq \text{null} \Rightarrow \text{ADR}(\text{X}(a)) = a$

The motivation of invariant (Ot1) is we never store a value for the address 0. The motivation of invariant (Ot2) is that the address in the hashtable is unique.

- Ot3: $\text{return}_{gA} = \{1, 10, 20, 30, 36, 46, 51\} \wedge \text{return}_{rA} = \{0, 59, 77, 90\}$
 $\wedge \text{return}_{ref} = \{10, 20, 30, 36, 46, 51\} \wedge \text{return}_{nT} = \{30, 46\}$
- Ot4: $pc \in \{0, 1, 5, 6, 7, 8, 10, 11, 13, 14, 15, 16, 17, 18, 20,$
 $21, 25, 26, 27, 28, 30, 31, 32, 33, 35, 36, 37, 41,$
 $42, 43, 44, 46, 47, 48, 49, 50, 51, 52, 57, 59, 60,$
 $61, 62, 63, 65, 67, 68, 69, 70, 71, 72, 77, 78, 81,$
 $82, 83, 84, 90, 94, 95, 97, 98, 99, 100, 101, 102,$
 $103, 104, 105, 110, 111, 114, 116, 117, 118, 120,$
 $121, 123, 125, 126\}$

B Dependencies between invariants

Let us write “ $\varphi \text{ from } \psi_1, \dots, \psi_n$ ” to denote that φ can be proved to be an invariant using ψ_1, \dots, ψ_n hold. We write “ $\varphi \Leftarrow \psi_1, \dots, \psi_n$ ” to denote that φ can be directly derived from ψ_1, \dots, ψ_n . We have verified the following “from” and “ \Leftarrow ” relations mechanically:

- Co1 **from** fi10, Ot3, fi1
- Co2 **from** de5, Ot3, de6, del, de11
- Co3 **from** in5, Ot3, in6, in1, in11
- Cn1 **from** Cn6, Ot3
- Cn2 **from** Cn8, Ot3, del
- Cn3 **from** Cn10, Ot3, in1, in5
- Cn4 **from** Cn11, Ot3
- No1 \Leftarrow No2
- No2 **from** nT1, He2, rA2, Ot3, Ha2, Ha1, rA1, rA14, rA3, nT14, rA4
- He1 **from** Ha1
- He2 **from** Ha3, rA5, Ha1, He1, rA2
- He3, He4 **from** Ot3, rA6, rA7, mi12, rA11, rA5
- He5 **from** He1
- He6 **from** rA8, Ha3, mi8, nT2, rA5
- Ha1 **from** true
- Ha2 **from** Ha1
- Ha3 **from** Ha2, Ha1, He2, He1
- Ha4 \Leftarrow Ha3, He3, He4
- Cn5 **from** Cn6, Ot3
- Cn6 **from** Cn5, Ot3
- Cn7 **from** Cn8, Ot3, del
- Cn8 **from** Cn7, Ot3
- Cn9 **from** Cn10, Ot3, in1, in5

Cn10 **from** Cn9, Ot3, in5
 Cn11 **from** Cn11, Ot3
 Cu1 **from** Ot3, Ha4, rA6, rA7, nT13, nT12, Ha2, He3, He4, rA11, nT9, nT10, mi13, rA5
 $\text{Cu2} \Leftarrow \text{Cu6}, \text{cu7}, \text{Cu3}, \text{He3}, \text{He4}$
 Cu3 **from** rA6, rA7, nT13, nT12, mi5, mi4, Ne8, rA5
 Cu4 **from** del, in1, as1, rA6, rA7, Ha2, nT13, nT12, Ne9, Cu9, Cu10, de7, in7, as5, He3,
 He4, mi5, mi4, Ot3, Ha4, de3, mi9, mi10, de5, rA5
 Cu6 **from** Ot3, rA6, rA7, Ha2, nT13, nT12, Ha3, in3, as3, Ne23, mi5, mE6, mE7, mE10,
 mE3, Ne3, mi1, mi4, rA5
 Cu7 **from** Ot3, rA6, rA7, Ha2, nT13, nT12, Ha3, in3, as3, in5, mi5, mE6, mE7, mE10, mE3,
 Ne3, mi4, de7, in7, as5, Ne22, mi9, mi10, rA5, He3, mi12, mi1, Cu9, de1, in1, as1
 Cu8 **from** Cu8, FT, Ha2, nT9, nT10, rA6, rA7, mi5, mi4, mC2, mC5, He3, He4, Cu1, Ha4,
 mC6, mi16, rA5
 Cu9, Cu10 **from** rA6, rA7, nT13, nT12, Ha2, He3, He4, Cu1, Ha4, de3, in3, as3, mE3, mi9,
 mi10, mE10, mE7, rA5
 Cu11, Cu12 **from** Cu9, Cu10, Cu13, Cu14, del, in1, as1, rA6, rA7, Ha2, nT13, nT12, He3,
 He4, Cu1, Ha4, in3, as3, mi14, mi15, de3, in10, as8, mi12, Ot2, fi5, de8, in8, as6, Cu15,
 de11, in11, rA5
 Cu13, Cu14 **from** He3, He4, Ot2, del, in1, as1, Ot1, rA6, rA7, nT13, nT12, Ha2, Cu9, Cu10,
 Cu1, Ha4, de3, in3, as3, Cu11, Cu12, in10, as8, fi5, de8, in8, as6, Cu15, mi17, mi18,
 mi12, mi4, de11, rA5
 Cu15 **from** He3, He4, rA6, rA7, nT13, nT12, Ha2, Cu1, Ha4, del, in1, as1, de3, in3, as3, fi5,
 de8, in8, as6, mi12, mi19, mi4, Ot2, Cu9, Cu10, Cu11, Cu12, Cu13, Cu14, rA5
 $\text{Cu16} \Leftarrow \text{Cu13}, \text{Cu14}, \text{Cu15}, \text{He3}, \text{He4}, \text{Ot1}$
 Ne1 **from** nT9, nT10, mi7
 Ne2 **from** Ne5, nT3, mi8, nT9, nT10
 Ne3 **from** Ne1, nT9, nT10
 Ne4 **from** Ne1, nT9, nT10
 Ne5 **from** Ot3, nT9, nT10, mi5
 $\text{Ne6} \Leftarrow \text{Ne10}, \text{Ne24}, \text{He6}, \text{He3}, \text{He4}, \text{Cu4}$
 Ne7 **from** Ha3, rA6, rA7, rA8, nT13, nT12, nT11, He3, He4, mi8, nT7, Ne5, Ha2, He6, rA5
 Ne8 **from** Ha3, rA8, nT11, T, mi8, nT6, Ne5, rA5
 Ne9 **from** Ha3, Ha2, Ne3, Ne5, de3, as3, rA8, rA6, rA7, nT8, nT11, mC2, nT4, mi8, rA5
 Ne9a **from** Ha3, Ne3, rA5, de3, rA8, nT4, mi8
 Ne10 **from** Ha3, Ha2, de3, rA8, nT11, Ne3, He6, mi8, nT8, mC2, nT2, Ne5, rA5
 Ne11 **from** Ha3, Ha2, He6, T, nT2, nT8, rA8, nT11, mi8, Ne3, mC2, rA5
 Ne12, Ne13 **from** Ha3, Ha2, Cu8, He6, He3, He4, Cu1, de3, in3, as3, rA8, rA6, rA7, nT11,
 nT13, nT12, mi12, mi16, mi5, mi4, de7, in7, as5, Ot2, del, in1, as1, Cu9, Cu10, Cu13,
 Cu14, Cu15, as9, fi5, de8, in8, as6, mC2, Ne3, Ot1, Ne14, Ne20, mE16, mE7, mE4, mE1,
 mE12, mE2, Ne15, Ne16, Ne17, Ne18, mi20, de11, in11, rA5
 Ne14 **from** Ha3, Ha2, He6, He3, He4, T, nT2, nT8, de3, in3, as3, rA8, nT11, Ot2, del, in1,
 as1, Cu9, Cu10, mi8, Ne3, mC2, mE7, mE16, mE1, mE4, mE12, Ne17, Ne18, Cu1, rA5
 Ne15, Ne16 **from** Ha3, Ha2, Cu8, He6, He3, He4, Cu1, de3, in3, as3, rA8, rA6, rA7, nT11,
 nT13, nT12, mi12, mi16, mi5, mi4, de7, in7, as5, Ot2, del, in1, as1, Cu9, Cu10, Cu13,
 Cu14, Cu15, as9, fi5, de8, in8, as6, mC2, Ne3, Ot1, Ne19, Ne20, Ne12, Ne13, mE16, mE7,
 mE4, mE1, mE12, mE10, mE2, in11, de11, rA5
 Ne17, Ne18 **from** Ha3, Ha2, mi8, He6, He3, He4, Cu1, nT2, de3, in3, as3, rA8, rA6, rA7,
 nT11, nT13, nT12, de7, in7, as5, Ot2, del, in1, as1, Cu9, Cu10, T, nT8, mE2, fi5, de8,
 in8, as6, mC2, Ne3, mC11, mC6, mC12, mE7, mE10, mE1, Cu8, Cu15, Cu13, Cu14,
 Cu11, Cu12, as8, de11, rA5
 Ne19 **from** Ha3, Ha2, He6, nT2, nT8, de3, in3, as3, rA8, nT11, mi8, Ne3, mE7, Ne14, mE16,
 Ot1, mE1, mE4, mE12, Ne17, Ne18, rA5
 Ne20 **from** Ha3, Ha2, Cu8, He6, He3, He4, Cu1, Ha4, de3, in3, as3, rA8, rA6, rA7, nT11,
 nT13, nT12, mi12, mi16, mi5, mi4, Ne1, de7, in7, as5, del, in1, as1, Cu9, Cu10, Cu13,

Cu14, Cu15, as9, fi5, de8, in8, as6, mC2, Ne3, Ot1, mi20, in11, rA5
 Ne22 **from** Ot3, rA8, Ha2, nT11, Ha3, de3, in3, as3, mi5, mi4, Ne3, nT18, mE3, mi8, mE10,
 mE7, mE6, Ne5, nT5, nT2, rA5, nT8, nT12, mC2, mE2
 Ne23 \Leftarrow Cu6, cu7, Ne6, Ne7, He3, He4, Ne22, He6
 Ne24 \Leftarrow Ne27, He6
 Ne25 \Leftarrow Ne19, Ne17, Ne18, He6
 Ne26 \Leftarrow Ne17, Ne18, He6
 Ne27 \Leftarrow Cu16, Ne25, Ne26, Ne17, Ne18, He6
 fi1, del, in1, as1 **from**
 fi2 **from** fi2, Ot3
 fi3 **from** fi4, Ot3, rA6, rA7, Ha2, rA5
 fi4 **from** Ot3, rA6, rA7, nT13, nT12
 fi5, de8, in8, as6 \Leftarrow Cu2, de10, in10, as8, fi8, He3, He4
 fi6 **from** Ot3, fi1, del, in1, as1, rA6, rA7, Ha2, nT13, nT12, mi9, mi10, Cu9, Cu10, He3,
 He4, Cu1, Ha4, fi4, in3, as3, rA5
 fi7 **from** fi8, fi6, fi2, Ot3, fi1, del, in1, as1, rA6, rA7, Ha2, nT13, nT12, mi9, mi10, Cu9,
 Cu10, He3, He4, Cu1, Ha4, fi4, in3, as3, rA5
 fi8 **from** fi4, fi7, fi2, Ot3, fi1, del, in1, as1, rA6, rA7, Ha2, nT13, nT12, mi9, mi10, Cu9,
 Cu10, He3, He4, Cu1, Ha4, in3, as3, rA5
 fi9 \Leftarrow Cu1, Ha4, Cu9, Cu10, Cu11, Cu12, fi8, fi3, fi4, fi5, de8, in8, as6, He3,
 He4
 fi10 **from** fi9, Ot3
 fi11, de12, in12, as10 **from** Ot3, nT9, nT10, mi9, mi10, Cu8, fi4, de3, in3, as3, fi3, de2,
 in2, as2
 de2 **from** de3, Ot3, rA6, rA7, Ha2, rA5
 de3 **from** Ot3, rA6, rA7, nT13, nT12
 de4, in4, as4 **from** Ot3
 de5 **from** Ot3
 de6 **from** Ot3, de1, de11
 de7, in7, as5 \Leftarrow de3, in3, as3, Cu1, Ha4, de13, in13, as11
 de9 **from** Ot3, del, in1, as1, rA6, rA7, Ha2, nT13, nT12, mi9, mi10, Cu9, Cu10, de3, de7,
 in7, as5, rA5
 de10 **from** de3, de9, Ot3, del, in1, as1, rA6, rA7, Ha2, nT13, nT12, mi9, mi10, Cu9, Cu10,
 de7, in7, as5, He3, He4, rA5
 de11 \Leftarrow de10, de2, de3, He3, He4, Cu1, Ha4, Cu9, Cu10, Cu11, Cu12, fi5, de8, in8, as6
 de13, in13, as11 \Leftarrow Ax2, de2, de3, de4, in2, in3, in4, as2, as3, as4
 in2 **from** in3, Ot3, rA6, rA7, Ha2, rA5
 in3 **from** Ot3, rA6, rA7, nT13, nT12
 in5 **from** Ot3
 in6 **from** Ot3, in1, in11
 in9 **from** Ot3, del, in1, as1, rA6, rA7, Ha2, nT13, nT12, mi9, mi10, Cu9, Cu10, He3, He4,
 in3, de7, in7, as5, rA5
 in10 **from** in9, fi2, Ot3, del, in1, as1, rA6, rA7, Ha2, nT13, nT12, mi9, mi10, Cu9, Cu10,
 He3, He4, in3, de7, in7, as5, rA5
 in11 \Leftarrow in10, in2, in3, Cu1, Ha4, Cu9, Cu10, Cu11, Cu12, fi5, de8, in8, as6
 as2 **from** as3, He3, He4, Ot3, rA6, rA7, Ha2, rA5
 as3 **from** Ot3, rA6, rA7, nT13, nT12
 as7 **from** Ot3, del, in1, as1, rA6, rA7, Ha2, nT13, nT12, mi9, mi10, Cu9, Cu10, as3, de7,
 in7, as5, rA5
 as8 **from** as7, Ot3, del, in1, as1, rA6, rA7, Ha2, nT13, nT12, mi9, mi10, Cu9, Cu10, He3,
 He4, as3, de7, in7, as5, rA5
 as9 \Leftarrow as8, as2, as3, He3, He4, Cu1, Ha4, Cu9, Cu10, Cu11, Cu12, fi5, de8, in8, as6
 rA1 **from** Ha2
 rA2 **from** Ot3

rA3 **from** Ot3, rA9, He2, He1, rA2, rA13
 rA4 **from** Ot3, nT14
 rA5 **from** Ot3, rA1, rA2, Ha3, He2
 rA6, rA7 **from** Ot3, nT13, nT12, nT14, rA11, mi4, bu2, bu3, Ha3, mi6, Ha2, He3, He4,
 He2, rA2
 rA8 **from** Ot3, bu4, nT14, mi6, Ne2, mi5
 rA9 **from** Ot3, Ha2, nT14, He1, He2
 rA10 **from** Ot3
 rA11 **from** Ot3, nT13, nT12, mi2
 rA12 **from** Ot3, nT9, nT10
 rA13 **from** Ot3, rA5
 rA14 **from** Ot3, rA4, He1, rA2
 nT1 **from** Ot3, pr5, Ha3, nT14, nT16, Ha2
 nT2 **from** Ot3, nT14, Ha3, rA5
 nT3 **from** Ot3, nT9, nT10
 nT4 **from** Ot3, Ha3, de3, nT13, nT12, nT15, rA5
 nT5 **from** Ot3, Ha3, in3, as3, nT13, nT12, nT15, nT18, mE3, mi4, rA5
 nT6 **from** Ot3, nT13, nT12, nT14, Ha3, rA5
 nT7 **from** Ot3, nT13, nT12, nT15, rA6, rA7, Ha2, mi9, mi10, nT14, Ha3, nT16, rA5
 nT8 **from** Ot3, de3, in3, as3, nT13, nT12, nT15, nT18, mE3, mi4, Ha3, mC2, nT16, nT2,
 Ha2, rA5
 nT9, nT10 **from** Ot3, pr2, pr3, nT18
 nT11 **from** Ot3, pr4, nT16, mi8
 nT13, nT12 \Leftarrow nT9, nT10, Ha3, He3, He4
 nT14 **from** Ot3, nT9, nT10, nT18, nT16, pr7
 nT15 \Leftarrow nT14, Ha3, nT2
 nT16 **from** Ot3, pr8
 nT17 **from** Ot3, mi5, pr4, nT11, mi10
 nT18 **from** Ot3, pr9, mi5, nT11
 mi1 **from** Ot3, mi9, mi10, mi10
 mi2 **from** Ot3, Ne4
 mi3 **from** Ot3, fi11, de12, in12, as10, nT9, nT10, Ne5
 mi4 **from** Ot3, mi9, mi10, mi3
 mi5 **from** Ot3, nT9, nT10, Ne5, mi10, mi4
 mi6 **from** Ot3, mi5, bu6, rA8, mi9, mi10, bu4, mi4
 mi7 **from** Ot3, mi2, mi7, mi4, nT18, Ne2, mi10, nT17, mi3
 mi8 **from** Ot3, mi10, Ne2, mi3
 mi9, mi10 **from** Ot3, He3, He4, nT9, nT10, nT18, Ne3, Ha3, mi3, nT17, mi10, He2, mi4,
 mi12, mi6, He6
 mi11 **from** Ot3, nT18, mi9, mi6, mi6
 mi12 **from** Ot3, rA8, nT2, He6, mi9, mi5, mi3, Ha3, mi4, rA5
 mi12 **from** Ot3, mi12, nT18, mi6, Ha3, mi4, rA5
 mi13 **from** Ot3, rA6, rA7, Ha2, nT13, nT12, He3, He4, mi9, mi10, mC9, rA5
 mi14, mi15 \Leftarrow Ne12, Ne13, mi5, Cu15, mi13, Ot2, He3, He4, Ne17, Ne18, Cu8, He6, He5,
 mi4, Ot1
 mi16 \Leftarrow Ne11, mi5, mi4
 mi17, mi18 \Leftarrow Ne15, Ne16, mi5, Cu15, mi13, Ot2, He3, He4, Ne17, Ne18, Cu8, He6, He5, mi4
 mi19 \Leftarrow Ne20, mi5, Cu15, mi13, Ot2, He3, He4
 mi20 **from** Ha3, Ha2, Cu8, He6, He3, He4, Cu1, Ha4, de3, in3, as3, rA8, rA6, rA7, nT11,
 nT13, nT12, mi5, mi4, de7, in7, as5, Ot2, del, in1, as1, Cu9, Cu10, Cu13, Cu14, Cu15,
 as9, fi5, de8, in8, as6, mC6, Ne3, Ot3, mC11, mi13, mi9, mi10, mC2, mE3, mE10, mE7,
 mC12, mE1, mE13, Ne17, Ne18, mE2, mE4, Ot1, mE6, Ne10, in11, rA5
 mC1 **from** Ot3, mi6, mi11, nT18
 mC2 **from** Ot3, rA6, rA7, nT13, nT12, mC2

mC3 **from** Ot3, mC3, nT13, nT12, rA6, rA7, Ha2, rA5
 mC4 **from** Ot3, mC4, mC2, mC3, He3, He4, rA6, rA7, Ha2, rA5
 mC5 **from** Ot3
 mC6 **from** Ot3, rA6, rA7, Ha2, nT13, nT12, mC2, rA5
 mC7 **from** Ot3, rA6, rA7, Ha2, nT13, nT12, mC2, rA5
 mC8 **from** Ot3, rA6, rA7, Ha2, nT13, nT12, He3, He4, mC7, rA5
 mC9 **from** Ot3, rA6, rA7, Ha2, nT13, nT12, He3, He4, mi9, mi10, He5, mC7, mC8, rA5
 mC10 **from** Ot3, rA6, rA7, Ha2, nT13, nT12, mC2, del, in1, as1, mi6, Ha3, mi4, nT18, mE15,
 mC11, mi5, rA5
 mC11 **from** Ot3, rA6, rA7, Ha2, nT13, nT12, mC2, rA5
 mC12 **from** Ot3, rA6, rA7, mC2, mC11, Cu9, Cu10, de7, in7, as5, mi9, mC6
 mE1 **from** Ot3
 mE2 **from** Ot3
 mE3 **from** mC1, Ot3, mi6, nT18
 mE4 **from** Ot3, mE1
 mE5 **from** Ot3, mE3, Ha3, mi6, mi4, nT18, Ha2, rA5
 mE6 **from** Ot3
 mE7 **from** Ot3, Ha2, Ha3, mi6, mi4, mE3, rA5
 mE8 **from** Ot3, Ha3, mi6, mi4, nT18, Ha2, mE3, rA5
 mE9 **from** Cu1, Ha4, Ot3, Ha2, Ha3, mi6, mi4, mE3, mC2, mC10, mE1, mC1, del, in1, as1,
 mi13, mi12, mC6, mE2, rA5
 mE10 **from** del, in1, as1, mE3, mi6, Ot3, Ha2, Ha3, mi4, mE11, mE9, mE7, rA5
 mE11 \Leftarrow mE10, mi13, mE16, mi16, mi5, mE3, Ne12, Ne13, mC12, mE2, mE1, mE4, mC6,
 mE12, mi12, Cu13, Cu14, He3, He4, mi4
 mE12 \Leftarrow Ne23, Ne22, mE16, He6, Ne8
 mE13 **from** Ot3, Ha2, mE14, del, in1, as1, Ha3, mi6, mi4, mE3, rA5
 mE14 **from** Ot3, Ha2, del, in1, as1, Ha3, mi6, mi4, nT18, mE3, mE2, rA5
 mE15 **from** Ot3, mE1, Ha2, del, in1, as1, Ha3, mi6, mi4, nT18, mE3, mE2, mE7, mE14,
 mE4, rA5
 mE16 **from** Ha3, Ha2, mE3, del, in1, as1, mi6, mE2, mE4, mE1, mE7, mi4, Ot3, mE14,
 mE13, rA5
 pr1 **from** Ot3, rA11, rA10, nT9, nT10, Ne5, mi2, mi4, mi8, mi5
 pr2, pr3 **from** pr1, Ot3, rA11, mi1
 pr4 \Leftarrow pr1
 pr5 \Leftarrow pr6, pr1, bu1
 pr6 **from** Ot3, Ha2, nT9, nT10, nT14, nT16, He2, rA2, pr1, bu1, pr10, rA9, He1, rA4
 pr7, pr8, pr9 \Leftarrow pr1, mi4
 pr10 **from** Ot3, pr1, nT9, nT10, nT14, nT17
 bu1 **from** Ot3, rA11, rA10, nT9, nT10, Ne5, mi2, mi8, mi5, bu5
 bu2, bu3 \Leftarrow bu1, Ot3, rA10
 bu4 \Leftarrow bu1
 bu5 **from** Ot3, nT9, nT10, nT16, nT18, pr1, bu1
 bu6 \Leftarrow bu1, mi4
 Ot1 **from** del, in1, as1
 Ot2 **from** del, in1, as1
 Ot3 **from** true
 Ot4 **from** Ot3

References

- [1] Attiya, H., Bar-Noy, A., Dolev, D., Peleg, D. Reischuk, R.: Renaming in an asynchronous environment. J. ACM **37** (1990) 524–548

- [2] Bar-Noy, A., Dolev, D.: Shared-memory vs. message-passing in an asynchronous distributed environment. In Proc. 8th ACM Symp. on principles of distributed computing, pp. 307–318, 1989
- [3] Cassez, F., Jard, C., Rozoy, B., Dermot, M. (Eds.): Modeling and Verification of Parallel Processes. 4th Summer School, MOVEP 2000, Nantes, France, June 19-23, 2000.
- [4] Groote, J.F., Hesselink, W.H., Mauw, S., Vermeulen, R.: An algorithm for the asynchronous write-all problem based on process collision. *Distributed Computing* **14** (2001) 75–81
- [5] Harbison, S.P.: Modula-3, Prentice Hall 1992
- [6] Herlihy, M.P.: Wait-free synchronization. *ACM Trans. on Program. Languages and Systems* **13** (1991) 124–149
- [7] Herlihy, M.: A methodology for implementing highly concurrent data objects. *ACM Trans. on Programming Languages and Systems* **15** (1993), 5
- [8] Herlihy, M.P. and Moss, J.E.B.: Lock-free garbage collection for multiprocessors. *IEEE Transactions on Parallel and Distributed Systems* **3** 304–311, 1992
- [9] Hesselink, W.H.: Wait-free linearization with a mechanical proof. *Distrib Comput* **9** (1995) 21–36
- [10] Hesselink, W.H.: Bounded Delay for a Free Address. *Acta Informatica* **33** (1996) 233–254
- [11] Hesselink, W.H., Groote, J.F.: Wait-free concurrent memory management by Create, and Read until Deletion (CaRuD). *Distributed Computing* **14** (2001) 31–39
- [12] <http://www.cs.rug.nl/~wim/mecchver/hashtable>
- [13] Kanellakis, P.C. and Shvartsman, A.A.: *Fault-tolerant parallel computation*. Kluwer Academic Publishers, 1997
- [14] Lamport, L.: The temporal logic of actions. *ACM Trans. on Programming Languages and Systems* **16** (1994) 872–923.
- [15] Knuth, D.E.: The Art of Computer Programming. Part 3, Sorting and searching. Addison-Wesley, 1973.
- [16] Lynch, N.A.: Distributed Algorithms. Morgan Kaufman, San Francisco, 1996.
- [17] Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems: Specification. Springer-Verlag, 1992.
- [18] Owre, S., Shankar, N., Rushby, J.M., Stringer-Calvert, D.W.J.: PVS Version 2.4 (2001). System Guide, Prover Guide, PVS Language Reference. <http://pvs.csl.sri.com>
- [19] Valois, J.D.: Lock-free linked lists using compare-and-swap. *Proceedings of the 14th Annual Principles of Distributed Computing*, pages 214-222, 1995. See also J.D. Valois. ERRATA. Lock-free linked lists using compare-and-swap. Unpublished manuscript, 1995
- [20] Valois, J.D.: Implementing Lock-Free Queues, *Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems*, pages. 64-69, Las Vegas, October 1994
- [21] Wirth, N.: Algorithms + Data Structures = Programs. Prentice Hall, 1976.

Technical Report

UCAM-CL-TR-639
ISSN 1476-2986

Number 639



UNIVERSITY OF
CAMBRIDGE

Computer Laboratory

Non-blocking hash tables with open addressing

Chris Purcell, Tim Harris

September 2005

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500
<http://www.cl.cam.ac.uk/>

© 2005 Chris Purcell, Tim Harris

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

<http://www.cl.cam.ac.uk/TechReports/>

ISSN 1476-2986

Non-blocking Hashtables with Open Addressing

Chris Purcell

Tim Harris

Abstract

We present the first non-blocking hashtable based on open addressing that provides the following benefits: it combines good cache locality, accessing a single cacheline if there are no collisions, with short straight-line code; it needs no storage overhead for pointers and memory allocator schemes, having instead an overhead of two words per bucket; it does not need to periodically reorganise or replicate the table; and it does not need garbage collection, even with arbitrary-sized keys. Open problems include resizing the table and replacing, rather than erasing, entries. The result is a highly-concurrent set algorithm that approaches or outperforms the best externally-chained implementations we tested, with fixed memory costs and no need to select or fine-tune a garbage collector or locking strategy.

1 Introduction

This paper presents a new design for non-blocking hashtables in which collisions are resolved by open addressing, i.e. probing through the other buckets of the table, rather than external chaining through linked lists.

The key idea is that rather than leaving tombstones to mark where deletions occur, we store per-bucket upper bounds on the number of other buckets that need to be consulted. This means that unlike the earlier designs we discuss in Section 2.2, ours supports a mixed workload of insertions and deletions without the need to periodically replicate the table’s contents to clean out tombstones. Consequently, the table can operate without the need for dynamic storage management so long as its load factor remains acceptable.

Our design is split into three parts. Section 3.1 deals with maintaining the shared bounds associated with each bucket. The key difficulty here is ensuring that a bound remains correct when several entries are being inserted and removed at once. Section 3.2 builds on this to provide a hashtable. The main problem in doing so is guaranteeing non-blocking progress while ensuring that at most one instance of any key can be present in the table. In Section 3.3, we present a more complicated design allowing larger keys and a better progress guarantee, and in Section 3.4 we discuss open problems with the algorithm.

Section 4 evaluates the performance of our algorithm, compared to state-of-the-art designs based on external chaining. As with these, we rely only on the single-word atomic operations found on all modern processor families. Additionally, our algorithm has many properties that machines rely on for optimal performance: operations run independently, updating disjoint memory locations (*disjoint access parallel*) and not modifying shared memory during logically read-only operations (*read parallel*), and hence typically run in

parallel on multi-processor machines. Finally, a low *operation footprint* (shared memory touched per operation) gives greater throughput under stress by easing pressure on the memory subsystem.

Our results reflect this, demonstrating performance comparable with the best existing designs in all tested cases. On highly-parallel workloads with many updates, our algorithm ran 35% faster; while a single-threaded run with mostly read-only operations was the worst case, running 40% slower than the best existing design.

Proof of correctness and progress properties can be found in Appendix A.

2 Background

2.1 Non-blocking Progress Guarantees

Data structures are easiest to implement when accessed in isolation, but general schemes for enforcing that isolation — for instance, using mutual exclusion locks — typically result in poor scalability and robustness in the face of contention and failure. Concurrent algorithms that avoid mutual exclusion are generally *non-blocking*: suspension of any subset of threads will not prevent forward progress by the rest of the system.

The weakest non-blocking guarantee is *obstruction-freedom*: if at any time a thread runs in isolation, it will complete its operation within a bounded number of steps. This precludes mutual exclusion, as suspension of a lock-holding thread will prevent others waiting on that lock from making progress. *Lock-freedom* combines this with guaranteed throughput: any active thread taking a bounded number of steps ensures global progress. Unfortunately, creating *practical* non-blocking forms of even simple data structures is notoriously difficult.

2.2 Related Work

Externally-chained hashtables store each bucket’s collisions¹ in a list. Michael introduced the first practical lock-free hashtables based on external chaining with linked lists [8]. Shalev and Shavit described *split-ordered lists* that allow the number of buckets to vary dynamically [9]. Fraser detailed lock-free skip-lists and binary search trees [2]. Recently, Lea has contributed a high-performance, scalable, *lock-based*, externally-chained hashtable design to the latest version of Java (5.0), which avoids locking on most read-operations, preserving read-parallelism.

All of the above designs rely on an out-of-band garbage collector. Michael reported that reference counting was unacceptably slow for this purpose as it did not preserve read-parallelism; he proposed using safe memory reclamation [7] to get a strictly bounded memory overhead. Fraser used a simple low-overhead garbage collection scheme, *epoch-based reclamation*, where all threads maintain a current epoch, and memory is reclaimed only after all epochs change; this has a potentially unbounded memory footprint, and a large one in practice.

Tombstones are the traditional means of handling deletion in an open addressed hashtable [3], but cause degenerate search times in the face of a random workload with frequent deleting. Martin and Davis [5] proposed using periodic table replication to limit

¹We refer to a key stored outside its primary bucket as a *collision*.

Probe bound	0	2	0	0	1	0	0	0
Key	-	9	2	-	17	12	-	7
2 steps in probe sequence								

Figure 1: Bounds on collision indices for a hash table holding keys $\{3, 7, 9, 12, 17\}$. Hash function is (key mod 8), probe sequence is quadratic $[\frac{1}{2}(i^2 + i)]$. Key 17 is stored two steps along the probe sequence for bucket 1, so the probe bound is 2.

tombstone growth, relying on garbage collection to reclaim old tables. More recently, Gao *et al.* [1] presented a design with in-built garbage collection.

Both designs limit tombstone reuse to reinsertions of the old key, to achieve linearizability, and do not address the issue of storing multi-word keys directly in the table. The rest of our paper presents solutions to these problems, which we believe are compatible with the replication algorithms already proposed.

3 Memory-Management-Free Open Addressing

Each bucket in our hashtable stores a bound on its collisions’ indices in the probe sequence (Figure 1). When running in isolation, a reader follows the probe sequence this number of steps before terminating; an insert that collides raises the bound if necessary; and an erase that empties the last bucket in this truncated probe sequence searches back for the previous collision and decreases the bound accordingly.

We make this safe for concurrent use in two steps, first maintaining each bucket’s bound in Section 3.1, then ensuring keys are not duplicated in Section 3.2.

3.1 Bounding Searches

Maintaining the probe bounds concurrently is complicated by the need to lower them: simply scanning the probe sequence for the previous collision and swapping it into the bound field may result in the bound being too large if the collision is removed, slowing searches, or too small if another collision is inserted, violating correctness (Figure 2).

In order to keep the bounds correct during erasures, we use a *scanning phase* during which the thread erasing the last collision in the probe sequence searches through the previous buckets to compute the new bound (lines 18–22). A thread announces that it is in this phase by setting a *scanning bit* to true (line 18); this bit is held in the same word as the bound itself, so both fields are updated atomically.

Dealing with insertions is now easy: they atomically clear the scanning bit and raise the bound if necessary (lines 9–12). Deletions also clear the scanning bit (line 16), but are complicated by the scanning phase. We rely on the fact that at most one thread can be in the process of erasing a given collision, and that threads only start scanning when erasing the last collision in the probe sequence. The collision’s index value thus identifies the scanning thread and, if it is still present as the bound when scanning completes, and if the scanning bit is still set, we know there have been no concurrent updates (line 22). Otherwise, we retry the scanning phase.

Probe bound	0	3	0	0	0	0	0	0
Key	-	17	1	-	-	5	-	-

After a collision is removed, a thread scans for the previous collision.

Probe bound	0	1	0	0	0	0	0	0
Key	-	17	-	-	-	5	-	-

If a concurrent erasure is missed, the bound may be left too large.

Probe bound	0	1	0	0	0	0	0	0
Key	-	17	1	-	9	5	-	-

Worse, if a concurrent insertion is missed, the bound may be made too small.

Figure 2: Problems maintaining a shared bound after a collision is removed from the end of the probe sequence.

Given a lock-free atomic compare-and-swap (CAS) function, the pseudocode in Figure 3 is lock-free. We represent the packing of an int and a bit into a machine word with the $\langle ., . \rangle$ operator.

3.2 Inserting and Removing Keys

Inserting and removing keys concurrently is complicated by the lack of a pre-determined bucket for any given key. Inserting into the first empty bucket is not sufficient because, as Figure 4 shows, a concurrent erasure may alter which bucket is ‘first’, and a key may be duplicated. If duplicate keys are allowed in the table, concurrent key erasure becomes impractical.

To ensure uniqueness, we split insertions into three stages (Figure 5). First, a thread reserves an empty bucket and publishes its attempt by storing the key it is inserting, along with an ‘inserting’ flag. Next, the thread checks the other positions in the probe sequence for that key, looking for other threads with ‘inserting’ entries, or for a completed insertion of the same key. If it finds another insertion in progress in a bucket then it changes that bucket’s state to ‘busy’, stalling the other insertion at that point in time. If it finds another completed insertion of the same key, then its own insertion has failed: it empties its bucket and returns ‘false’. In the final stage, it attempts to finish its own insert, changing the ‘inserting’ flag in its bucket to ‘member’. It must do this with a CAS instruction so that it fails if stalled by another thread; if stalled, the thread republishes its attempt and restarts the second stage.

```

1  class Set {
2      word bounds[size] // ⟨bound,scanning⟩
3
4      void InitProbeBound(int h):
5          bounds[h] := ⟨0,false⟩
6
7      int GetProbeBound(int h): // Maximum offset of any collision in probe seq.
8          ⟨bound,scanning⟩ := bounds[h]
9          return bound
10
11     void ConditionallyRaiseBound(int h, int index): // Ensure maximum ≥ index
12         do
13             ⟨old_bound,scanning⟩ := bounds[h]
14             new_bound := max(old_bound,index)
15             while ¬CAS(&bounds[h],⟨old_bound,scanning⟩,⟨new_bound,false⟩)
16
17     void ConditionallyLowerBound(int h, int index): // Allow maximum < index
18         ⟨bound,scanning⟩ := bounds[h]
19         if scanning = true
20             CAS(&bounds[h],⟨bound,true⟩,⟨bound,false⟩)
21         if index > 0 // If maximum = index > 0, set maximum < index
22             while CAS(&bounds[h],⟨index,false⟩,⟨index,true⟩)
23                 i := index-1 // Scanning phase: scan cells for new maximum
24                 while i > 0 ∧ ¬DoesBucketContainCollision(h, i)
25                     i--
26                     CAS(&bounds[h],⟨index,true⟩,⟨i,false⟩)

```

Figure 3: Per-bucket probe bounds (continued below)

The pseudocode in Figure 6 is obstruction-free. Each bucket contains a four-valued state, one of *empty*, *busy*, *inserting* or *member*, and, for the latter two states, a key. The key and state must be modified atomically; we use the $\langle \cdot, \cdot \rangle$ operator to represent packing them into a single word. A key k is considered inserted if some bucket in the table contains $\langle k, \text{member} \rangle$. The *Hash* function selects a bucket for a given key. The three insertion stages can be found in lines 42–50, 51–60 and 61, respectively.

Unlike Martin and Davis’ approach [5], deleted buckets are immediately free for arbitrary reuse, so table replication is not needed to clear out tombstones. The algorithm preserves read parallelism and, assuming disjoint keys hash to separate memory locations, disjoint access parallelism. In the expected case where the bucket contains no collisions, the operation footprint is two words — a single cache line if buckets and bounds are interleaved.

3.3 Extensions: Lock-Freedom and Multi-word Keys

We now turn to two flaws in the above algorithm. The first is that concurrent insertions may live-lock, each repeatedly stalling the other. One solution is to use an out-of-line contention manager: Scherer and Scott have described many suitable for use in any obstruction-free algorithm [10], which are easy to adopt. Another solution, which we cover in more detail as it is a non-trivial problem, is to make the algorithm lock-free.

The standard method of achieving lock-freedom is to allow operations to assist as well as obstruct each other. As given, however, the hash table cannot support concurrent assistance, as Figure 7 demonstrates: a cell’s contents can change arbitrarily before returning to a previous state, allowing a CAS to succeed incorrectly. This is known as the ABA problem, and we return to it in a moment.

The second problem is storing keys larger than a machine word: in the algorithm as given, this requires a multi-word CAS, which is not generally available. However, we note

Probe bound	0	2	0	0	0	1	0	0
Key	-	9	-	-	1	13	5	-

One thread determines that the first empty bucket is at offset 1, and prepares to insert key 17 there.

Probe bound	0	2	0	0	0	1	0	0
Key	-	-	-	-	1	13	5	-

Another thread removes key 9, and prepares to insert key 17. The first empty bucket is now at offset 0.

Probe bound	0	2	0	0	0	1	0	0
Key	-	17	17	-	1	13	5	-

The two threads now insert, creating a duplicate of the key.

Figure 4: Problems concurrently inserting keys

that a cell’s key is only ever modified by a single writer, when the cell is in busy state. This means we only need to deal with concurrent single-writer multiple-reader access to the cell, rather than provide a general multi-word atomic update. We can therefore use Lamport’s version counters [4] (Figure 8).

If a cell’s state is stored in the same word as its version count, the ABA problem is circumvented, allowing threads to assist concurrent operations. This lets us create a lock-free insertion algorithm (diagram in Figure 9, pseudo-code in Figure 10).

Each bucket contains: a version count; a state field, one of *empty*, *busy*, *collided*, *visible*, *inserting* or *member*; and a key field, publically readable during the latter three stages. The version count and state are maintained so that no state (except busy) will recur with the same version.

As before, a thread finds an empty bucket and moves it into ‘inserting’ state (lines 65–76), and checks the probe sequence for other threads with ‘inserting’ entries, or a completed insertion of the same key (lines 86–106). However, if multiple ‘inserting’ entries are found, the earliest in the probe sequence is left unaltered, and the others moved into ‘collided’ state. When the whole probe sequence has been scanned and all contenders removed, the earliest entry is moved into ‘member’ state (line 105) and the insertion concludes (lines 78–83).

This version of the hashtable is lock-free.

Probe bound	0	2	0	0	1	0	0	0
State	empty	member	member	empty	member	inserting	empty	member
Key	-	9	1	-	17	12	-	7

Initial state

Probe bound	0	2	0	0	1	1	0	0
State	empty	member	member	empty	member	inserting	inserting	member
Key	-	9	1	-	17	12	12	7

Publish the attempted insertion in the second cell in the probe sequence, and raise the probe bound to cover it.

Collision offset bound	0	2	0	0	1	1	0	0
State	empty	member	member	empty	member	busy	inserting	member
Key	-	9	1	-	17	-	12	7

Stall all concurrent insertion attempts.

Probe bound	0	2	0	0	1	1	0	0
State	empty	member	member	empty	member	busy	member	member
Key	-	9	1	-	17	-	12	7

Move bucket into ‘member’ state.

Figure 5: Inserting key 12

3.4 Open Problems: Dynamic Growth and Key Replacement

If the set population approaches the number of buckets, we must replicate into a larger table. The Gao *et al.* [1] replication algorithm may be adaptable for this purpose. No aggregate time or memory cost is incurred on operations, as if the population stabilises, no further replications are required. Assuming each new table doubles in size, discarding the old table after growth is a memory overhead no greater than the final size of the table.

Even if a garbage collector is running, the bounded memory footprint provides several advantages. Many collectors are only activated when memory becomes scarce, so will benefit from less memory usage. Lacking pointers, no costly read or write barriers are needed to ensure memory is not leaked. Finally, the small number of memory allocations needed helps avoid any synchronization the allocator code may contain. The performance and latency benefits of these will depend on the memory management algorithms used.

As given, the algorithm cannot implement a dictionary, storing a value with each key, as there is no way to replace keys.

We hope to report these modifications in future work.

4 Results

In order to assess the performance of our new obstruction-free hashtable, we implemented a range of designs from the literature: Michael’s ‘dynamic lock-free hashtable’, which uses external chains to manage collisions and safe-memory-reclamation (MM-SMR) to manage storage, a variant of Michael’s design using epoch-based garbage collection (MM-Epoch), a further variant of Michael’s design using reference counting (MM-RC), and

```

23     word buckets[size] // <key,state>
24     word* Bucket(int h, int index): // Size must be a power of 2
25         return &buckets[(h + index*(index+1)/2) % size] // Quadratic probing
26
27     bool DoesBucketContainCollision(int h, int index):
28         <k,state> := *Bucket(h,index)
29         return (k ≠ - ∧ Hash(k) = h)
30
31     public:
32         void Init():
33             for i := 0 .. size-1
34                 InitProbeBound(i)
35             buckets[i] := <-,empty>
36
37         bool Lookup(Key k): // Determine whether k is a member of the set
38             h := Hash(k)
39             max := GetProbeBound(h)
40             for i := 0 .. max
41                 if *Bucket(h,i) = <k,member>
42                     return true
43                 return false
44
45         bool Insert(Key k): // Insert k into the set if it is not a member
46             h := Hash(k)
47             i := 0 // Reserve a cell
48             while ¬CAS(Bucket(h,i), <-,empty>, <-,busy>)
49                 i++
50                 if i ≥ size
51                     throw "Table full"
52                 do // Attempt to insert a unique copy of k
53                     *Bucket(h,i) := <k,inserting>
54                     ConditionallyRaiseBound(h,i)
55                     max := GetProbeBound(h) // Scan through the probe sequence
56                     for j := 0 .. max
57                         if j ≠ i
58                             if *Bucket(h,j) = <k, inserting> // Stall concurrent inserts
59                                 CAS(Bucket(h,j), <k,inserting>, <-,busy>)
60                             if *Bucket(h,j) = <k,member> // Abort if k already a member
61                                 *Bucket(h,i) := <-,busy>
62                                 ConditionallyLowerBound(h,i)
63                                 *Bucket(h,i) := <-,empty>
64                             return false
65                         while ¬CAS(Bucket(h,i), <k,inserting>, <k,member>)
66                             return true
67
68         bool Erase(Key k): // Remove k from the set if it is a member
69             h := Hash(k)
70             max := GetProbeBound(h) // Scan through the probe sequence
71             for i := 0 .. max
72                 if *Bucket(h,i) = <k,member> // Remove a copy of <k, member>
73                     if CAS(Bucket(h,i), <k,member>, <-,busy>)
74                         ConditionallyLowerBound(h,i)
75                         *Bucket(h,i) := <-,empty>
76                     return true
77                 return false
78     }

```

Figure 6: Obstruction-free set (continued from Figure 3)

State	empty	inserting
Key	-	12

A single thread is about to complete its insertion of key 12. The next step is to atomically move the cell from inserting to member state.

State	empty	member
Key	-	12

The thread is suspended, and its insertion assisted to completion by another thread.

State	member	inserting
Key	12	12

The key is now removed, and two other threads are concurrently attempting to reinsert key 12. One has just succeeded, and the other is about to remove itself. If the first thread wakes up at this point, it will still atomically move the cell from inserting to member state, duplicating key 12.

Figure 7: Problems assisting concurrent operations

Shalev and Shavit’s ‘split-ordered lists’ using epoch-based garbage collection (SS-Epoch). We also tested Lea’s lock-based hashtable design, again using epoch-based collection. Since performance depends on the locking algorithm and the level of granularity (number of locks), we used a basic spinlock and the MCS lock [6] at different granularities. We compared these against our new design, as presented in Figures 3, 8 and 10 (PH).

Our benchmark is parameterized by the number of concurrent threads and by the range of key values used. We present results for 1–12 threads (running on a Sun Fire V880 with eight 900MHz UltraSPARC-III CPUs) and with 2^{15} keys chosen from $[0, 2^{15}M]$, $M = 2$ or 10. Each update step consists of removing a key then inserting another; finding keys and empty slots is done by trial-and-repetition, choosing candidates uniformly at random, giving $\frac{M^2}{M-1}$ searches on average for each update step. This was designed to avoid hashtable resizing, which simplifies our algorithm, as well as allowing a fine locking granularity and greater read-parallelism in Lea’s, but which unfortunately negates the benefit of split-ordered lists.

Each trial lasted ten seconds, after a three second warm-up period to fill caches, and trials were repeated 20 times, interleaved to avoid short-lived anomalies, to obtain a 90% confidence interval. Our results are shown in Figure 11.

MM-Epoch and MM-SMR consistently outperform MM-RC and SS-Epoch (which, for clarity, are not shown in the results), thanks to low overhead and read-parallelism. Below 8 threads, DL-Epoch performs best with low-overhead spinlocking, avoiding the high cost of spinning with a fine locking granularity.

Searching for a key that is not in the table requires two memory accesses for the PH algorithm, but only one for all others tested. In the absence of contention, this is clearly visible in the results. Applications with a higher lookup hit rate would lower this cost. However, in all test with at least four threads, PH outperforms the other designs; this can largely be attributed to touching fewer cachelines (one rather than two) in the common-case code path for update operations — inter-processor cacheline exchange dominates

```

23  struct BucketT {
24      word vs // <version,state>
25      Key key
26  } buckets[size]
27  word buckets[size] // <key,state>
28
29  BucketT* Bucket(int h, int index): // Size must be a power of 2
30      return &buckets[(h + index*(index+1)/2) % size] // Quadratic probing
31
32  bool DoesBucketContainCollision(int h, int index):
33      <version1,state1> := Bucket(h,index)→vs
34      if state1 = visible ∨ state1 = inserting ∨ state1 = member
35          if Hash(Bucket(h,index)→key) = h
36              <version2,state2> := Bucket(h,index)→vs
37              if state2 = visible ∨ state2 = inserting ∨ state2 = member
38                  if version1 = version2
39                      return true
40                  return false
41
42  public:
43      void Init():
44          for i := 0 .. size-1
45              InitProbeBound(i)
46              buckets[i].vs := <0,empty>
47
48      bool Lookup(Key k): // Determine whether k is a member of the set
49          h := Hash(k)
50          max := GetProbeBound(h)
51          for i := 0 .. max
52              <version,state> := Bucket(h,index)→vs // Read cell atomically
53              if state = member ∧ Bucket(h,index)→key = k
54                  if Bucket(h,index)→vs = <version,member>
55                      return true
56                  return false
57
58      bool Erase(Key k): // Remove k from the set if it is a member
59          h := Hash(k)
60          max := GetProbeBound(h)
61          for i := 0 .. max
62              <version,state> := Bucket(h,index)→vs // Atomically read/update cell
63              if state = member ∧ Bucket(h,index)→key = k
64                  if CAS(Bucket(h,i)→vs, <version,member>, <version,busy>)
65                      ConditionallyLowerBound(h,i)
66                      Bucket(h,i)→vs := <version+1,empty>
67                  return true
68              return false

```

Figure 8: Version-counted derivative of Figure 6 (continued in Figure 10)

Probe bound	0	2	0	0	1	0	0	0
Version	18	2	3	6	4	3	24	7
State	empty	member	member	empty	member	inserting	empty	member
Key		9	1		17	12		7

Initial state

Probe bound	0	2	0	0	1	1	0	0
Version	18	2	3	6	4	3	24	7
State	empty	member	member	empty	member	inserting	inserting	member
Key		9	1		17	12	12	7

Write key and raise probe sequence bound

Probe bound	0	2	0	0	1	1	0	0
Version	18	2	3	6	4	3	24	7
State	empty	member	member	empty	member	inserting	collided	member
Key		9	1		17	12	12	7

Earlier ‘inserting’ entry found; move bucket into ‘collided’ state.

Probe bound	0	2	0	0	1	1	0	0
Version	18	2	3	6	4	3	24	7
State	empty	member	member	empty	member	member	collided	member
Key		9	1		17	12	12	7

Assist completion of earlier entry

Probe bound	0	2	0	0	1	0	0	0
Version	18	2	3	6	4	3	25	7
State	empty	member	member	empty	member	member	empty	member
Key		9	1		17	12		7

Empty bucket, lower probe sequence bound and return **false**.

Figure 9: Inserting key 12 (lock-free algorithm)

```

65     bool Insert(Key k): // Insert k into the set if it is not a member
66         h := Hash(k)
67         i := -1 // Reserve a cell
68         do
69             if ++i ≥ size
70                 throw "Table full"
71             ⟨version,state⟩ := Bucket(h,i)→vs
72             while ¬CAS(&Bucket(h,i)→vs, ⟨version,empty⟩, ⟨version,busy⟩)
73                 Bucket(h,i)→key := k
74             while true // Attempt to insert a unique copy of k
75                 *Bucket(h,i)→vs := ⟨version,visible⟩
76                 ConditionallyRaiseBound(h,i)
77                 *Bucket(h,i)→vs := ⟨version,inserting⟩
78                 r := Assist(k,h,i,version)
79                 if Bucket(h,i)→vs ≠ ⟨version,collided⟩
80                     return true
81                 if ¬r
82                     ConditionallyLowerBound(h,i)
83                     Bucket(h,i)→vs := ⟨version+1,empty⟩
84                     return false
85             version++
86
87     private:
88         bool Assist(Key k,int h,int i,int ver_i): // Attempt to insert k at i
89             // Return true if no other cell seen in member state
90             max := GetProbeBound(h) // Scan through probe sequence
91             for j := 0 .. max
92                 if i ≠ j
93                     ⟨ver_j,state_j⟩ := Bucket(h,j)→vs
94                     if state_j = inserting ∧ Bucket(h,j)→key = k
95                         if j < i // Assist any insert found earlier in the probe sequence
96                             if Bucket(h,j)→vs = ⟨ver_j,inserting⟩
97                                 CAS(&Bucket(h,i)→vs, ⟨ver_i,inserting⟩, ⟨ver_i,collided⟩)
98                                 return Assist(k,h,j,ver_j)
99                         else // Fail any insert found later in the probe sequence
100                            if Bucket(h,i)→vs = ⟨ver_i,inserting⟩
101                                CAS(&Bucket(h,j)→vs, ⟨ver_j,inserting⟩, ⟨ver_j,collided⟩)
102                                ⟨ver_j,state_j⟩ := Bucket(h,j)→vs // Abort if k already a member
103                                if state_j = member ∧ Bucket(h,j)→key = k
104                                    if Bucket(h,j)→vs = ⟨ver_j,member⟩
105                                        CAS(&Bucket(h,i)→vs, ⟨ver_i,inserting⟩, ⟨ver_i,collided⟩)
106                                        return false
107                                CAS(&Bucket(h,i), ⟨ver_i,inserting⟩, ⟨ver_i,member⟩)
108                                return true

```

Figure 10: Lock-free insertion algorithm (continued from Figure 8)

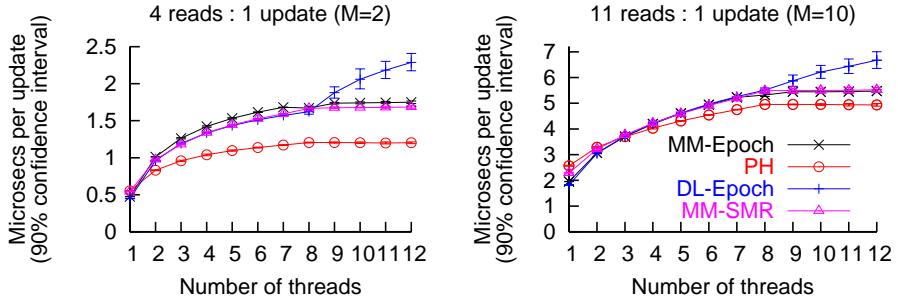


Figure 11: Performance on 8-way SPARC machine

runtime in massively parallel workloads. Applications with much larger, multi-cacheline keys would lose most of this advantage, and may favour an externally-chained scheme to lower the memory footprint of empty buckets.

5 Conclusions

We have presented a lock-free, disjoint-access and read parallel set algorithm based on open addressing, with no need for garbage collection, and touched upon removing population constraints. It has high straight-line speeds and a low operation footprint leading to excellent performance, matching and besting state-of-the-art external-chaining implementations in the tests we performed.

We wish to thank Sun Microsystems, Inc. for donating the SPARC v880 server on which this work was evaluated, and the University of Rochester, New York, for hosting it.

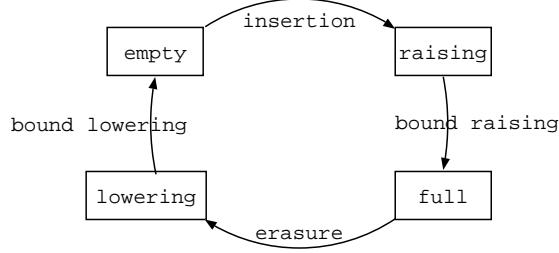


Figure 12: Shared bound state-machine

A Proofs

In the following, we use t_c and t_r for the call and return times of a function, respectively, and t_n for the linearization point of the last pass through line n during that call.

A.1 Shared Bound: Proof of Correctness

For simplicity, we consider a single bucket h , with a probe sequence bound (bound) and a scanning bit (scanning). We model the probe sequence at time t as an infinite sequence $(b_i^t)_{i \geq 0}$ of state machines, as shown in Figure 12, where $b_i^t \in \{\text{empty}, \text{raising}, \text{full}, \text{lowering}\} \forall i, t$, and where $b_i^0 = \text{empty} \forall i$.

The pre-condition for the *bound raising* transition for bucket i is that $\text{bound} \geq i$ and $\text{scanning} = \text{false}$, and for *bound lowering* that $\text{bound} \neq i$ and $\text{scanning} = \text{false}$; the *insertion* and *erasure* transitions are handled by the calling code.

Our correctness criteria consist of the following claims about the code, given certain restrictions on the calling code:

CRITERION. *ConditionallyRaiseBound(h,i)* post-condition: $b_i^{t_r} = \text{full}$, assuming $b_i^t \in \{\text{raising}, \text{full}\} \forall t \in [t_c, t_r]$

CRITERION. *ConditionallyLowerBound(h,i)* post-condition: $b_i^{t_r} = \text{empty}$, assuming $b_i^t \in \{\text{lowering}, \text{empty}\} \forall t \in [t_c, t_r]$

CRITERION. *GetProbeBound(h)* returns $\max(\{i : b_i^{t_6} \in \{\text{full}, \text{lowering}\}\} \cup \{0\})$

The following two conditions are assumed to hold:

COLLISION CONDITION. *DoesBucketContainCollision(h,i)* returns **true** (resp. **false**) only if $b_i^t \in \{\text{raising}, \text{full}\}$ (resp. $\{\text{lowering}, \text{empty}\}$) holds for some $t \in [t_c, t_r]$, where *DoesBucketContainCollision(h,i)* is called at t_c and returns at t_r

EXCLUSIVITY CONDITION. $|R_i^t \cup L_i^t| = 1 \forall i, t$

where $R_i^t = \{\text{threads in ConditionallyRaiseBound}(h,i) \text{ at time } t\}$
and $L_i^t = \{\text{threads in ConditionallyLowerBound}(h,i) \text{ at time } t\}$

FIRST SCANNING LEMMA. The scanning bit is cleared whenever the bound changes, and the scanning bit is only set at line 18.

PROOF. Examination of the code. Only lines 12, 16, 18 and 22 change the bound and scanning bit, and none violate the lemma.

POST-CONDITION ON CONDITIONALLYRAISEBOUND(H,I). $b_i^{t_r} = \text{full}$, assuming $\forall t \in [t_c, t_r], b_i^t \in \{\text{raising}, \text{full}\}$

PROOF. The CAS at line 12 must succeed before t_r , and a successful CAS ensures $\text{bound}^{t_{12}} \geq i$ and $\text{scanning}^{t_{12}} = \text{false}$, triggering the *bound raising* state transition if bucket i is in raising state. By assumption, there are no erasure transitions, so the post-condition must hold.

LEMMA 2. $\exists t \in [t_{14}, t_{16}]$ s.t. $\text{scanning}^t = \text{false}$

PROOF. If $\text{scanning}^{t_{14}} = \text{true}$ and $\text{scanning}^{t_{16}-} = \text{true}$, either $\text{bound}^{t_{14}} = \text{bound}^{t_{16}-}$ and the CAS at line 16 succeeds, ensuring $\text{scanning}^{t_{16}} = \text{false}$, or we appeal to the First Scanning Lemma. The lemma follows.

POST-CONDITION ON LINE 22. $\langle \text{bound}, \text{scanning} \rangle^{t_{22}} \neq \langle \text{index}, \text{true} \rangle$

SECOND SCANNING LEMMA. If $\text{bound} = \langle \text{index}, \text{true} \rangle$, some thread is executing `ConditionallyLowerBound(h, index)` lines 19–22.

PROOF. Only line 18 can set $\text{bound} = \langle \text{index}, \text{true} \rangle$; if this occurs, lines 19–22 will be executed. The post-condition on line 22 then implies the lemma.

PRE-CONDITION ON LINE 18. $\langle \text{bound}, \text{scanning} \rangle^{t_{22}} \neq \langle \text{index}, \text{true} \rangle$

PROOF. Consequence of the exclusivity condition and the Second Scanning Lemma.

LEMMA 6. If $\text{index} > 0$, $\exists t \in [t_{14}, t_r]$ s.t. $\text{bound}^t \neq \text{index}$ and $\text{scanning}^t = \text{false}$

PROOF. By lemma 2, $\exists t_* \in [t_{14}, t_{16}]$ s.t. $\text{scanning}^{t_*} = \text{false}$. By the pre-condition on line 18, the loop of lines 18–22 will only end when $\text{bound}^{t_{18}} \neq \text{index}$. Thus, if $\text{bound}^{t_*} = \text{index}$, we can appeal to the First Scanning Lemma to find t ; otherwise, $t = t_*$ satisfies the lemma.

CLAIM. `ConditionallyLowerBound(h, i)` post-condition: $b_i^{t_r} = \text{empty}$ assuming $b_i^t \in \{\text{lowering}, \text{empty}\}$ $\forall t \in [t_c, t_r]$

PROOF. By lemma 6, the *bound lowering* state transition must occur during `ConditionallyLowerBound(h, i)` if bucket i is in lowering state. The pre- and during-conditions prevent an insertion transition, so the post-condition must hold.

THEOREM. $\forall i, t, (b_i^t \in \{\text{empty}, \text{raising}\} \Rightarrow \text{bound}^t \neq i),$

$(b_i^t \in \{\text{full}, \text{lowering}\} \Rightarrow \text{bound}^t \geq i)$ and $\text{bound}^t \geq 0$

COROLLARY. $\text{bound}^t = \max(\{i : b_i^t \in \{\text{full}, \text{lowering}\}\} \cup \{0\}) \quad \forall t$

COROLLARY. `GetProbeBound(h)` returns $\max(\{i : b_i^{t_6} \in \{\text{full}, \text{lowering}\}\} \cup \{0\})$

LEMMA 7. If bucket i has a *bound raising* transition at time t , $\text{bound}^t \geq i$.

LEMMA 8. If bucket i has a *bound lowering* transition at time t , $\text{bound}^t \neq i$.

PROOFS. Both lemmas follow immediately from the pre-conditions of the state transitions.

LEMMA 9. Bucket j cannot remain in raising state at time t if $\text{bound}^{t-} \neq \text{bound}^t$ and $\text{bound}^t > j$.

LEMMA 10. Bucket j cannot remain in lowering state at time t if $\text{bound}^{t-} \neq \text{bound}^t$ and $\text{bound}^t \neq j$.

PROOFS. Both lemmas follow immediately from the pre-conditions of the state transitions and the First Scanning Lemma.

POST-CONDITION ON LINE 12. $\text{bound}^{t_{12}} \geq \text{bound}^{t_{12}-}$

LEMMA 12. If a call to `ConditionallyRaiseBound(h, i)` alters bound at line 12 and the conditions of the Theorem hold at time $t_{12}-$, they hold at time t_{12} .

PROOF. By Lemmas 7 and 8, we need only consider buckets that do not undergo a *bound raising* or *bound lowering* transition at time t_{12} . By Lemmas 9 and 10, this leaves all buckets in empty or full state, and any bucket $j > \text{bound}^{t_{12}}$ in raising state.

$b_j^{t_{12}-} \in \{\text{raising, full}\}$ by the precondition on `ConditionallyRaiseBound(h,i)`, so any bucket j in empty state satisfies $j \neq \text{bound}^{t_{12}}$. If $b_j^{t_{12}-} = \text{full}$, then by hypothesis $\text{bound}^{t_{12}-} > j$, so by the post-condition on line 12, $\text{bound}^{t_{12}} > j$. Finally, any j where $b_j^{t_{12}-} = b_j^{t_{12}} = \text{raising}$ satisfies $j > \text{bound}^{t_{12}}$ as already stated.

Hence the conditions of the Theorem hold at time t_{12} .

THIRD SCANNING LEMMA. If $\langle \text{bound}, \text{scanning} \rangle^{t_{22}-} = \langle \text{index}, \text{true} \rangle$, no buckets have undergone either *bound raising* or *bound lowering* transitions on the interval $[t_{19}, t_{22}]$.

PROOF. Consequence of exclusivity condition, Second Scanning Lemma and pre-conditions of the state transitions.

POST-CONDITIONS ON LINE 22. If the CAS succeeds, $\text{bound}^{t_{22}} < \text{bound}^{t_{22}-}, \forall j \in (\text{bound}^{t_{22}}, \text{bound}^{t_{22}-}], \exists t \in [t_{19}, t_{22}]$ s.t. $b_j^t \in \{\text{lowering, empty}\}$ and $\text{bound}^{t_{22}} = i > 0 \Rightarrow b_i^{t_{20}} \in \{\text{raising, full}\}$.

LEMMA 15. If a call to `ConditionallyLowerBound(h,i)` alters bound at line 22 at time t_{22} and the conditions of the Theorem hold at time t_{22-} , they hold at time t_{22} .

PROOF. By Lemmas 7 and 8, we need only consider buckets that do not undergo a *bound raising* or *bound lowering* transition at time t_{22} . By Lemmas 9 and 10, this leaves all buckets in empty or full state, and any bucket $j > \text{bound}^{t_{22}}$ in raising state.

By the Third Scanning Lemma, any bucket in empty or full state at time t_{22} must have been so at time t_{19} , so by the post-conditions on line 22 and by hypothesis, $\forall i, b_i^{t_{22}} = \text{empty} \Rightarrow \text{bound}^{t_{22}} \neq i$ and $b_i^{t_{22}} = \text{full} \Rightarrow \text{bound}^{t_{22}} \geq i$. Finally, any j where $b_j^{t_{22}-} = b_j^{t_{22}} = \text{raising}$ satisfies $j > \text{bound}^{t_{22}}$ as already stated.

Hence the conditions of the Theorem hold at time t_{22} .

PROOF OF THEOREM. By construction, the Theorem holds at time 0. We proceed by induction on the steps taken by each thread under some global ordering.

Suppose a thread executes an operation at time t' , and the Theorem holds $\forall t < t'$. The theorem can only be false at time t' if a bucket i undergoes a *bound raising* or *bound lowering* transition, or if $\text{bound}^{t'-} \neq \text{bound}^{t'}$. By Lemmas 7 and 8, neither state transition will invalidate the theorem. The bound field is only altered by lines 12 and 22, and by Lemmas 12 and 15, neither line will invalidate the theorem.

Hence the Theorem holds at time t' , and by induction for all t .

A.2 Shared Bound: Proof of Progress

`GetProbeBound` is trivially lock-free. We show that the remaining two functions in Figure 3 are lock-free, assuming correct behaviour by the calling code, using an amortization argument.

Inspection of the pseudocode reveals that failed CASes either result in local progress (lines 16 and 18) or result from a concurrent, successful CAS during a loop (lines 9–12 and 18–22); we therefore ignore failed CASes.

The scanning bit is assigned a credit of one when clear, and zero when set; lines 12, 16 and 22, when successful, may clear the scanning bit, and hence have an amortized cost of 2. Line 18 has no amortized cost when successful, as it always sets the scanning bit, and hence can be charged against the scanning bit's credit.

CLAIM. `ConditionallyRaiseBound(h,i)` and `ConditionallyLowerBound(h,i)` have amortized costs of at most 2 or 4 successful CAS operations, respectively, provided the code calling `ConditionallyLowerBound(h,i)` ensures $b_i^t \in \{\text{lowering, empty}\} \forall t \in [t_c, t_r]$.

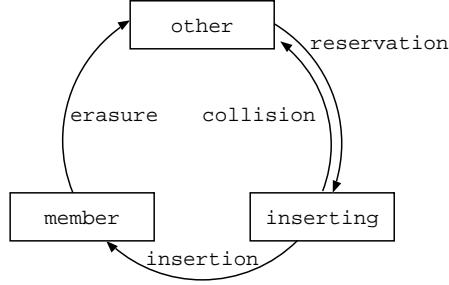


Figure 13: Uniqueness state-machine

LEMMA. The CAS at line 22 will succeed at most once per call.

PROOF. Let t^* be the time of the first successful CAS at line 22; this CAS must trigger the *bound lowering* state transition. Hence, by the Theorem above, $\text{bound}_t \neq i \forall t \in [t^*, t_r]$, and by inspection the loop of lines 18–22 will subsequently terminate without repeating.

PROOF OF CLAIM. `ConditionallyRaiseBound(h,i)` executes line 12 successfully exactly once, and hence has an amortized cost of 2. By inspection and the Lemma, `ConditionallyLowerBound(h,i)` will execute lines 16 and 22 successfully at most once, and hence has an amortized cost of no more than 4.

A.3 Sets: Proof of Uniqueness

For simplicity, we consider a single key k . We model the probe sequence for bucket $h = \text{Hash}(k)$ at time t as an infinite sequence $(s_i^t)_{i \geq 0}$ of state machines, as shown in Figure 13, where $s_i^t \in \{\text{inserting}, \text{member}, \text{other}\} \forall i, t$ and $s_i^0 = \text{other} \forall i$.

In both obstruction-free and lock-free sets, b_j = either `inserting` or `member` if $\text{bound}_h \geq j$, bucket j holds key k and is in `inserting` or `member` state respectively, and b_j = `other` in all other cases.

UNIQUENESS CRITERION. $|\{i : b_i^t = \text{member}\}| \leq 1 \forall t$

Our correctness criterion relies on a single condition:

COLLISION CONDITION. If $\exists i, u, v (u < v)$ such that $s_i^{u-} = \text{other}$, $s_i^t = \text{inserting} \forall t \in [u, v)$ and $s_i^v = \text{member}$, then $\forall j \neq i, \exists t \in [u, v) \text{ with } s_j^t = \text{other}$.

PROOF OF CRITERION. Suppose $\exists i, j, y$ with $i \neq j$ and $s_i^y = s_j^y = \text{member}$. The state-machine and starting conditions imply that $\exists u, v, w, x$ with $s_i^{u-} = \text{other}$, $s_i^t = \text{inserting} \forall t \in [u, v)$, $s_i^t = \text{member} \forall t \in [v, y]$, $s_j^{w-} = \text{other}$, $s_j^t = \text{inserting} \forall t \in [w, x)$ and $s_j^t = \text{member} \forall t \in [x, y]$. Without loss of generality, suppose $u \geq w$. Then by the collision condition, $\exists t \in [u, v)$ with $s_j^t = \text{other}$, but $t \geq u \geq w$ and we derive a contradiction.

Hence the uniqueness criterion holds.

In fact, this proof can be extended to prove a stronger claim:

UNIQUENESS LEMMA. $\forall i, j, u, v (i \neq j, u \leq v), s_i^u = \text{member} = s_j^v \Rightarrow \exists t \in (u, v) \text{ with } s_i^t \neq \text{member} \text{ and } s_j^t \neq \text{member}$.

LOOKUP LEMMA. $\forall u, v, (\forall i, \exists t \in (u, v) \text{ s.t. } s_i^t \neq \text{member}) \Rightarrow \exists t \in (u, v) \text{ s.t. } n_t = 0$.

PROOF. By the uniqueness lemma, $n_t = 1 \forall t \in (u, v) \Rightarrow \exists i \text{ s.t. } s_i^t = \text{member} \forall t \in (u, v)$. The result follows.

A.4 Obstruction-Free Set: Proof of Correctness

We wish to show the pseudo-code in Figures 3 and 6 maintain a logical set, S , of keys. For simplicity, we consider a single key k .

DEFINITIONS.

$$n_t = |\{i : s_i^t = \text{member}\}|$$

$$k \in S_t \iff n_t = 1$$

$$I' = \{t : k \notin S_{t-}, k \in S_t\}$$

$$E' = \{t : k \in S_{t-}, k \notin S_t\}$$

$$L_s = \{\text{Calls to } \text{Lookup}(k) \text{ that return true}\}$$

$$L_f = \{\text{Calls to } \text{Lookup}(k) \text{ that return false}\}$$

$$I_s = \{\text{Calls to } \text{Insert}(k) \text{ that return true}\}$$

$$I_f = \{\text{Calls to } \text{Insert}(k) \text{ that return false}\}$$

$$E_s = \{\text{Calls to } \text{Erase}(k) \text{ that return true}\}$$

$$E_f = \{\text{Calls to } \text{Erase}(k) \text{ that return false}\}$$

If x is a function call, t_c^x is the time it was called, t_r^x the time it returns, and t_n^x the last time it executed line n .

The code is correct iff it satisfies the following criteria:

LOOKUP CRITERION. $\forall x \in L_s, \exists t \in [t_c^x, t_r^x] \text{ s.t. } k \in S_t. \forall x \in L_f, \exists t \in [t_c^x, t_r^x] \text{ s.t. } k \notin S_t.$

INSERTION CRITERION. $\exists f : I_s \rightarrow I', f \text{ a bijection, with } f(x) \in [t_c^x, t_r^x], k \notin S_{f(x)-} \text{ and } k \in S_{f(x)} \forall x \in I_s. \forall x \in I_f, \exists t \in [t_c^x, t_r^x] \text{ s.t. } k \in S_t.$

ERASURE CRITERION. $\exists g : E_s \rightarrow E', g \text{ a bijection, with } g(x) \in [t_c^x, t_r^x], k \in S_{g(x)-} \text{ and } k \notin S_{g(x)} \forall x \in E_s. \forall x \in E_f, \exists t \in [t_c^x, t_r^x] \text{ s.t. } k \notin S_t.$

BOUND LEMMA. $\text{bound}^t = i \Rightarrow s_j^t = \text{other } \forall j > i$

PROOF. Immediate consequence of definitions.

CLAIM. The lookup criterion holds.

PROOF. $\text{Lookup}(k)$ returns true only if $\exists t, i (t \in [t_c, t_r])$ s.t. $s_i^t = \text{member}$, which by the uniqueness criterion implies $n_t = 1$. Appealing to the bound lemma, $\text{Lookup}(k)$ returns false only if $\forall i, \exists t \in [t_c, t_r]$ s.t. $s_i^t \neq \text{member}$, which by the lookup lemma implies $\exists t \in [t_c, t_r]$ s.t. $n_t = 0$.

INSERT LEMMA. inserting \rightarrow member state transitions occur only at line 61, at time t_{61}

PRE-CONDITION ON LINE 57. $s_j^{t_{56}} = \text{member}$.

PRE-CONDITIONS ON LINE 62. $s_i^{t_{61}-} \neq \text{member}, s_i^{t_{61}} = \text{member}$ and $\forall j \neq i, \exists t \in [t_{49}, t_{61}]$ s.t. $s_i^t \neq \text{member}$.

PROOFS. Inspection of the pseudo-code, and appeal to the bound lemma.

CLAIM. The insertion criterion holds.

PROOF. $\text{Insert}(k)$ returns after passing through either lines 57–60, returning false, or line 62, returning true. The pre-condition on line 57 implies $n_{t_{56}} = 1$. The pre-conditions on line 62 satisfy the collision condition, and hence by the uniqueness lemma, $n_{t_{61}-} = 0$ and $n_{t_{61}} = 1$. Further, only one thread can succeed in the CAS at line 61 at time t_{61} for bucket i , and no other threads can succeed at that time for any other bucket by the uniqueness lemma; hence if f maps $x \in I_s$ to t_{61}^x , f is an injection. By the insert lemma, f is also a surjection, and hence a bijection from I_s to I' with the desired properties.

ERASE LEMMA. member → other state transitions occur only at line 68, at time t_{68} .

PRE-CONDITIONS ON LINE 69. $s_i^{t_{68}-} = \text{member}$ and $s_i^{t_{68}} \neq \text{member}$.

PRE-CONDITION ON LINE 72. $\forall i, \exists t \in [t_{65}, t_{72}]$ s.t. $s_i^t \neq \text{member}$.

CLAIM. The erasure criterion holds.

PROOF. `Erase(k)` returns after passing through either lines 69–71, returning true, or line 72, returning false. By the lookup lemma, the pre-condition on line 62 implies $\exists t \in [t_{65}, t_{72}]$ s.t. $n_t = 0$. The pre-conditions on line 69 satisfy the requirements of the erasure condition. Further, only one thread can succeed in the CAS at line 68 at time t_{68} for bucket i , and no other threads can succeed for any other bucket at that time by the uniqueness lemma; hence if g maps a successful `Erase(k)` to its t_{68} , g is an injection. By the erase lemma, g is also a surjection, and hence a bijection from J to J' as desired.

A.5 Obstruction-Free Set: Proof of Progress

Both `Lookup` and `Erase` are lock-free, and hence obstruction-free, as the bounds returned by `GetProbeBound` cannot exceed the largest index of any bucket in the table. `Insert(k)` only repeats lines 48–61 if the ultimate CAS fails, which only occurs if the value written at line 49 is concurrently altered. In isolation the loop will terminate, making the function obstruction-free.

A.6 Lock-Free Set: Proof of Correctness

The proof of correctness for the lock-free set is identical in structure to that of the obstruction-free set, and will not be duplicated.

A.7 Lock-Free Set: Proof of Progress

DEFINITIONS.

$$\text{Let } e_t = \begin{cases} -1 & \text{if } \exists j \text{ s.t. } s_j^t = \text{member} \\ \infty & \text{if } \forall j, s_j^t = \text{other} \\ \min \{j : s_j^t = \text{inserting}\} & \text{otherwise} \end{cases}$$

Let (t_i) be the increasing sequence s.t. $\{t_i\} = \{t : e_t \neq e_{t-}\}$; let $e_i = e_{t_i}$ for brevity.

If $e_i \in [0, \infty)$, let T_i be the last thread that put cell e_i into inserting state by time t_i .

Thread T running `Insert(k)` is said to *abort after time t* if $t \in [t_{76}, t_r]$ and T returns failure.

PRECEDENCE LEMMA. $\exists i, u, v (u < v)$ s.t. $s_i^u = \text{other}$, $s_i^t = \text{inserting } \forall t \in [u, v]$, $s_i^v = \text{other} \Rightarrow \exists j, t \in [u, v]$ s.t. either $s_j^t = \text{member}$ or $s_j^t = \text{inserting}$ and $j < i$.

ASSIST LEMMA. If `Assist(k,...)` returns true, $\forall i \exists t \in [t_c, t_r]$ s.t. $s_i^t = \text{other}$. If `Assist(k,...)` returns false after executing line 76 at t_{76} , either $e_t = -1 \forall t \in [t_{76}, t_r]$, or $\exists t \in [t_{76}, t_r]$ with $e_{t-} \neq e_t$.

PROOF. Inspection of pseudo-code.

SAWTOOTH LEMMA. $0 < e_{i-1} < e_i \Rightarrow e_{i-2} < e_{i-1}$.

PROOF. $0 < e_{i-1} < e_i$ only if $s_{e_i}^{t_i} = \text{other}$ and $s_{e_i}^{t_{i-1}} = \text{inserting}$. By the precedence lemma, $\exists t' < t_i$ s.t. $s_{e_i}^{t'} = \text{inserting } \forall t \in [t', t_i)$, and $\exists j$ s.t. either $s_j^{t'} = \text{member}$ or $j < i$ and $s_j^{t'} = \text{inserting}$. Since $s_{e_i}^{t_i} = \text{inserting} \Rightarrow e_t \leq e_i$ by definition, $s_j^{t'} = \text{member} \Rightarrow e_j = -1$, and $s_j^{t'} = \text{inserting} \Rightarrow e_j < j$, it follows that $e_{t'} < j$. The lemma follows.

INCREASE LEMMA. If $s_i < s_{i+1} < \dots < s_j$, some i, j , then

$$|\{k \in [i, j) : T_k \text{ aborts after } t_k\}| \geq j - i - N, \text{ where } N \text{ is the number of threads.}$$

PROOF. $T_i = T_j$ only if thread T_i completed its first operation and started a new one.

If $e_i < e_{i+1}$, thread T_i can only return from `Insert(k)` after retrying `Assist(k,...)` or failing.

$$|\{T_k : k \in [i, j)\}| \leq N \text{ gives the result.}$$

DECREASE LEMMA. If $s_i > s_{i+1} > \dots > s_j$, some i, j , then

$$|\{k \in [i, j) : T_k \text{ aborts after } t_k\}| \geq j - i - N, \text{ where } N \text{ is the number of threads.}$$

PROOF. $T_i = T_j$ only if thread T_i completed its first operation and started a new one.

If $e_k < -1 \forall k \in [i, j]$, thread T_i can only return from `Insert(k)` after retrying `Assist(k,...)` or failing. $|\{T_k : k \in [i, j)\}| \leq N$ gives the result.

COST LEMMA. $\left| \left\{ (t, j) : t \in [t_i, t_{i+1}), s_j^{t-} = \text{inserting}, s_j^t = \text{other} \right\} \right| \leq 2N \quad \forall i \text{ where } e_i \neq -1$

PROOF. Consequence of assist lemma: observe that if any thread call `Assist(k,...)`, either $e_t = -1$ for the entire duration, or the value of (e_t) must change, and every state transition `other` → `inserting` must be followed by a call to `Assist(k,...)`.

CLAIM. The pseudo-code in Figure 10 is lock-free.

PROOF. Let p_t be the *progress* at time t . Moving a cell from `inserting` state increments the progress counter, while each call to `Insert(k)` decrements it by $4N^2$. We wish to show that $p_t \leq 0 \forall t$.

By the cost lemma, $p_{t_{i+1}} - p_{t_i} \leq 2N \quad \forall i$. By the sawtooth, increase and decrease lemmas, e_t can change at most $2N$ times before an operation completes. The claim follows.

References

- [1] GAO, H., GROOTE, J. AND HESSELINK, W. Almost Wait-Free Resizable Hashtables In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, April 2004, p.50a.
- [2] FRASER, K. Practical Lock-Freedom. *University of Cambridge Computer Laboratory, Technical Report number 579*, February 2004.
- [3] KNUTH, D. The Art of Computer Programming. Part 3, Sorting and Searching. Addison-Wesley, 1973.
- [4] LAMPORT, L. Concurrent Reading and Writing. In *Communications of the ACM*, 1977, pp.806-811.
- [5] MARTIN, D. AND DAVIS, R. A Scalable Non-Blocking Concurrent Hash Table Implementation with Incremental Rehashing. Unpublished manuscript, 1997.
- [6] MELLOR-CRUMMEY, J. AND SCOTT, M. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. In *ACM Transactions on Computer Systems*, Volume 9, Issue 1, February 1991, pp. 21–65.
- [7] MICHAEL, M. Safe Memory Reclamation for Dynamic Lock-Free Objects using Atomic Reads and Writes. In *Proceedings of the 21st Annual Symposium on Principles of Distributed Computing*, July 2002, pp.21-30.
- [8] MICHAEL, M. High performance dynamic lock-free hash tables and list-based sets In *Proceedings of the 14th Annual Symposium on Parallel Algorithms and Architectures*, August 2002, pp.73-82.
- [9] SHALEV, O. AND SHAVIT, N. Split-Ordered Lists: Lock-free Extensible Hash Tables. In *Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing*, July 2003, pp.102-111.
- [10] SCHERER, W. AND SCOTT, M. Contention Management in Dynamic Software Transactional Memory. In *PODC Workshop on Concurrency and Synchronization in Java Programs*, July 2004, pp.70–79.
- [11] PURCELL, C. AND HARRIS, T. Non-blocking Hashtables with Open Addressing. *University of Cambridge Computer Laboratory, Technical Report number 639*, September 2005.

A Lazy Concurrent List-Based Set Algorithm

Steve Heller¹, Maurice Herlihy², Victor Luchangco¹, Mark Moir¹, William N. Scherer III³, and Nir Shavit¹

¹ Sun Microsystems Laboratories

² Brown University

³ University of Rochester

Abstract. List-based implementations of sets are a fundamental building block of many concurrent algorithms. A skip list based on the lock-free list-based set algorithm of Michael will be included in the Java™ Concurrency Package of *JDK 1.6.0*. However, Michael’s lock-free algorithm has several drawbacks, most notably that it requires all list traversal operations, including membership tests, to perform cleanup operations of logically removed nodes, and that it uses the equivalent of an atomically markable reference, a pointer that can be atomically “marked,” which is expensive in some languages and unavailable in others.

We present a novel “lazy” list-based implementation of a concurrent set object. It is based on an optimistic locking scheme for inserts and removes, eliminating the need to use the equivalent of an atomically markable reference. It also has a novel wait-free membership test operation (as opposed to Michael’s lock-free one) that does not need to perform cleanup operations and is more efficient than that of all previous algorithms.

Empirical testing shows that the new lazy-list algorithm consistently outperforms all known algorithms, including Michael’s lock-free algorithm, throughout the concurrency range. At high load, with 90% membership tests, the lazy algorithm is more than twice as fast as Michael’s. This is encouraging given that typical search structure usage patterns include around 90% membership tests. By replacing the lock-free membership test of Michael’s algorithm with our new wait-free one, we achieve an algorithm that slightly outperforms our new lazy-list (though it may not be as efficient in other contexts as it uses Java’s RTTI mechanism to create pointers that can be atomically marked).

1 Introduction

Lists are a fundamental building block for concurrent data structures, both in their own right, and as the basis for many types of search and dictionary data types [12]. We consider three kinds of list operations: inserting a list entry, removing a list entry, and testing whether an entry is in the list.

This paper introduces the *lazy list*, a simple new concurrent *list-based set* algorithm with a number of novel concurrency-related properties. To explain the novel aspects of lazy lists, we start with an overview of different ways to synchronize lists. *Coarse-grained* locking, which uses a single lock to protect the entire

list, has the advantage of simplicity, but provides no concurrency. With *lock coupling* (sometimes called “hand-over-hand” locking) [1], a thread acquires the lock for each successive entry before releasing the lock for its predecessor. Lock coupling provides more concurrency than coarse-grained locking, but threads may acquire many successive locks, which is undesirable because lock acquisition typically involves expensive atomic operations (such as compare-and-swap). Moreover, concurrent threads moving through the list may contend for locks even if they are searching for unrelated list entries. Valois [14] was the first to suggest a non-blocking implementation of a concurrent list-based set. Harris [3] and later Michael [10], presented highly efficient lock-free algorithms for list-based sets. Fomitchev and Ruppert [10] present more complex algorithms that provide an amortized cost guarantee for all operations that is provably linear in the length of the list. Michael’s algorithm is the basis for a concurrent skip-list data structure in the Java™ Concurrency Package of JDK 1.6.0.

As in most previous list-based set algorithms, we represent a set as a sorted linked list. In our new lazy list algorithm, insertion and removal operations are *optimistic*: each operation searches the list without acquiring any locks or interfering with other threads. When an operation locates the entry it is seeking it locks that entry and its predecessor and checks for synchronization conflicts. If no conflict is detected, an entry is inserted or removed, and otherwise the locks are released and the operation is restarted.

This optimistic approach to insertion and removal has the advantage that insert and remove calls that access non-adjacent list entries never interfere. In the absence of synchronization conflicts, these operations acquire only a constant number of locks. Entries are removed from the list in a *lazy* manner: the entry is first marked as removed (the “logical” removal), and then it is physically unlinked from the list (the “physical” removal). The simplifying power of lazy techniques has been exploited by Harris [3] and Michael [10] for concurrent lists, and by Maier [9] in more general contexts. Nevertheless, the algorithms of Harris and Michael require the ability to perform an atomic compare-and-swap on two fields at once: a Boolean marked field and a reference field to the next entry in the list (the equivalent of an `AtomicMarkableReference` in the Java Programming Language). Since in many systems it is unacceptable to “steal a bit” from a reference, one must use alternative techniques. In modern object oriented languages, one can have two trivial (empty) subclasses of a node object and use a *run time type identification* (RTTI) mechanism [2] to determine which subclass the current instance belongs to, where each subclass represents a state of the bit. In languages without RTTI support, one can use an additional level of indirection, adding a pointer to a special dummy node to signify that the bit is set. This is the mechanism used to implement `AtomicMarkableReference` in the Java Concurrency Package, which unfortunately can introduce significant performance penalties.

Perhaps the most substantial advantage of the new algorithm is that membership test operations are *wait-free* [4]. The lock-freedom progress property of the membership test in Michael’s algorithm guarantees that if some threads are

executing method calls, and at least one thread continues taking steps, then at least one thread will complete its call, but makes no progress guarantee for any individual thread. Wait-freedom is a stronger progress property that guarantees that any thread that continues taking steps in executing a method call, will eventually complete the call.

The membership test of our algorithm acquires no locks, requires no synchronization, and never interferes with any concurrent operations. This last property is particularly important because it is reasonable to expect that in most real-world applications, membership tests are by far the most common operations. In Michael's lock-free list algorithm, and unlike in ours, if a thread traversing the list encounters an entry that has been logically but not physically removed, then the thread must stop to complete the physical removal. Physical removal requires calling a compare-and-swap operation, and if several concurrent threads attempt to remove the same entry, then only one will succeed, and the rest will be forced to abandon their traversals and start over. While the number of such removals is likely to be small, our empirical testing shows that when there is a high level of concurrent traversals, contention among threads competing to perform the removal causes a large number of traversals to be abandoned and restarted.

By contrast, in the new lazy list algorithm, only the remove operations are required to perform physical removals, while the insertion and (more importantly) membership query traversals are not delayed by physical removals. The wait-free nature of the membership operation means that ongoing changes to the list cannot delay even a single thread from deciding membership. We note that our wait-free membership test is of independent value: one can readily replace the membership test in Michael's algorithm with the lazy list's new membership test, allowing it to obtain improved performance by eliminating the need for physical removals.

To evaluate our new lazy list algorithm, we implemented it in the JavaTM programming language and conducted a series of benchmarks comparing our new algorithm to known algorithms on a 16 node SunFireTM 6800 cache coherent bus-based multiprocessor machine. We found that when there is a high fraction of membership tests (as in search structures) the new lazy list algorithm and a new version of Michael's algorithm that uses our wait-free membership test, outperform all others by a factor of two or more. The good performance of our new version of Michael's lock-free list depended on the use of Java's RTTI mechanism. We also found that as the fraction of membership queries dropped, the relative performance advantage of the lazy list disappeared, and the new version of Michael's list with our wait-free membership test showed the best performance.

In summary, we conclude that adding the new wait-free membership test always offers a performance advantage and has no performance penalties. For applications with a high fraction of membership tests, one should definitely use the new algorithms, while the choice of which algorithm to use—the new lazy list, or Michael's lock-free list with our new wait-free membership test—seems to

depend on the cost and availability of mechanisms for implementing the equivalent of `AtomicMarkableReference` in a given system and language.

Following our initial presentation of the algorithms in this paper, a complete formal treatment was provided by Vafeiadis et al in [13]. We therefore focus on providing an informal and easily accessible explanation of why our new algorithm works, and refer the interested reader to [13] for the detailed correctness proofs.

2 The New Algorithm

We present our concurrent linked-list implementation in the context of a list-based set object. For our purposes, a *Set* provides three methods:

- The `add(x)` method adds x to the set, returning *true* if and only if x was not already in the set.
- The `remove(x)` method removes x from the set, returning *true* if and only if x was in the set.
- The `contains(x)` method returns *true* if and only if the set contains x .

For each method, we say that a call is *successful* if it returns *true*, and *unsuccessful* otherwise.

Linearizability [6] is a standard correctness condition for concurrent data structures. The list-based set implementation that we present is a linearizable implementation of a set object. To prove this it is enough to identify, for each method call in each possible execution history, a *linearization point*, a single operation when the method call “takes effect”. For example, the linearization point defines exactly when `add(a)` adds an entry, a point during the execution of the method immediately before which a is not in the set, and immediately after which a is in the set.

Lock-freedom is a progress property that guarantees that if some threads are executing method calls, and at least one thread continues taking steps, then at least one thread will complete its call. It guarantees that the system as a whole continues to make progress, but makes no progress guarantee for any individual thread. *Wait-freedom* is a stronger progress property that guarantees that any thread that continues taking steps in executing a method call, will eventually complete the call.

As noted earlier, following our initial presentation of the algorithms in this paper, a complete formal treatment was provided by Vafeiadis et al in [13]. We therefore focus here on giving an informal and easily readable explanation of why our new algorithm works.

We represent the set as a sorted list of entries. As shown in Figure 1, the `Entry` class has four fields. The `key` field is the set element. Our algorithm works for any ordered set of keys that has maximum and minimum values and is well-founded, that is, for any given key, there are only finitely many smaller keys. This is trivially satisfied by most real-world key types because the size of the `key` is fixed; for simplicity, we present our algorithm assuming that the keys are integers. We will use the well-foundedness assumption to technically capture

```

private class Entry {
    int key;
    Entry next;
    boolean marked;
    lock lock;
}

```

Fig. 1. List entry: an entry keeps track of the set element itself (the key), the next entry in the list, a marked field to denote logical removal of the entry, and a `lock` field for synchronization.

the notion that the progress of a membership query in Michael's algorithm is lock-free while the new algorithm's membership query is wait-free.

The list is maintained in `key` order, providing an efficient way to determine whether a given key is in the list. We sometimes abuse notation slightly and use the same symbol to refer to an entry and its associated key (entry a will have key a and so on.). The `next` field is a reference to the next entry in the list, the `marked` field indicates if its associated key is logically removed or still in the data structure, and the `lock` field is a lock used for synchronization.

We assume that the `add()`, `remove()`, and `contains()` methods are the *only* ones that modify entries, a property sometimes called *freedom from interference*. We require freedom from interference even for entries that have been removed from the list, since a thread may unlink an entry while it is being traversed by others. In a language such as Java, we can rely on the garbage collector to recycle unreachable entries. In a programming language without garbage collection, this property can be maintained by using methods like ROP [5] or SMR [11].

The list has two kinds of entries. In addition to *regular* entries that hold elements (keys) in the set, we use two *sentinel* entries, called `head` and `tail`,

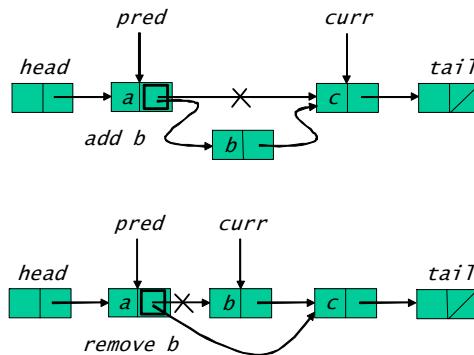


Fig. 2. Insertion and removal of list entries.

```

public boolean remove(int key) {
    while (true) {
        Entry pred = this.head;
        Entry curr = head.next;
        while (curr.key < key) {
            pred = curr; curr = curr.next;
        }
        pred.lock();
        try {
            curr.lock();
            try {
                if (validate(pred, curr)) {
                    if (curr.key != key) { // present
                        return false;
                    } else { // absent
                        curr.marked = true; // logically remove
                        pred.next = curr.next; // physically remove
                        return true;
                    }
                }
            } finally { // always unlock curr
                curr.unlock();
            }
        } finally { // always unlock pred
            pred.unlock();
        }
    }
}

```

Fig. 3. The lazy `remove()` method: removes entries in two steps, logical and physical.

as the first and last list entries. The sentinel entries contain the minimum and maximum key values, respectively; we assume that these values are never added, removed or searched for. Ignoring the details of synchronization for the moment, the top part of Figure 2 shows a schematic description of how a key is added to the set. Each thread has two local variables used to traverse down the list: `curr` is the current entry and `pred` is its predecessor.

To add a new key to the set, a thread sets the local variable `pred` to `head` and `curr` to `head`'s successor, and moves down the list, comparing `curr`'s key to the key being added. If they match, the key is already present in the set, so the thread returns *false*. If `pred` precedes `curr` in the list, `pred`'s key is lower than the inserted key, and `curr`'s key is higher, then the key is not present in the list. Therefore, the thread creates a new entry *b* to hold the key, sets *b* to point to `curr`, and then sets `pred` to point to *b*. The key is now a member of the set.

```

private boolean validate(Entry pred, Entry curr) {
    return !pred.marked && !curr.marked && pred.next == curr;
}

```

Fig. 4. The lazy lists validation.

Removing a key is similar: we scan the list to find the relevant adjacent pair of entries. The target entry is removed from the list in two steps: first, its `marked` field is set to *true*, indicating that the entry has been *logically* removed from the list, and second, the predecessor entry’s `next` field is redirected to point to the successor entry, *physically* removing the entry from the list. As discussed more precisely later, the removal “actually happens” when an entry is marked, and the physical removal is just a way to clean up.

2.1 The `remove()` method

As shown in Figure 3, when the `remove()` method attempts to remove the entry with key k , it scans through the list without acquiring any locks, traversing both marked and unmarked entries. The `remove()` method uses two local variables: `curr` is the current entry and `pred` is its predecessor. When `curr` is set to the first entry with a key greater than or equal to k , the traversal stops, and the method locks `curr` and `pred`. Because there is a gap between the unsynchronized traversal and the lock acquisition, it is necessary to *validate* that the method has locked the correct entries. What can go wrong? There are three obvious problems: the `curr` entry could have been removed, the `pred` entry could have been removed, or another entry may have been inserted between `pred` and `curr`. Surprisingly, perhaps, these are the *only* things that can go wrong, and moreover, they can be detected very efficiently. It is enough to check that `curr` and `pred` are both unmarked, and that `pred`’s `next` pointer points to `curr` (see Figure 4). If these conditions hold, the entries are adjacent and present in the list. If the validation succeeds, the `remove()` method logically removes the entry, physically removes the entry, releases both locks and returns *true*. If the entry with key k is absent, the method unlocks the entries and returns *false*. If the validation fails, the thread restarts the method.

For an unsuccessful `remove()` call, the linearization point is the point at which it finds (reads the pointer to) a marked entry with the same key or the first unmarked entry with a larger key. For a successful `remove()` method call, the linearization point is the moment the entry is marked (line LR of Figure 3).

2.2 List traversal

We pause momentarily to discuss list traversal. The list traversal in the `remove()` method in Figure 3 seems straightforward: simply follow the list pointers. The same approach is used in the `add()` and `contains()` methods. It is important

to note that this traversal differs from those of other concurrent list-based set algorithms in the literature in two important ways:

- it requires no additional synchronization (such as acquiring locks [1] or cleaning up logically removed nodes [10]), and
- it traverses both logically and physically removed nodes.

This latter property, which allows us to achieve the former, is the key to our algorithm’s good performance. Figure 7 shows how a concurrent physical removal of a node during thread A ’s traversal can cause it to traverse a physically removed part of the list. The traversal works correctly because we assume the freedom from interference property which implies that nodes, even if they are removed from the list, are not recycled (freed back to the available memory pool) as long as they are reachable. Thus, if a node is removed while it is being traversed, the traversing thread will continue to follow the list of pointers and eventually reach its target node. Our algorithm maintains the property that if an entry was in the list when a given thread started searching for it, it will remain reachable from this thread’s `curr` pointer as long as it is not removed.

2.3 The `add()` method

Like the `remove()` method, the `add()` method (Figure 5) scans the list without acquiring locks, until `curr` is set to the first entry with a key greater than or equal to the key to be inserted. The method locks both entries, validates them, and if an entry with the specified key is not already present in the list, inserts a new entry, unlocks the entries, and returns `true`. The remaining cases are just as in the `remove()` method. For an unsuccessful `add()` method call, the linearization point is the moment at which the entry is observed to be unmarked in the list. For a successful `add()` method call, it is the moment when `pred.next` is set. We note that one can make the `add()` method more efficient by locking only the `pred` node, but for the sake of keeping our algorithm simple, we omit this optimization here.

2.4 The wait-free `contains()` method

The key to the performance of our algorithm is the new wait-free `contains()` method. This method is of independent interest. For example, we show in Section 3 that it can readily replace the lock-free `contains()` method in the algorithm of Michael [10] to provide improved performance.

The `contains()` method scans the list, just like the `remove()` and `add()` methods, ignoring whether nodes are marked or not, until `curr` is set to the first entry with a key greater than or equal to the sought-after key. Instead of locking the entry, however, it simply returns `true` if and only if the `curr` entry is unmarked with the desired key. This is correct since the list is ordered and so, if a node is removed, it must be marked or not present in the list.

It is easy to see that this method is wait-free. First, notice that because the universe of keys is well-founded there are only a finite number of keys that are

```

public boolean add(int key) {
    while (true) {
        Entry pred = this.head;
        Entry curr = head.next;
        while (curr.key < key) {
            pred = curr; curr = curr.next;
        }
        pred.lock();
        try {
            curr.lock();
            try {
                if (validate(pred, curr)) {
                    if (curr.key == key) { // present
                        return false;
                    } else { // not present
                        Entry entry = new Entry(key);
                        entry.next = curr;
                        pred.next = entry;
                        return true;
                    }
                }
            } finally { // always unlock
                curr.unlock();
            }
        } finally { // always unlock
            pred.unlock();
        }
    }
}

```

Fig. 5. The `add()` method.

```

public boolean contains(int key) {
    Entry curr = this.head;
    while (curr.key < key)
        curr = curr.next;
    return curr.key == key && !curr.marked;
}

```

Fig. 6. The lazy list's wait-free `contains()` method.

smaller than the one being searched for. According to the algorithm, entries with lower or equal keys to a given entry will never be added ahead of it (i.e. so that they are reachable from it) even if the entry points into the list but is logically and physically removed from the list. Thus, each time the traversal moves to a

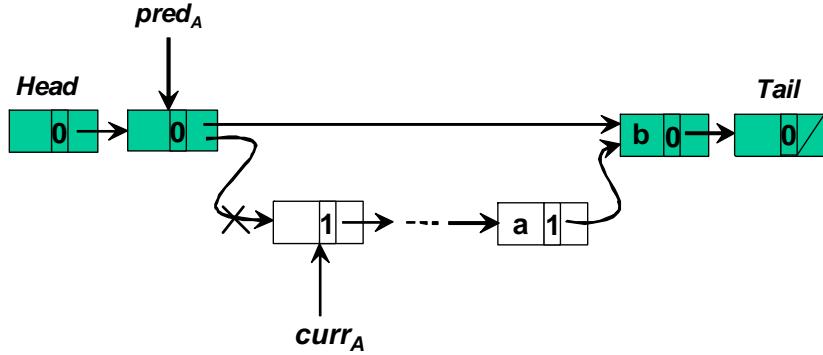


Fig. 7. Linearizing an unsuccessful `contains()` method call is a bit tricky. Dark nodes are physically in the list and white nodes are physically removed. During a traversal of the list by thread A , the sublist starting at the node pointed to by curr (and schematically represented by “...”) may be disconnected from the main list by a concurrent `remove()` method execution. Both nodes with items a and b can still be reached, and the determination if an item is in the list is based solely on the mark-bit.

new node, the new node has a larger key than the previous one, and this can happen only finitely many times, which implies that traversal is wait-free. This contrasts with Michael’s membership test [10] which is only lock-free [4], since it can be forced to restart its traversal from the beginning of the list infinitely often if the same item is re-inserted and removed, and it fails each time when attempting to clean it up.

A successful `contains()` method call is linearized when the marked field of a matching entry is observed to be false. Linearizing an unsuccessful `contains()` method call is a bit tricky, and is a good example showing that it is not always possible to define a single linearization point for each method that works for all method calls in all executions. In particular, simply choosing the linearization point for an unsuccessful `contains()` as the point at which a marked entry with the sought-after key or an entry greater than the sought-after key is found is incorrect. Consider the following scenario. Assume that entry a is marked and thread A is attempting to find the entry matching a ’s key. While A is traversing the list, curr_A and all entries between curr_A and a including a are removed logically and physically. Thread A would still proceed to the point where curr_A points to a . It would then detect that a is marked and therefore no longer in the list. Linearizing at this point is correct in this case. However, consider what happens if while thread A is traversing the removed section of the list leading to a , and before it reaches the removed a , another thread adds a new entry with a key a to the reachable part of the list. Linearizing the unsuccessful `contains()` method at the point at which it observed the marked entry a would be wrong, since it occurs *after* the insertion of the new entry with key a to the list.

We therefore linearize an unsuccessful `contains()` method call within its execution interval at the earlier of the following points: (1) the point where a removed matching entry is found and (2) the point immediately before a new matching entry is added to the list. As can be seen, this linearization point is determined by the ordering of events in the execution, and not predetermined as a specific point in the method execution.

3 Performance

We evaluated our new algorithm on a SunFire™ 6800 cache coherent bus-based multiprocessor machine with 16 1.2 GHz processors. The algorithms were implemented in Java 1.5.0. We varied the percentage of `contains()` method calls and the percentage of `add()` and `remove()` method calls. Each thread randomly selected both the type of call to make (respecting the given percentages) and the operand for it; operands are integers in the range 0..1023. We repeated this test suite both with and without an additional load of 16 threads performing computation in order to evaluate the sensitivity of our results to background load, but do not report the additional load tests here as there were no significant differences noted. In all our benchmarks, we measured *throughput*: the total number of calls completed over the course of 8 seconds, averaged across three runs. We tested six different list algorithms in all.

- *Coarse* – We use a single `java.util.concurrent.ReentrantLock`s lock to protect all access to the list.
- *Fine* – This is a *fine grained* hand-over-hand locking (lock-coupling) [8, 1] list-based implementation using a lock per list entry. Threads traverse down the list holding multiple locks at a time, releasing the earlier acquired entry’s lock only after acquiring the next one in the list.
- *LockFree* – This is a *lock-free* list implemented according to Michael’s algorithm [10], using the `AtomicMarkableReference` of JDK 1.5.0 to allow a markable next pointer per entry. As in our algorithm, the mark is used to denote that an entry is logically removed. Unlike in our algorithm, the `contains()` method is lock-free and not wait-free as calls do not traverse marked entries, instead, they clean them up before continuing traversal down the list.
- *LockFreeRTTI* – This is the lock-free list of Michael’s algorithm [10] using the Java RTTI mechanism to distinguish marked entries. Such mechanisms are not available in all languages. Achieving the effect of marking a bit in the `next` pointer is done more efficiently than with `AtomicMarkableReference` by having two trivial (empty) subclasses of each entry object and using RTTI to determine at runtime which subclass the current instance is, where each subclass represents a state of the mark bit.
- *NewLockFreeRTTI* – This is *LockFreeRTTI* with Michael’s lock-free `contains()` method directly replaced by the new wait-free `contains()` method of this paper, one that does not clean up marked entries and instead traverses them in a wait-free manner.

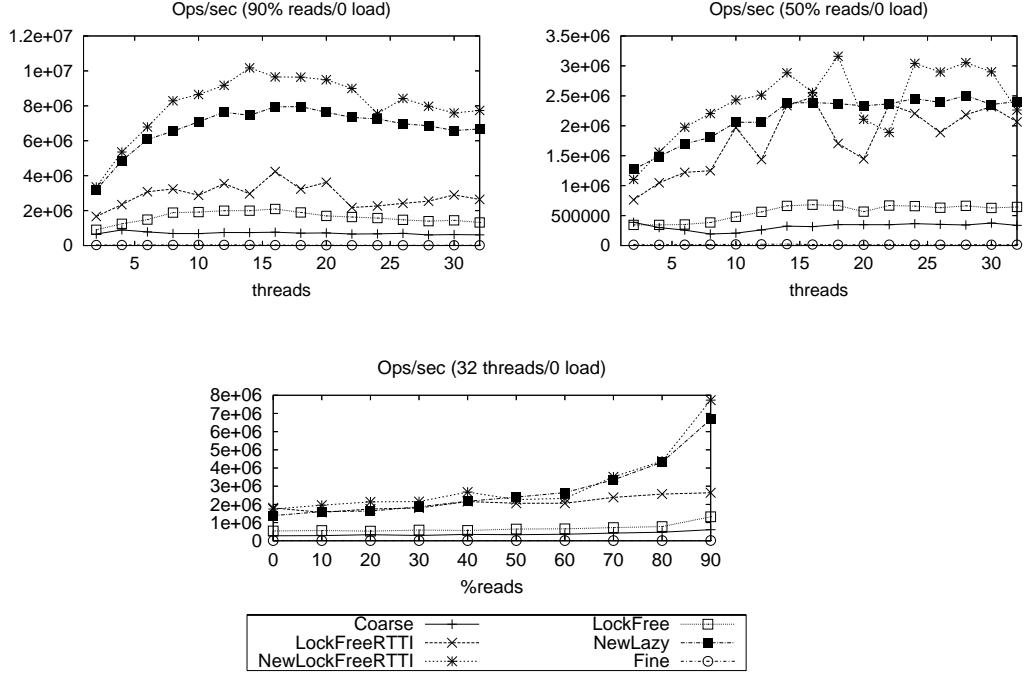


Fig. 8. The top two graphs show the change in throughput as concurrency increases to 32 threads with 60% and 90% of the operations being `contains()` method calls, and a 9/1 ratio of `add()` to `remove()` method calls. The bottom graph shows the change in throughput for the case of 32 threads as the fraction of `contains()` calls increases to 90%

- *NewLazy* – This is the new lazy list algorithm of this paper, with its new wait-free `contains()` and an optimization of the `add()` method to use only a single lock.

The top of Figure 8 shows the results of running a benchmark with 90% `contains()` method calls, 9% `add()` method calls and 1% `remove()` method calls (left) and another benchmark with 50% `contains()` method calls, 45% `add()` method calls and 5% `remove()` method calls (right). The 90/9/1 ratio and the high fraction of `add()` method calls to `remove()` method calls are considered typical of search structures, a common application of linked-lists [7].

If we look at the graph of the 90% test on the lefthand side of Figure 8, we see that the two new algorithms, the lazy list and the new lock-free list with a wait-free `contains()` method, outperform all others by a factor of two or more, including both versions of Michael’s lock-free list, the one implemented with `AtomicMarkableReference` and the one implemented with the RTTI mechanism. The reason for this is as follows: even though there is a very small fraction

of `remove()` method calls, there are many concurrent `contains()` method traversals, and in both of the original versions of Michael's algorithm they all compete to clean up the same small set of logically removed entries. All traversals that fail must restart, leading to a significant overhead. The new version of Michael's algorithm with RTTI and our wait-free `contains()` method performs slightly better than the lock-based lazy list. However, the reader is reminded that many languages do not have the equivalent of RTTI.

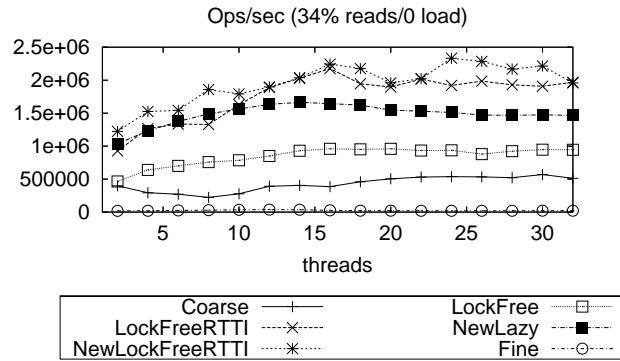


Fig. 9. The graph shows throughput as concurrency increases with a 34%, 33% and 33% ratio respectively of `contains()`, `add()`, and `remove()` method calls.

The graph of the 50% test on the righthand side of Figure 8 shows what happens when we drop the fraction of `contains()` method calls. As can be seen, the lock-free RTTI-based implementation of Michael's algorithm stays at about the same throughput level, yet the performance of the two new algorithms deteriorates because (1) the large number of additional `add()` method calls in the new version of Michael's algorithm incur cleanup contention (they fail attempts at cleaning up the same entries) and must restart their traversals, and (2) the `add()` method calls in the lazy list acquire more costly locks and fail validation at a much higher rate, forcing them to restart their traversals.

The bottom graph of Figure 8 shows the change in throughput for the case of 32 threads as the fraction of `contains()` method calls increases (maintaining the 9/1 ratio of `add()` and `remove()` method calls). As can be seen, from 50% and onward the two new algorithms outperform all others, and have more than twice their throughput at 90%. The choice of which algorithm to use, the new lazy list, or Michael's lock-free list with our new wait-free membership test, for typical search applications with a high fraction of memberships tests, seems to depend on the cost of implementing the equivalent of `AtomicMarkableReference` in a given system and language.

The graph in Figure 9 shows the change in throughput when running a benchmark with 34% `contains()` method calls, 33% `add()` method calls and 33% `remove()` method calls. Though this is not a typical search structure access pattern, we present it here to explore how the algorithms compare across a wider range of loads. As can be seen, the throughput of the lock-free RTTI based implementations drops slightly, and the performance of the lazy list drops more significantly. As before, this is due to the further increase in the number of costly lock acquisitions and of failed validations.

We conclude that even with higher `add()` and `remove()` method call rates than we expect in many applications, our results show how to improve on the performance of previous algorithms. Furthermore, without using any nonstandard language tricks, our new algorithms soundly beat previous ones.

4 Conclusions

We introduced the *lazy list*, a simple new concurrent list algorithm based on lazy marking and deletion of nodes. Perhaps the most substantial advantage of the new algorithm is a *wait-free* membership test operation, an operation that can readily replace membership tests in other list-based set algorithms such as Michael’s lock-free lists [10].

Various optimizations to our algorithm are possible. As noted earlier, one can make the `add()` method more efficient by locking only the `pred` node. One can also add an optimization whereby threads “prevalidate” the state of an entry before acquiring the entry locks, thereby saving the cost of acquiring them upon failure.

Most importantly, we believe the algorithmic approach introduced in this paper, the combination of lazy lock-based list manipulation coupled with wait-free traversal, can lead to simpler and possibly more efficient algorithms for related data structures such as concurrent skip-lists and other search structures.

References

1. R. Bayer and M. Schkolnick. Concurrency of operations on b-trees. *Acta Informatica*, 9:1–21, 1977.
2. B. Eckel. *Thinking in Java (2nd Edition)*. Pearson Education, 2000.
3. T. Harris. A pragmatic implementation of non-blocking linked-lists. *Lecture Notes in Computer Science*, 2180:300–314, 2001.
4. M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.
5. M. Herlihy, V. Luchangco, and M. Moir. The repeat offender problem: A mechanism for supporting lock-free dynamic-sized data structures. In *Proceedings of the 16th International Symposium on DIStributed Computing*, volume 2508, pages 339–353. Springer-Verlag Heidelberg, January 2002. A improved version of this paper is in preparation for journal submission; please contact authors.
6. M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

7. D. Lea. Personal communication.
8. D. Lea. *Concurrent Programming in Java(TM): Design Principles and Patterns*. Addison-Wesley, second edition edition, 1999.
9. Corinne Maier. *Hello Laziness: Why Hard Work Doesn't Pay*. Orion, London, 2005. ISBN: 0752871862.
10. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 73–82. ACM Press, 2002.
11. M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *The 21st Annual ACM Symposium on Principles of Distributed Computing*, pages 21–30. ACM Press, 2002.
12. M. Moir and N. Shavit. *Chapter 47 – Concurrent Data Structures – Handbook of Data Structures and Applications*. Chapman and Hall/CRC, first edition edition, 2004.
13. V. Vafeiadis, M. Herlihy, T. Hoare, and M. Shapiro. Proving correctness of highly-concurrent linearisable objects. Technical report, Microsoft Research, Cambridge, UK, 2005.
14. J. Valois. Lock-free linked lists using compare-and-swap. In *ACM Symposium on Principles of Distributed Computing*, pages 214–222, 1995.

High Performance Dynamic Lock-Free Hash Tables and List-Based Sets

Maged M. Michael

IBM Thomas J. Watson Research Center
P.O. Box 218 Yorktown Heights NY 10598 USA

magedm@us.ibm.com

ABSTRACT

Lock-free (non-blocking) shared data structures promise more robust performance and reliability than conventional lock-based implementations. However, all prior lock-free algorithms for sets and hash tables suffer from serious drawbacks that prevent or limit their use in practice. These drawbacks include size inflexibility, dependence on atomic primitives not supported on any current processor architecture, and dependence on highly-inefficient or blocking memory management techniques.

Building on the results of prior researchers, this paper presents the first CAS-based lock-free list-based set algorithm that is compatible with all lock-free memory management methods. We use it as a building block of an algorithm for lock-free hash tables. In addition to being lock-free, the new algorithm is dynamic, linearizable, and space-efficient.

Our experimental results show that the new algorithm outperforms the best known lock-free as well as lock-based hash table implementations by significant margins, and indicate that it is the algorithm of choice for implementing shared hash tables.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming; D.4.1 [**Operating Systems**]: Process Management—*concurrency, multiprocessing/multiprogramming/multitasking, synchronization*; E.2 [**Data Storage Representations**]: *hash table representations*

General Terms: Algorithms, Performance, Reliability

1. INTRODUCTION

The hash table is a ubiquitous data structure widely used in system programs and applications as a search structure. Its appeal lies in its guarantee of completing each operation in expected constant time, with the proper choice of a hashing function and assuming a constant load factor [3, 12].

To ensure correctness, concurrent access to shared objects must be synchronized. The most common synchronization method is the use of mutual exclusion locks. However, lock-

based shared objects suffer significant performance degradation when faced with the inopportune delay of a thread while holding a lock, for instance due to preemption. While the lock holder is delayed, other active threads that need access to the locked shared object are prevented from making progress until the lock is released by the delayed thread.

A lock-free (also called non-blocking) implementation of a shared object guarantees that if there is an active thread trying to perform an operation on the object, some operation, by the same or another thread, will complete within a finite number of steps regardless of other threads' actions [8]. Lock-free objects are inherently immune to priority inversion and deadlock, and offer robust performance, even with indefinite thread delays and failures.

Shared sets (also called dictionaries) are the building blocks of hash table buckets. Several algorithms for lock-free set implementations have been proposed. However, all suffer from serious drawbacks that prevent or limit their use in practice.

Lock-free set algorithms fall into two main categories: array-based and list-based. Known array-based lock-free set algorithms [5, 13] are generally impractical. In addition to restricting maximum set size inherently, they do not provide mechanisms for preventing duplicate keys from occupying multiple array elements, thus limiting the maximum set size even more, and requiring excessive overallocation in order to guarantee lower bounds on maximum set sizes.

Prior list-based lock-free set algorithms involve one or more serious problems: dependence on the DCAS (double-compare-and-swap)¹ atomic primitive that is not supported on any current processor architecture [5, 14], susceptibility to livelock [25], and/or dependence on problematic memory management methods [6, 14, 25] (i.e., memory management methods that are impractical, very inefficient, blocking (not lock-free), and/or dependent on special operating system support).

The use of universal lock-free methodologies [1, 2, 8, 11, 22, 24] for implementing hash tables or sets in general is too inefficient to be practical.

This paper presents a lock-free list-based set algorithm that we use as a building block of a lock-free hash table algorithm. The algorithm is dynamic, allowing the object size and memory use to grow and shrink arbitrarily. It satisfies the linearizability correctness condition [9].

It uses CAS (compare-and-swap) or equivalently restricted LL/SC (load-linked/store-conditional). CAS takes three arguments: the address of a memory location, an expected

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SPAA'02, August 10-13, 2002, Winnipeg, Manitoba, Canada.
Copyright 2002 ACM 1-58113-529-7/02/0008 ...\$5.00.

¹DCAS takes six arguments: the addresses of two independent memory locations, two expected values and two new values. If both memory locations hold the corresponding expected values, they are assigned the corresponding new values atomically. A Boolean return value indicates whether the replacements occurred.

value and a new value. If the memory location holds the expected value, it is assigned the new value atomically. A Boolean return value indicates whether the replacement occurred. SC takes two arguments: the address of a memory location and a new value. If no other thread has written the memory location since the current thread last read it using LL, it is assigned the new value atomically. A Boolean return value indicates whether the replacement occurred. All architectures that support LL/SC restrict some or all memory accesses between LL and SC, and allow SC to fail spuriously. All current major processor architectures support one of these two primitives.

This algorithm is the first CAS-based list-based set or hash table algorithm that is compatible with simple and efficient methods [10, 17], as well as all other memory management methods for lock-free objects.

Our experimental results show significant performance advantages of the new algorithm over the best known lock-free as well as lock-based hash table implementations. The new algorithm outperforms the best known lock-free algorithm [6] by a factor of 2.5 or more, in all lock-free cases. It outperforms the best lock-based implementations, under high and low contention, with and without multiprogramming, often by significant margins.

In Section 2 we review prior lock-free algorithms for sets and hash tables and relevant memory management methods. In Section 3 we present the new algorithm. In Section 4 we discuss its correctness. In Section 5 we present performance results relative to other hash table implementations. We conclude with Section 6.

2. BACKGROUND

2.1 The Hash Table Data Structure

A hash table is a space efficient representation of a set object K when the size of the universe of keys U that can belong to K is much larger than the average size of K . The most common method of resolving collisions between multiple distinct keys in K that hash to the same hash value h is to chain nodes containing the keys (and optional data) into a linked list (also called bucket) pointed to by a head pointer in the array element of the hash table array with index h . The load factor α is the ratio of $|K|$ to m , the number of hash buckets [3, 12].

With a well-chosen hash function $h(k)$ and a constant average α , operations on a hash table are guaranteed to complete in constant time on the average. This bound holds for shared hash tables in the absence of contention.

The basic operations on hash tables are: *Insert*, *Delete* and *Search*. Most commonly, they take a key value as an argument and return a Boolean value. *Insert*(k) checks if nodes with key k are in the bucket headed by the hash table array element of index $h(k)$. If found (i.e., $k \in K$), it returns false. Otherwise it inserts a new node with key k in that bucket and returns true.

Delete(k) also checks the bucket with index $h(k)$ for nodes with key k . If found, it removes the nodes from the list and returns true. Otherwise, it returns false. *Search*(k) returns true if the bucket with index $h(k)$ contains a node with key k , and returns false otherwise.

For time and space efficiency most implementations do not allow multiple nodes with the same key to be present concurrently in the hash table. The simplest way to achieve this is to keep the nodes in each bucket ordered by their key values.

Figure 1 shows a list-based hash table representing a set K of positive integer keys. It has seven buckets and the hash function $h(k) = k \bmod 7$.

By definition, a hash *function* maps each key to one and only one hash value. Therefore, operations on different hash

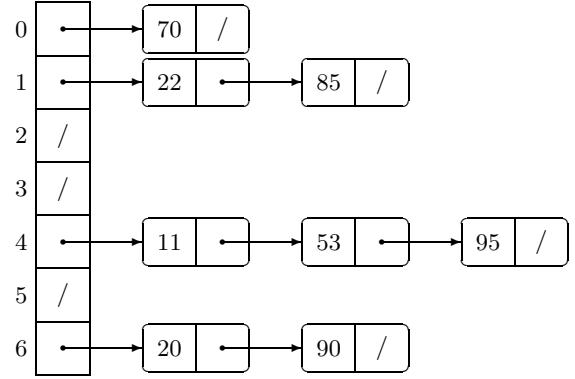


Figure 1: An example of a hash table with 7 buckets and hash function $h(k) = k \bmod 7$.

buckets are inherently disjoint and are obvious candidates for concurrency. Generally, hash table implementations allow concurrent access to different buckets or groups of buckets to proceed without interference. For example if locks are used, different buckets or groups of buckets can be protected by different locks, and operations on different bucket groups can proceed concurrently. Thus, shared set implementations are obvious building blocks of concurrent hash tables.

2.2 Prior Lock-Free Set Algorithms

Array-Based Lock-Free Set Algorithms

Lanin and Shasha [13] presented two array-based set algorithms that can be implemented using CAS. The first algorithm allows Delete and Search operations without locking, but requires Insert operations to use locks, in order to avoid the concurrent insertion of multiple instances of the same key into the hash table. The use of locks precludes the algorithm from being lock-free.

The second algorithm allows duplicate key values to occupy multiple array elements. Delete operations must search for and delete all matching keys in the array. Using arrays to represent sets requires the maximum bucket size to be static. For a hash table the maximum bucket size is too large to allow static allocation, and doing so would eliminate any advantage of hash tables over direct addressing of arrays. Furthermore, even if an upper bound on the bucket set size is known, the array cannot guarantee accommodating that size, without excessive overallocation, as the same key value may occupy multiple array entries. The algorithm is impractical even when a maximum set size can be set.

Greenwald [5] presented an incomplete lock-free array-based set algorithm using DCAS that does not address the issue of handling duplicate copies of the same key in the array.

List-Based Lock-Free Set Algorithms

Massalin and Pu [14] presented a sketch of a lock-free linked list algorithm that uses the Motorola 68040 CAS2 instruction (more commonly known as DCAS in recent years). The algorithm performs deletions from the middle of the list in two steps, it first marks a node as deleted and then removes it from the list. The algorithm deals with memory management of deleted nodes by using reference counting. Many details of the algorithm are lacking.

Also using DCAS, Greenwald [5] presented a lock-free linked list algorithm. The algorithm allows insert and delete operations to complete in one step, in the absence of contention, by using a version number that must be incremented with every change to the list.

DCAS was supported on some generations of the Motorola 68000 processor family (as CAS2) in the 1980s. These implementations were extremely inefficient. Since then no processor architecture supports DCAS. Algorithms using DCAS remain impractical until such an instruction is supported efficiently on processor architectures.

Valois [25] presented CAS-based lock-free linked list algorithms that place one or more auxiliary nodes between every two normal nodes (i.e., nodes containing keys) to allow safe deletions. The algorithms also rely on a shared cursor that has to be positioned using CAS, at the target nodes of operations on the list, before these operations can be attempted. However, the concept of a shared cursor is actually inherently inconsistent with lock-freedom. For example, it is possible that two threads attempting operations (e.g., Insert) on the list at different locations, indefinitely alternate moving the shared cursor to their respective desired positions, with neither of them completing its intended operation. That is, the algorithm is susceptible to livelock, thus violating a basic correctness condition, and by definition is not lock-free. In addition, the algorithm is inefficient due to the extra work manipulating auxiliary nodes, and is not compatible with simple and efficient lock-free memory management methods.

Harris [6] presented the first correct CAS-based lock-free list-based set algorithm. A Delete operation, first marks a node as deleted using CAS to prevent new nodes from being linked to it, and then removes it from the list by swinging the next pointer of the previous node to the next node in the list, also using CAS.

The algorithm allows a thread traversing the list to access the contents of a node or a sequence of nodes after they have been removed from the list. If removed nodes are allowed to be reused immediately, the traversing thread may reach an incorrect result or corrupt the list.

This precludes the algorithm from using simple and efficient lock-free memory management methods, the IBM freelists [10, 23] and the safe memory reclamation method [17]. It forces the algorithm to use problematic memory management methods as discussed in the following subsection.

2.3 Memory Management

Initially, Harris used Valois' [25] reference counting memory management method. The method requires the inclusion of a reference counter in each dynamic node, that reflects the maximum number of references to that node in the object and the registers and local variables of threads operating on the object. A node can be reused only after its reference counter goes to zero. As reported by Harris [6], the method entailed prohibitive degradation in execution time by a factor of 10 to 15 times, relative to experiments without memory management, where removed nodes are not reused. Obviously, prohibiting memory reuse is not a generally practical solution for this problem, since the address space no matter how large is a finite resource.

As an alternative, Harris suggested assuming the use of a tracing garbage collector. However, garbage collectors pose many problems for lock-free algorithms. First, the presence of a garbage collector is not universal. Thus, lock-free object libraries that assume the use of garbage collectors are not portable to systems without such support. Second, even if present, garbage collectors are not lock-free as they either require mutual exclusion or stop-the-world techniques, or require special operating system support to access private stack space and registers. Third, the failure or delay of the garbage collector may prevent threads operating on lock-free objects from making progress indefinitely, thus violating lock-freedom. Fourth, Harris' algorithm prohibits threads from nullifying the pointers of dynamic nodes after their removal, thus the indefinite delay of a single thread is certain to

prevent the automatic garbage collector from freeing an unbounded number of nodes, indefinitely. The latter problem applies to Harris' algorithm when using lock-free reference counting methods [25, 4] as well.

As a third memory management option, Harris proposed a sketch of a deferred freeing memory management method. Each node includes an extra field through which it can be linked into a to-be-freed list when it is removed from the set object, without changing its critical contents. Each thread must set a per-thread shared timestamp before it starts an operation on the list. The method uses two to-be-freed lists that alternate taking the roles of the old list and the new list. The nodes in the old to-be-freed list are freed only when the removal time of the latest node in the list precedes the minimum per-thread timestamp. Also, at that time the two to-be-freed lists exchange their labels as old and new lists.

The method has multiple flaws. First, it is prone to deadlock if one of the threads does not perform operations on the set indefinitely (even if the thread itself is active). As the thread's timestamp remains unchanged indefinitely, the nodes in the old to-be-freed list are not freed, indefinitely, and the new to-be-freed list never takes the role of the old to-be-freed list. Second, even if threads are somehow guaranteed to operate on the set infinitely often, the method is actually blocking (i.e., not lock-free), as the delay or failure of a thread prevents it from updating its timestamp, and hence prevents the reuse of an unbounded number of nodes, indefinitely.

Detlefs *et. al.* [4] presented a lock-free reference counting memory management method that uses DCAS. Its performance is expected to be at best similar to that of Valois' reference counting method (i.e., extremely inefficient). Most statements in an algorithm involving pointers to dynamic nodes (even reads and register-to-register instruction) are transformed to functions involving CAS and DCAS operations. Its advantage over Valois' method is allowing arbitrary reuse of the memory of removed nodes.

Other known memory management methods that may accommodate Harris' list algorithm are not without serious problems. They are either blocking or depend on special operating system support [15, 5].

The simplest and most efficient lock-free memory management methods are not compatible with Harris' list algorithm. The IBM freelist [10, 23] is implemented as a lock-free stack. A thread allocates a node by popping it from the freelist in one successful CAS operation, and frees a node for future reuse by pushing it into the freelist, also in one successful CAS operation.

The other efficient lock-free memory management method is the safe memory reclamation method [17] that allows arbitrary reuse of the memory of deleted nodes and provides a solution to the ABA-problem² for pointers to dynamic nodes, without the use of extra space per pointer or per node. It guarantees an upper bound on the number of deleted nodes not yet freed, regardless of thread failures and delays. It is wait-free³, and operating system-independent.

The new algorithm is the first CAS-based lock-free list-based set algorithm that is compatible with all lock-free memory management methods, including the latter two.

²The ABA problem [10] is historically associated with CAS. It happens if a thread reads a value A from a shared location, computes a new value, and then attempts a CAS operation. The CAS may succeed when it should not and corrupt the object, if between the read and the CAS other threads change the value of the shared location from A to B and back to A again.

³An operation is wait-free if it is guaranteed to complete successfully in a finite number of its own steps regardless of other threads' actions [7].

```

// types and structures
structure NodeType {
    Key : KeyType;
    ⟨Mark,Next,Tag⟩ : ⟨boolean,*NodeType,TagType⟩;
}
structure MarkPtrType {
    ⟨Mark,Next,Tag⟩ : ⟨boolean,*NodeType,TagType⟩;
}
// T is the hash array
// M is the number of hash buckets
T[M] : MarkPtrType; // Initially ⟨0,null,dontcare⟩

```

Figure 2: Types and structures.

```

// Hash function
h(key:KeyType):0..M-1 { ... }

// Hash table operations
HashInsert(key:KeyType):boolean {
    // Assuming new node allocations always succeed
    node ← AllocateNode();
    node^.Key ← key;
    if Insert(&T[h(key)],node) return true;
    FreeNode(node); return false;
}

HashDelete(key:KeyType):boolean {
    return Delete(&T[h(key)],key);
}

HashSearch(key:KeyType):boolean {
    return Search(&T[h(key)],key);
}

```

Figure 3: Hash table operations.

3. THE ALGORITHM

Structures and Hash Table Functions

Since compatibility with simple and efficient memory management methods is a central advantage of the new algorithm, we start by presenting a version that is compatible with freelists [10, 23]. We discuss implementations of the new algorithm using other memory management methods later in this section.

The simplest and earliest known ABA-prevention mechanism is to include a tag with the target memory location such that both are manipulated atomically, and the tag is incremented with updates of the target location [10]. CAS (or a validation condition) succeeds only if the tag has not changed since the thread last read the location, assuming that the tag has enough bits to make full wraparound between the read and the CAS or validation condition practically impossible.

Figure 2 shows the data structures and the initial values of shared variables used by the algorithm. The main structure is an array T of size M . Each element in T is basically a pointer to a hash bucket, implemented as a singly linked list. For simplicity, we include a deletion mark with the header pointer, although it is guaranteed to be always clear.

Each dynamic node must contain the following fields: *Key*, *Mark*, *Next*, and *Tag*. The *Key* field holds a key value. The *Mark* field indicates if the key in the node has been deleted from the set. The *Next* field points to the following node in the linked list if any, or has a null value otherwise. The *Tag* field is used for preventing the ABA problem. $\langle \text{Mark}, \text{Next}, \text{Tag} \rangle$ must occupy a contiguous aligned memory block that can be manipulated atomically using CAS or LL/SC.

The *Mark* bit and the *Next* pointer can be placed in one word, since pointers are at least word aligned on all current major systems. The *Mark* bit can occupy a low order bit. The ABA-prevention *Tag* field can be placed in an adjacent word such that both words are aligned on a double word boundary.⁴. Later, in this section, we present an implementation that uses only single-word CAS or restricted LL/SC.

Figure 3 shows the hash table functions that use the new list-based set algorithm. Basically, every hash table operation, maps the input key to a hash bucket and then calls the corresponding list-based set function with the address of the bucket header as an argument.

The List-Based Set Algorithm

Figure 4 shows the Insert, Delete and Search operations of the new list-based set algorithm. The function Find (described later in detail) returns a Boolean value indicating whether a node with a matching key was found in the list. In either case, by its completion, it guarantees that the private variables *prev*, $\langle \text{cur}, \text{ptag} \rangle$ and $\langle \text{next}, \text{ctag} \rangle$ have captured a snapshot of a segment of the list including the node (if any) that contains the lowest key value greater than or equal to the input key and its predecessor pointer. Find guarantees that there was a time during its execution when $*\text{prev}$ was part of the list, $*\text{prev} = \langle 0, \text{cur}, \text{ptag} \rangle$, and if $\text{cur} \neq \text{null}$, then also at that time $\text{cur}^\wedge.\langle \text{Mark}, \text{Next}, \text{Tag} \rangle = \langle 0, \text{next}, \text{ctag} \rangle$ and $\text{cur}^\wedge.\text{Key}$ was the lowest key value that is greater than or equal to the input key. If $\text{cur} = \text{null}$ then it must be that at that time all the keys in the list were smaller than the input key. Note that, we assume a sequentially consistent memory model. Otherwise, memory barrier instructions need to be inserted in the code between memory accesses whose relative order of execution is critical.

An Insert operation returns false if the key is found to be already in the list. Otherwise, it attempts to insert the new node, containing the new key, before the node cur^\wedge , in one atomic step using CAS in line A3 after setting the *Next* pointer of the new node to cur , as shown in Figure 5. The success of the CAS in line A3 is the linearization point of an Insert of a new key in the set. The linearization point of an Insert that returns false (i.e., finds the key in the set) is discussed later when presenting Find.

The failure of the CAS in line A3 implies that one or more of three events must have taken place since the snapshot in Find was taken. Either the node containing $*\text{prev}$ was deleted (i.e. its *Mark* is set), the node cur^\wedge was deleted and removed (i.e., no longer reachable from *head*), or a new node was inserted immediately before cur^\wedge .

A Delete operation returns false if the key is not found in the list, otherwise, $\text{cur}^\wedge.\text{Key}$ must have been equal to the input key. If the key is found, the thread executing Delete attempts to mark cur^\wedge as deleted, using the CAS in line B2, as shown in Figure 6. If successful, the thread attempts to remove cur^\wedge by swinging $\text{prev}^\wedge.\text{Next}$ to next , while verifying that $\text{prev}^\wedge.\text{Mark}$ is clear, using the CAS in line B3.

The key technique of marking the next pointer of a deleted node in order to prevent a concurrent insert operation from linking another node after the deleted node was used earlier in Harris' lock-free list-based set algorithm [6], and was first used in Prakash, Lee, and Johnson's [20] lock-free FIFO queue algorithm.

DeleteNode prepares the removed node for reuse and its implementation is dependent on the memory management

⁴Most current architectures (32-bit as well as 64-bit) that support CAS (Intel x86, Sun SPARC) or restricted LL/SC (PowerPC, MIPS, Alpha) support their operation on aligned 64-bit blocks, and aligned 128-bit operations are likely to follow on 64-bit architectures

```

// private variables
prev : *MarkPtrType;
⟨pmark,cur,ptag⟩ : MarkPtrType;
⟨cmark,next,ctag⟩ : MarkPtrType;

Insert(head:*MarkPtrType,node:*NodeType):boolean {
    key ← node^.Key;
    while true {
        A1: if Find(head,key) return false;
        A2: node^.⟨Mark,Next⟩ ← ⟨0,cur⟩;
        A3: if CAS(prev,⟨0,cur,ptag⟩,⟨0,node,ptag+1⟩)
            return true;
    }
}

Delete(head:*MarkPtrType,key:KeyType):boolean {
    while true {
        B1: if !Find(head,key) return false;
        B2: if !CAS(&cur^.⟨Mark,Next,Tag⟩,
                    ⟨0,next,ctag⟩,
                    ⟨1,next,ctag+1⟩) continue;
        B3: if CAS(prev,⟨0,cur,ptag⟩,⟨0,next,ptag+1⟩)
            DeleteNode(cur); else Find(head,key);
            return true;
    }
}

Search(head:*MarkPtrType,key:KeyType):boolean {
    return Find(head,key);
}

Find(head:*MarkPtrType;key:KeyType) : boolean {
try_again:
    prev ← head;
    D1: ⟨pmark,cur,ptag⟩ ← *prev;
    while true {
        D2: if cur = null return false;
        D3: ⟨cmark,next,ctag⟩ ← cur^.⟨Mark,Next,Tag⟩;
        D4: ckey ← cur^.Key;
        D5: if *prev ≠ ⟨0,cur,ptag⟩ goto try_again;
        if !cmark {
            D6: if ckey ≥ key return ckey = key;
            D7: prev ← &cur^.⟨Mark,Next,Tag⟩;
        } else {
            D8: if CAS(prev,⟨0,cur,ptag⟩,⟨0,next,ptag+1⟩)
                {DeleteNode(cur); ctag ← ptag+1;}
            else
                goto try_again;
        }
        D9: ⟨pmark,cur,ptag⟩ ← ⟨cmark,next,ctag⟩;
    }
}

```

Figure 4: The list-based set algorithm.

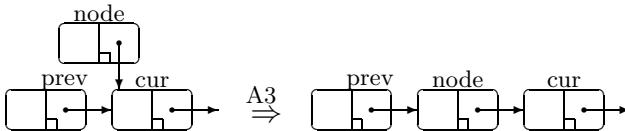


Figure 5: Insertion in the middle of the list.

method. For freelists, DeleteNode pushes the removed node onto the freelist.

The success of the CAS in line B2 is the linearization point of a Delete of a key that was already in the set. The linearization point of a Delete that does not find the input key in the set is discussed later when presenting the Find function.

The failure of the CAS in line B2 implies that one or more of three events must have taken place since the snapshot in Find was taken. Either the node cur^{\wedge} was deleted, a new

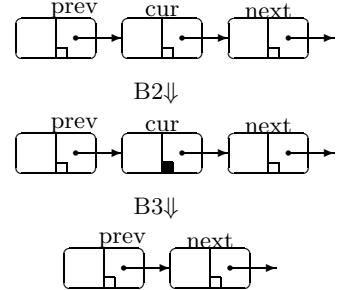


Figure 6: Deletion from the middle of the list.

node was inserted after cur^{\wedge} , or the node $next^{\wedge}$ was removed from the list. The failure of the CAS in line B3 implies that another thread must have removed the node cur^{\wedge} from the list after the success of the CAS in line B2 by the current thread. In such a case, a new Find is invoked in order to guarantee that the number of deleted nodes not yet removed never exceeds the maximum number of concurrent threads operating on the object.

The Search operation simply relays the response of the Find function.

The Find function starts by reading the header of the list $*head$ in line D1. If the Next pointer of the header is null, then the list must be empty, therefore Find returns false after setting $prev$ to $head$ and cur to $null$. The linearization point of finding the list empty is the reading of $*head$ in line D1. That is, it is the linearization point of all Delete and Search operations that return false after finding the set empty.

If the list is not empty, a thread executing Find traverses the nodes of the list using the private pointers $prev$, cur , and $next$. Whenever it detects a change in $*prev$, in lines D5 or D8, it starts over from the beginning. As discussed in Section 4, the algorithm is lock-free. A change in $*prev$ implies that some other threads have made progress in the meantime.

A thread keeps traversing the list until it either finds a node with a key greater than or equal to the input key, or reaches the end of the list without finding such node. If it is the former case, it returns the result of the condition $cur^.Key = key$ at the time of its last execution of the read in line D3, with $prev$ pointing to $cur^.⟨Mark,Next,Tag⟩$ and $cur^.Key$ is the lowest key in the set that is greater than or equal the input key, at that point (line D3). If the thread reaches the end of the list without finding a greater or equal key, it returns false, with $*prev$ pointing to the fields of the last node and $cur = null$.

In all cases of non-empty lists, the linearization point of the snapshot in Find is the last reading of $cur^.⟨Mark,Next,Tag⟩$ (line D3) by the current thread. That is, it is the linearization point of all Insert operations that return false and all Search operations that return true, as well as all Delete and Search operations that return false after finding the set non-empty.

During the traversal of the list, whenever the thread encounters a marked node, it attempts to remove it from the list, using CAS in line D8. If successful, the removed node is prepared for future reuse in DeleteNode and $cmark$ is updated for continued traversal.

Note that, for a snapshot in Find to be valid, $prev^.Mark$ and $cur^.mark$ must be found to be clear. If a Mark is found to be set the associated node must be removed first before capturing a valid snapshot.

On architectures that support restricted LL/SC but not CAS, implementing $CAS(addr,exp,new)$ using the following routine suffices for the purposes of the new algorithm.

```

while true {if LL(addr) ≠ exp return false;
           if SC(addr,new) return true;}

```

Moir [19] presented a general implementation of CAS using restricted LL/SC that allows $\exp = \text{new}$, infinitely often.

Using Other Memory Management Methods

As mentioned earlier, the algorithm is compatible with all memory management methods for lock-free objects. For example, if lock-free reference counting [4, 25] or automatic garbage collection is used, when a node is removed, the call to DeleteNode only needs to nullify the removed node’s fields, just in case their values match the address of some dynamic structure and thus may form a problematic garbage cycle. Since, as discussed in Section 2, all memory management methods other than freelists and the safe memory reclamation method (SMR) [17] are either extremely inefficient, blocking, dependent on special system support, and/or dependent on DCAS, we focus on using SMR with the new algorithm.

SMR’s advantages over freelists include allowing the memory of removed dynamic nodes to be reused arbitrarily. That is, it allows the memory use of a dynamic data structure to shrink. Another advantage is that it can prevent the ABA problem without the need for double-width CAS or LL/SC or any extra space per pointer or per node. Thus, it allows lock-free objects to use minimal space, which is an important issue for hash tables with large numbers of buckets.

SMR requires target lock-free algorithms to associate a number of shared pointers, called hazard pointers, (three in the case of this algorithm) with each participating thread. The method guarantees that no deleted⁵ dynamic node is freed or reused as long as some thread’s hazard pointers have been pointing to it continuously from a time when it was not deleted. It is impossible for Valois’ and Harris’ list-based set algorithms to comply with this requirement as they allow a thread to traverse a node or a sequence of nodes after these nodes have already been removed from the list, and hence possibly deleted.

Figure 7 shows a version of the new algorithm that is compatible with SMR. Before returning, Insert, Delete, and Search nullify the hazard pointers to guarantee that the amortized time of processing a deleted node until it is freed for reuse is (logarithmically) bounded by contention. That is, whenever a thread is not operating on the object, its hazard pointers are null. This is done after the end of hazards (i.e., accesses to dynamic structure when they are possibly deleted and the use of pointers to dynamic structures as expected values of ABA-prone CAS operations in lines A3, B2 and B3).

In the Find function, there are accesses to dynamic structures in lines D3, D4, and D8, and the addresses of dynamic nodes are used as expected values of ABA-prone validation conditions and CAS operations in lines D5 and D8.

Lines E1 and E2 serve to guarantee that the next time a thread accesses cur^\wedge in lines D3 and D4 and executes the validation condition in line D5, it must be the case that the hazard pointer $*\text{hp1}$ has been continuously pointing to cur^\wedge from a time when it was in the list, thus guaranteeing that cur^\wedge is not free during the execution of lines D3 and D4.

The ABA problem is impossible in the validation condition in line D5 and the CAS in line D8, even if the value of $*\text{prev}$ has changed since last read in line D1 (or line D3 for subsequent loop executions). The removal and reinsertion of cur^\wedge after D1 and before E2 do not cause the ABA problem in D5 or D8. The hazardous sequence of events that can cause the

⁵In this context, the term deleted refers to calls to DeleteNode and is not related to the HashDelete or list Delete operations. A deleted node is one that was passed as an argument of DeleteNode.

ABA problem in D5 and D8 is if cur^\wedge is removed and then reinserted in the list after line D3 and before D5 or D8. The insertion and removal of other nodes between $*\text{prev}$ and cur^\wedge never causes the ABA problem in D5 and D8. Thus, by preventing cur^\wedge from being removed and reinserted during the current thread’s execution of D3–D5 or D3–D8, SMR makes the ABA problem impossible in lines D5 and D8.

Lines E3, E4, E5, and E6 serve to prevent cur^\wedge in the next iteration of the loop (if any) from being removed and reinserted during the current thread’s execution of D3–D5 or D3–D8, and also to guarantee that if the current thread accesses cur^\wedge in the next iteration in lines D3 and D4, then cur^\wedge is not free.

Lines E3 and E4 must be between lines D3 and D5. Lines E5 and E6 can either immediately precede or immediately follow lines D7 and D9, respectively.

The protection of cur^\wedge in one iteration continues in the next iteration for protecting the node containing $*\text{prev}$, such that it is guaranteed that when the current thread accesses $*\text{prev}$ in lines D5 and D8, that node is not free. The same protections of $*\text{prev}$, cur^\wedge and next^\wedge continue through the execution of lines A3, B2, and B3.

The three hazard pointers $*\text{hp0}$, $*\text{hp1}$, and $*\text{hp2}$ shadow the movement of the three private pointers next , cur , and prev down the list, respectively ($*\text{hp2}$ holds the address of the node including $*\text{prev}$ not the value prev). The movement of the pointers resembles that of a worm with three segments, with next , cur , and prev as the head, midsection, and tail, respectively. In order to advance through the list, a thread assigns a variant of the value of cur ($\&\text{cur}^\wedge.\langle\text{Mark},\text{Next}\rangle$) to prev , then assigns next to cur and finally advances next . SMR requires that if any hazard pointer inherits its value from another, then the index of the latter must be less than that of the former in the shared hazard pointer array. This is needed because the SMR algorithm scans the hazard pointer array in ascending index order, non-atomically, i.e., one hazard pointer at a time [17]. Therefore, the indices of $*\text{hp0}$, $*\text{hp1}$, and $*\text{hp2}$ must be in ascending order, respectively.

Finally, it is worth noting that in the new algorithm, the Key field of a dynamic node does not need to be preserved after the node’s removal. Therefore, that field can be reused by the SMR algorithm to link deleted nodes, thus offering significant space savings.

Also, the bucket headers only need one word per bucket. The $\langle\text{Mark},\text{Next}\rangle$ field in dynamic nodes only needs to occupy one word, and no ABA tags are needed. Also, as an additional advantage, SMR allows the new algorithm to be completely dynamic using only single word CAS or restricted LL/SC.

4. CORRECTNESS

For brevity, we provide only informal proof sketches, with lemmas indicating the proof roadmap. First we introduce some definitions.

For all times t , a node is in the list at t , iff at t it is reachable by following the Next pointers of reachable nodes starting from $\text{head}^\wedge.\text{Next}$.

For all times t , the list is in state $S_{n,m}$ iff the following are all true:

1. $|\{\text{node } x: \text{at } t, x \text{ is in the list} \wedge x \text{ is not marked.}\}| = n$.
2. $|\{\text{node } x: \text{at } t, x \text{ is in the list} \wedge x \text{ is marked.}\}| = m$.
3. $\forall \text{nodes } x, y \text{ in the list}, x.\text{Next} = y \implies x.\text{Key} < y.\text{Key}$.

For example, a list is in state $S_{0,0}$ if it contains no nodes (i.e., its head pointer is null). A list is in state $S_{2,1}$ if it contains exactly three nodes, one is marked as deleted and the other two are not.

```

// types and structures
structure NodeType {
    Key : KeyType;
    ⟨Mark,Next⟩ : ⟨boolean,*NodeType⟩;
}
structure MarkPtrType {
    ⟨Mark,Next⟩ : ⟨boolean,*NodeType⟩;
}

// Shared variables
T[M] : MarkPtrType; // Initially ⟨0,null⟩
// private variables
prev : *MarkPtrType;
cur,next : *NodeType;

// No change in HashInsert, HashDelete, HashSearch.

Find(head:*MarkPtrType;key:KeyType) : boolean {
try_again:
    prev ← head;
D1: ⟨pmark,cur⟩ ← *prev;
E1: *hp1 ← cur;
E2: if *prev ≠ ⟨0,cur⟩ goto try_again;
    while true {
D2: if cur = null return false;
D3: ⟨cmark,next⟩ ← cur^.⟨Mark,Next⟩;
E3: *hp0 ← next;
E4: if cur^.⟨Mark,Next⟩ ≠ ⟨cmark,next⟩ goto try_again;
D4: ckey ← cur^.Key;
D5: if *prev ≠ ⟨0,cur⟩ goto try_again;
    if !cmark {
D6: if ckey ≥ key return ckey = key;
D7: prev ← &cur^.⟨Mark,Next⟩;
E5: *hp2 ← cur;
    } else {
D8: if CAS(prev,⟨0,cur⟩,⟨0,next⟩)
        DeleteNode(cur); else goto try_again;
    }
D9: cur ← next;
E6: *hp1 ← next;
    }
}
}
}

// SMR related variables
// static private variables
hp0,hp1,hp2 : **NodeType;
// HP is the shared array of hazard pointers
// j is thread id for SMR purposes
hp0 = &HP[3*j]
hp1 = &HP[3*j+1]
hp2 = &HP[3*j+2]
// The order is important

Insert(head:*MarkPtrType,node:*NodeType):boolean {
    key ← node^.Key;
    while true {
A1: if Find(head,key) {result ← false; break;}
A2: node^.⟨Mark,Next⟩ ← ⟨0,cur⟩;
A3: if CAS(prev,⟨0,cur⟩,⟨0,node⟩)
    {result ← true; break;}
    }
*hp0 ← null; *hp1 ← null; *hp2 ← null;
return result;
}

Delete(head:*MarkPtrType,key:KeyType):boolean {
    while true {
B1: if !Find(head,key) {result ← false; break;}
B2: if !CAS(&cur^.⟨Mark,Next⟩,
            ⟨0,next⟩,
            ⟨1,next⟩) continue;
B3: if CAS(prev,⟨0,cur⟩,⟨0,next⟩)
    DeleteNode(cur); else Find(head,key);
    result ← true; break;
    }
*hp0 ← null; *hp1 ← null; *hp2 ← null;
return result;
}

Search(head:*MarkPtrType,key:KeyType):boolean {
    result ← Find(head,key);
    *hp0 ← null; *hp1 ← null; *hp2 ← null;
    return result;
}
}

```

Figure 7: Version of the new algorithm using the SMR method [17] (SMR-related code is in a different font).

A list in state $S_{n,m}$ corresponds to an abstract set K with n elements, such that for all nodes x in the list that are not marked, $x.Key \in K$.

LEMMA 1. \forall nodes x, y in the list, $x \neq y \implies x.Key \neq y.Key$.

LEMMA 2. The Key field of a node never changes while the node is in the list.

LEMMA 3. Once set, the Mark field of a node remains set until the node's removal from the list.

LEMMA 4. A node is never removed from the list while its Mark field is clear.

Safety

To prove safety, we use the state definitions and the state transition diagram of Figure 8. The list is in a valid state, iff it matches the definition of some state $S_{n,m}$. The state of the list changes only on the success of the CAS operations in lines A3, B2, B3, and D8. Our goal is to prove that the following claim is always true.

CLAIM 1. The list is in a valid state, and if a CAS succeeds then a correct transition occurs as shown in the state transition diagram in Figure 8, and the transition is consistent with the abstract set semantics.

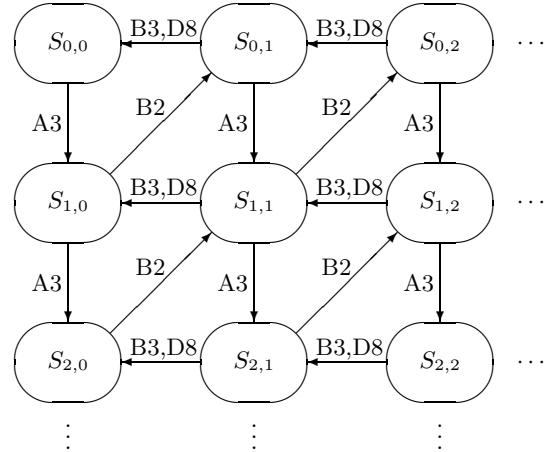


Figure 8: State transition diagram.

Initially, Claim 1 is true, assuming that the list starts in a valid state, e.g., $S_{0,0}$. For a proof by induction, we need to show that whenever a CAS operation succeeds at time t , and

Claim 1 (the induction hypothesis) has been true up to that time, then only a correct transition can take place and the transition is consistent with the abstract set semantics.

All the following theorems and lemmas are predicated on the assumption that Claim 1 has been true for all times before the success of the CAS operation or validation condition in question at time t .

LEMMA 5. *At time t , the validation condition in line D5 succeeds \wedge $\text{prev} \neq \text{head} \implies$ at t , $\exists \text{node } x \text{ in the list } :: \text{prev} = \&x.\langle \text{Mark}, \text{Next} \rangle \wedge \text{key} > x.\text{Key}$.*

Informally, on the success of the validation condition in D5, prev is in the list.

LEMMA 6. *At time t , the validation condition in line D5 succeeds \wedge $\text{cur} \neq \text{null} \implies$ at t , node cur^\wedge is in the list.*

LEMMA 7. *The CAS in line A3 succeeds \implies for all times since the current thread last executed the validation condition in line D5, $\text{prev}.\text{Mark}$ is clear.*

LEMMA 8. *The CAS in line A3 succeeds \implies for all times since the current thread last executed the validation condition in line D5, cur^\wedge is in the list \wedge $\text{cur}^\wedge.\text{Key} > \text{key}$.*

LEMMA 9. *The CAS in line B2 succeeds \implies for all times since the current thread last executed the validation condition in line D5, cur^\wedge is in the list $\wedge \text{cur}^\wedge.\text{Key} = \text{key}$.*

LEMMA 10. *The CAS in line B3 succeeds \implies for all times since the current thread last executed the validation condition in line D5, $\text{prev}.\text{Mark}$ is clear.*

LEMMA 11. *The CAS in line B3 succeeds \implies for all times since the current thread last executed the CAS in line B2 successfully, $\text{cur}.\text{Next} = \text{next}$.*

LEMMA 12. *The CAS in line D8 succeeds \implies for all times since the current thread last executed the validation condition in line D5, $\text{prev}.\text{Mark}$ is clear.*

LEMMA 13. *The CAS in line D8 succeeds \implies for all times since the current thread last read $\text{cur}^\wedge.\langle \text{Mark}, \text{Next} \rangle$ in line D3, node cur^\wedge is in the list $\wedge \text{cur}^\wedge.\langle \text{Mark}, \text{Next} \rangle = \langle 1, \text{next} \rangle$.*

THEOREM 1. *If successful, the CAS in line A3 takes the list to a valid state and inserts the new key into the set.*

THEOREM 2. *If successful, the CAS in line B2 takes the list to a valid state and removes $\text{cur}^\wedge.\text{Key}$ from the set.*

THEOREM 3. *If successful, the CAS in line B3 takes the list to a valid state and does not modify the set.*

THEOREM 4. *If successful, the CAS in line D8 takes the list to a valid state and does not modify the set.*

THEOREM 5. *Claim 1 is true at all times.*

Lock-Freedom

LEMMA 14. *Whenever one of the CAS operations or validation conditions in lines A3, B2, D5, and D8 (also E2 and E4 when SMR is used) fails, then the state of the list must have changed since the current thread last executed line D1.*

LEMMA 15. *If the list is in state $S_{n,m}$ and then the state of the list changes $m+1$ times, then at least one operation (Insert, Delete, or Search) on the list must have succeeded during that period.*

LEMMA 16. *If a thread starts an operation (Insert, Delete, or Search) on the list when it is in state $S_{n,m}$ and then executes line D1 $m+2$ times, then some operation on the list must have completed successfully since the start of the current operation.*

LEMMA 17. *If a thread starts an operation on the list when it is in state $S_{n,m}$ and then executes line D2 $n.m + 2$ times, then some operation on the list must have completed successfully since the start of the current operation.*

THEOREM 6. *The new algorithm is lock-free.*

Linearizability

An implementation of an object is linearizable if it can always give an external observer, observing only the abstract object operations, the illusion that each of these operations takes effect instantaneously at some point between its invocation and its response [9].

The new algorithm is linearizable, since every operation on the list has a specific linearization point, where it takes effect. By the safety properties and the definition of $S_{n,m}$, it can be shown that the responses of the list operations are consistent with the state of the abstract set object at these points. The following are the linearization points:

- Every Search and Delete operation that returns false, after searching an empty list, takes effect on its last reading of *head in line D1.
- Every Search and Delete operation that returns false, after searching a non-empty list, takes effect on its last reading of $\text{cur}^\wedge.\langle \text{Mark}, \text{Next} \rangle$ in line D3.
- Every Insert operation that returns false takes effect on its last reading of $\text{cur}^\wedge.\langle \text{Mark}, \text{Next} \rangle$ in line D3.
- Every Search operation that returns true takes effect on its last reading of $\text{cur}^\wedge.\langle \text{Mark}, \text{Next} \rangle$ in line D3.
- Every Insert operation that returns true takes effect on its only successful execution of the CAS in line A3.
- Every Delete operation that returns true takes effect on its only successful execution of the CAS in line B2.

THEOREM 7. *The new algorithm is linearizable.*

5. PERFORMANCE

We used a 4-processor (375 MHz 604e) IBM PowerPC multiprocessor to evaluate the performance of the new lock-free hash table algorithm, relative to hash tables that use Harris' lock-free list algorithm and various lock-based implementations.

For the new algorithm, we used two implementations, one with memory management and one without. For memory management, we used the safe memory reclamation (SMR) method [17]. Freelist with ABA tags [10] can also be used efficiently with the new algorithm, but we used SMR as it requires only single-word atomic operations. The implementation without memory management is impractical. We include this implementation only as a means to determine memory management cost for the new algorithm. For this implementation, we preallocated all free nodes before the beginning of each experiment and nodes removed from the hash table were not reused.

For Harris' algorithm, we also used two implementations, with and without memory management. However, for memory management, Harris' algorithm cannot use SMR or freelists. The only lock-free OS-independent memory management methods that can be used with Harris' algorithm are the reference counting methods of Valois'[25] and Detlefs *et. al.* [4]. The latter requires DCAS which renders it impractical, while the former uses only CAS. We used Valois' method with corrections we employed in earlier work [18], and applied it with extreme care to eliminate unnecessary manipulations of reference counters that would otherwise degrade performance even further if automatic transformations were employed.

We used four lock-based implementations. Two implementations with global locks for the whole hash table object. The other two implementations associate a lock with every hash bucket, in order to increase concurrency. In both cases, one implementation used mutual exclusion locks and the other used reader-writer locks for allowing concurrent read-only accesses (i.e., Search operations). For mutual exclusion locks, we used the Test-and-Test-and-Set lock [21]. For reader-writer locks, we used a variant of a simple centralized algorithm by Mellor-Crummey and Scott [16].

All lock and CAS operations were implemented in short assembly language routines using LL/SC with minimal function

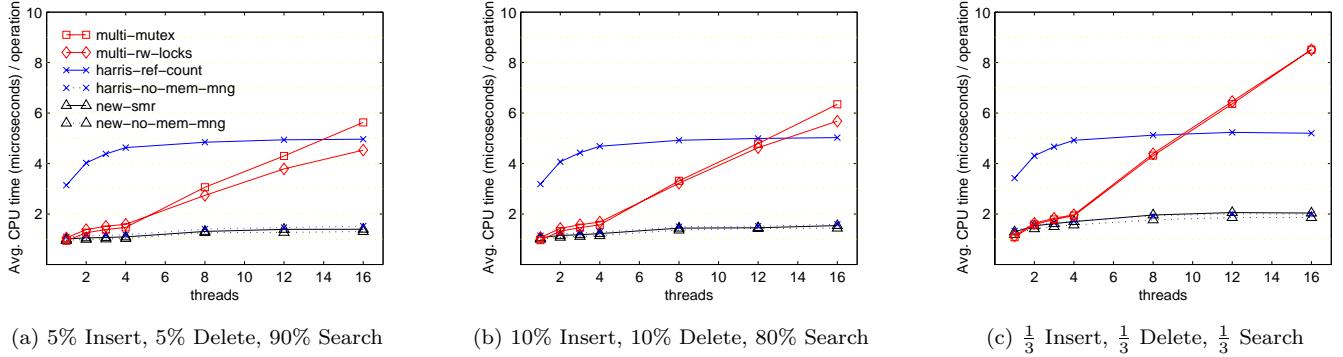


Figure 9: Average execution time of operations on a hash table with 100 buckets and average load factor 1.

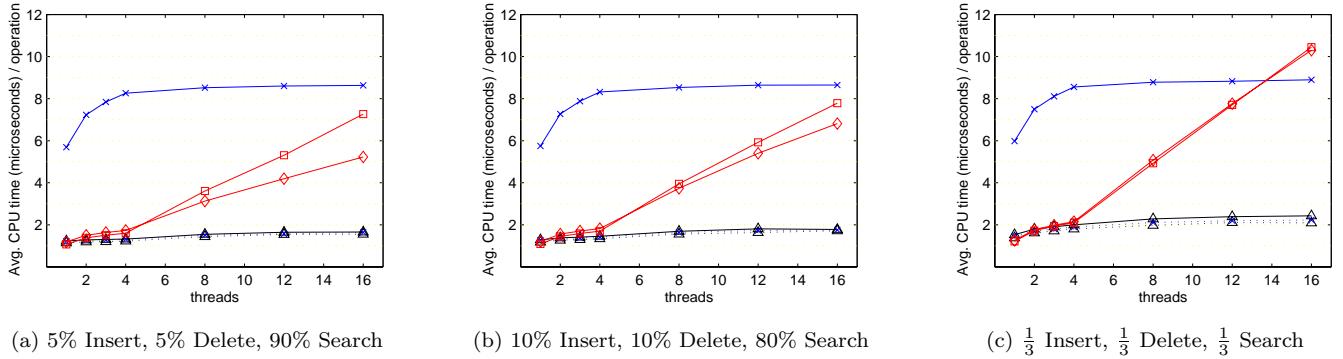


Figure 10: Average execution time of operations on a hash table with 100 buckets and average load factor 5.

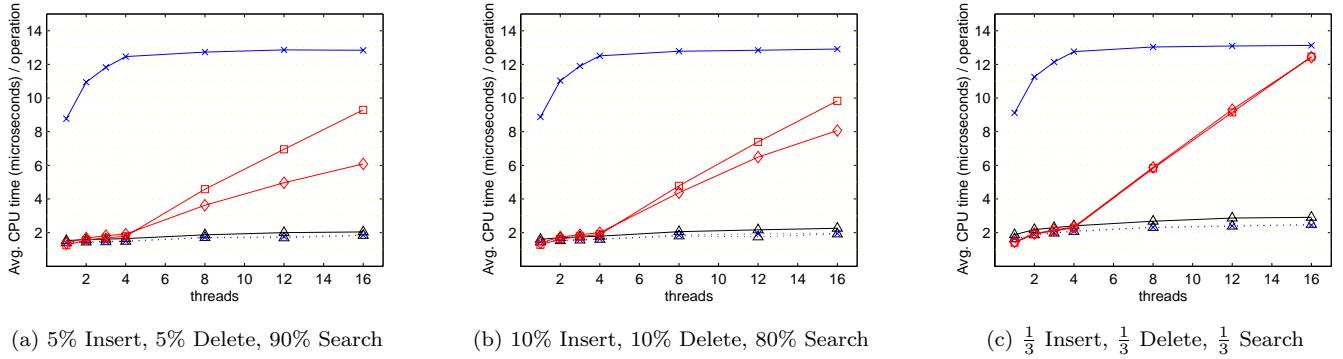


Figure 11: Average execution time of operations on a hash table with 100 buckets and average load factor 10.

call overheads. For all algorithms, when appropriate cache line padding was used to eliminate false sharing. The algorithms were compiled using the highest optimization level. In all experiments, all memory needs fit in physical memory.

We generated workloads of operations (Insert, Delete and Search) by choosing random keys uniformly from a range of integers. We varied several parameters: the number of hash buckets m , the mix of operations, and the range of keys U . We controlled the average load factor α (average number of keys per hash bucket) indirectly by initializing the hash table to include αm keys and by selecting the size of U to be equal to $2\alpha m$.

In all experiments we varied the number of threads operating concurrently on the hash table. In all cases, each thread performed 1,000,000 operations. We measured the total CPU time used by all threads to execute these operations. The pseudo-random sequences for different threads were non-

overlapping in each experiment, but were repeatable for each thread for fairness in comparing different algorithms.

We conducted many experiments with various parameters. All showed the same general trends. For brevity and clarity, we include only results of representative experiments using a hash table with 100 buckets.

Figures 9, 10 and 11 show the average execution time per operation for a hash table with various average load factors and operation mixes. By initializing the hash table to include 100, 500, and 1000 keys, and setting $|U|$ to 200, 1000 and 2000, the hash table had average load factors α of 1, 5 and 10, respectively.

For clarity, we omit the results for the single lock implementation, since as expected they exhibit significantly inferior performance in comparison to other implementations, including those using fine-grain locks.

As expected, with higher percentages of Search operations,

reader-writer locks outperform mutual exclusion locks. The significant effect of multiprogramming (more than 4 threads) on the performance of all lock-based implementations is clear, with varying degrees depending on the frequency of update operations and the level of multiprogramming.

Being lock-free, Harris' algorithm is immune to the performance degradation that affected the lock-based implementations under multiprogramming. However, its performance with memory management is substantially inferior to other practical alternatives, due to the high overhead of reference counting. Combined with the fact that its memory requirements are unbounded even with bounded object size using any memory management method, it is clear that Harris' algorithm is impractical with respect to both performance and robustness.

The new algorithm provides the best overall performance, with and without multiprogramming, with various operation mixes, and under high and low contention. It outperforms Harris' algorithm by a factor of 2.5 or more when using lock-free memory management methods, and matches or exceeds its performance under the unrealistic assumption of no memory management, which is not the same as assuming automatic garbage collection (which is likely to cost much more than SMR). The cost of memory management for the new algorithm is consistently low.

The results indicate that for general practical use, the new algorithm is the algorithm of choice for implementing shared hash tables.

6. CONCLUSIONS

Prior lock-free algorithms for sets and hash tables suffer from serious drawbacks that prevent or limit their use in practice. These drawbacks include size inflexibility, dependence on DCAS, and dependence on problematic and highly-inefficient memory management techniques.

In this paper we presented the first CAS-based lock-free list-based set algorithm that is compatible with all memory management methods. We used it as a building block of a dynamic lock-free hash table algorithm.

Our experimental results showed significant performance advantages of the new algorithm over the best known lock-free as well as lock-based hash table implementations. The new algorithm outperforms the best prior lock-free algorithm, Harris' [6], by a factor of 2.5 or more, in all lock-free cases. It generally outperforms the best lock-based implementations, with and without multiprogramming, under high and low contention, often by significant margins. The results indicate that it is the algorithm of choice for implementing shared hash tables. Also, the new algorithm offers upper bounds on its memory use relative to the set size with all lock-free as well as blocking memory management method, while Harris' algorithm cannot provide such bound even with bounded maximum set size with any memory management method.

This paper also demonstrates the significant effect of memory management characteristics of dynamic lock-free algorithms on their performance, robustness and practicality. Evaluating the merits of prior and future lock-free algorithms must take into account their compatibility with memory management methods. A dynamic lock-free algorithm cannot be considered generally practical unless it is compatible with freelists.

7. REFERENCES

- [1] James H. Anderson and Mark Moir. Universal Constructions for Large Objects. *IEEE Transactions on Parallel and Distributed Systems* 10(12): 1317–1332, December 1999.
- [2] Greg Barnes. A Method for Implementing Lock-Free Shared-Data Structures. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 261–270, June–July 1993.
- [3] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.
- [4] David L. Detlefs, Paul A. Martin, Mark Moir, and Guy L. Steele Jr. Lock-Free Reference Counting. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing*, pages 190–199, August 2001.
- [5] Michael B. Greenwald. *Non-Blocking Synchronization and System Design*. Ph.D. Thesis, Stanford University Technical Report STAN-CS-TR-99-1624, August 1999.
- [6] Timothy L. Harris. A Pragmatic Implementation of Non-Blocking Linked Lists. In *Proceedings of the 15th International Symposium on Distributed Computing*, pages 300–314, October 2001.
- [7] Maurice P. Herlihy. Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems* 13(1): 124–149, January 1991.
- [8] Maurice P. Herlihy. A Methodology for Implementing Highly Concurrent Objects. *ACM Transactions on Programming Languages and Systems* 15(5): 745–770, November 1993.
- [9] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems* 12(3): 463–492, July 1990.
- [10] IBM System/370 Extended Architecture, Principles of Operation. IBM Publication No. SA22-7085, 1983.
- [11] Amos Israeli and Lihu Rappoport. Disjoint-Access-Parallel Implementations of Strong Shared Memory Primitives. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, pages 151–160, August 1994.
- [12] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley 1973.
- [13] Vladimir Lanin and Dennis Shasha. Concurrent Set Manipulation without Locking. In *Proceedings of the 7th ACM Symposium on Principles of Database Systems*, pages 211–220, March 1988.
- [14] Henry Massalin and Carlton Pu. A Lock-Free Multiprocessor OS Kernel. Technical Report No. CUCS-005-91, Columbia University, 1991.
- [15] Paul E. McKenney and John D. Slingwine. Read-Copy Update: Using Execution History to Solve Concurrency Problems. In *Proceedings of the 11th International Conference on Parallel and Distributed Computing Systems*, October 1998.
- [16] John M. Mellor-Crummey and Michael L. Scott. Scalable Reader-Writer Synchronization for Shared-Memory Multiprocessors. In *Proceedings of the 3rd ACM Symposium on Principles and Practice of Parallel Programming*, pages 106–113, April 1991.
- [17] Maged M. Michael. Safe Memory Reclamation for Dynamic Lock-Free Objects Using Atomic Reads and Writes. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing*, July 2002.
- [18] Maged M. Michael and Michael L. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, pages 267–275, May 1996.
- [19] Mark Moir. Practical Implementations of Non-Blocking Synchronization Primitives. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, pages 219–228, August 1997.
- [20] Sundeep Prakash, Yann-Hang Lee, and Theodore Johnson. A Nonblocking Algorithm for Shared Queues Using Compare-and-Swap. *IEEE Transactions on Computers* 43(5): 548–559, May 1994.
- [21] Larry Rudolph and Zary Segall. Dynamic Decentralized Cache Schemes for MIMD Parallel Processors. In *Proceedings of the 11th International Symposium on Computer Architecture*, pages 340–347, June 1984.
- [22] Nir Shavit and Dan Touitou. Software Transactional Memory. *Distributed Computing* 10(2): 99–116, 1997.
- [23] R. Kent Treiber. Systems Programming: Coping with Parallelism. Research Report RJ 5118, IBM Almaden Research Center, April 1986.
- [24] John Turek, Dennis Shasha, and Sundeep Prakash. Locking without Blocking: Making Lock Based Concurrent Data Structure Algorithms Nonblocking. In *Proceedings of the 11th ACM Symposium on Principles of Database Systems*, pages 212–222, June 1992.
- [25] John D. Valois. Lock-Free Linked Lists Using Compare-and-Swap. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 214–222, August 1995.

Split-Ordered Lists: Lock-Free Extensible Hash Tables

ORI SHALEV

Tel-Aviv University, Tel-Aviv, Israel

AND

NIR SHAVIT

Tel-Aviv University and Sun Microsystems Laboratories, Tel-Aviv, Israel

Abstract. We present the first lock-free implementation of an extensible hash table running on current architectures. Our algorithm provides concurrent insert, delete, and find operations with an expected $O(1)$ cost. It consists of very simple code, easily implementable using only load, store, and compare-and-swap operations. The new mathematical structure at the core of our algorithm is *recursive split-ordering*, a way of ordering elements in a linked list so that they can be repeatedly “split” using a single compare-and-swap operation. Metaphorically speaking, our algorithm differs from prior known algorithms in that extensibility is derived by “moving the buckets among the items” rather than “the items among the buckets.” Though lock-free algorithms are expected to work best in multiprogrammed environments, empirical tests we conducted on a large shared memory multiprocessor show that even in non-multiprogrammed environments, the new algorithm performs as well as the most efficient known lock-based resizable hash-table algorithm, and in high load cases it significantly outperforms it.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; D.4.1 [**Operating Systems**]: Process Management—*Synchronization; concurrency; multiprocessing/multiprogramming/multitasking*; E.2 [**Data Storage Representation**]—*Hash-table representations*

General Terms: Algorithms, Theory, Performance, Experimentation

Additional Key Words and Phrases: Concurrent data structures, hash table, non-blocking synchronization, compare-and-swap

This work was performed while N. Shavit was at Tel-Aviv University, supported by a Collaborative Research Grant from Sun Microsystems.

A preliminary version of this article appeared in *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing* (Boston, MA), ACM, New York, 2003, pp. 102–111.

Copyright is held by Sun Microsystems, Inc.

Authors’ address: School of Computer Science, Tel-Aviv University, Tel-Aviv, Israel 69978, e-mail: orish@post.tau.ac.il; shanir@sun.com.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 1515 Broadway, New York, NY 10036 USA, fax: +1 (212) 869-0481, or permissions@acm.org.

© 2006 ACM 0004-5411/06/0500-0379 \$5.00

1. Introduction

Hash tables, and specifically extensible hash tables, serve as a key building block of many high performance systems. A typical extensible hash table is a continuously resized array of buckets, each holding an expected constant number of elements, and thus requiring an expected constant time for insert, delete and find operations [Cormen et al. 2001]. The cost of resizing, the redistribution of items between old and new buckets, is amortized over all table operations, thus keeping the average complexity of any one operation constant. As this is an extensible hash table, “resizing” means extending the table. It is interesting to note, as argued elsewhere [Hsu and Yang 1986; Lea (e-mail communication 2005)], that many of the standard concurrent applications using hash tables require tables to only increase in size.”

We are concerned in implementing the hash table data structure on multiprocessor machines, where efficient synchronization of concurrent access to data structures is essential. Lock-free algorithms have been proposed in the past as an appealing alternative to lock-based schemes, as they utilize strong primitives such as CAS (*compare-and-swap*) to achieve fine grained synchronization. However, lock-free algorithms typically require greater design efforts, being conceptually more complex.

This article presents the first lock-free extensible hash table that works on current architectures, that is, uses only loads, stores and CAS (or LL/SC [Moir 1997]) operations. In a manner similar to sequential linear hashing [Litwin 1980] and fitting real-time¹ applications, resizing costs are split incrementally to achieve expected $O(1)$ operations per insert, delete and find. The proposed algorithm is simple to implement, leading us to hope it will be of interest to practitioners as well as researchers. As we explain shortly, it is based on a novel *recursively split-ordered* list structure. Our empirical testing shows that in a concurrent environment, even without multiprogramming, our lock-free algorithm performs as well as the most efficient known lock-based extensible hash-table algorithm due to Lea [2003], and in high-load cases, it significantly outperforms it.

1.1. BACKGROUND. There are several lock-based concurrent hash table implementations in the literature. In the early eighties, Ellis [1983, 1987] proposed an extensible concurrent hash table for distributed data based on a two level locking scheme, first locking a table directory and then the individual buckets. Michael [2002a] has recently shown that on shared memory multiprocessors, simple algorithms using a reader-writer lock [Mellor-Crummey and Scott 1991] per bucket have reasonable performance for non-extensible tables. However, to resize one would have to hold the locks on all buckets simultaneously, leading to significant overheads. A recent algorithm by Lea [2003], proposed for *java.util.concurrent*, the Java™ Concurrency Package, is probably the most efficient known concurrent extensible hash algorithm. It is based on a more sophisticated locking scheme that involves a small number of high level locks rather than a lock per bucket, and allows concurrent searches while resizing the table, but not concurrent inserts or deletes. In general, lock-based hash-table algorithms are expected to suffer from the typical drawbacks of blocking synchronization: deadlocks, long delays, and

¹In this article, by *real-time* we mean *soft real-time* [Buttazzo et al. 2005], where some flexibility on the real-time requirements is allowed.

priority inversions [Greenwald 1999]. These drawbacks become more acute when performing a *resize* operation, an elaborate “global” process of redistributing the elements in all the hash table’s buckets among newly added buckets. Designing a lock-free extensible hash table is thus a matter of both practical and theoretical interest.

Michael [2002a], builds on the work of Harris [2001] to provide an effective compare-and-swap (CAS) based lock-free linked-list algorithm (which we will elaborate upon in the following section). He then uses this algorithm to design a lock-free hash structure: a fixed size array of hash buckets with lock-free insertion and deletion into each. He presents empirical evidence that shows a significant advantage of this hash structure over lock-based implementations in multiprogrammed environments. However, this structure is not extensible: if the number of elements grows beyond the predetermined size, the time complexity of operations will no longer be constant.

As part of his “two-handed emulation” approach, Greenwald [2002] provides a lock-free hash table that can be resized based on a double-compare-and-swap (DCAS) operation. However, DCAS, an operation that performs a CAS atomically on two non-adjacent memory locations, is not available on current architectures. Moreover, although Greenwald’s hash table is extensible, it is not a true extensible hash table. The average number of steps per operation is not constant: it involves a helping scheme where that under certain scheduling scenario would lead to a time complexity linearly dependant on the number of processes.

Independently of our work, Gao et al. [2004] have developed a extensible and “almost wait-free” hashing algorithm based on an open addressing hashing scheme and using only CAS operations. Their algorithm maintains the dynamic size by periodically switching to a global resize state in which multiple processes collectively perform the migration of items to new buckets. They suggest performing migration using a write-all algorithm [Hesselink et al. 2001]. Theoretically, each operation in their algorithm requires more than constant time on average because of the complexity of performing the write-all [Hesselink et al. 2001], and so it is not a true extensible hash-table. However, the nonconstant factor is small, and the performance of their algorithm in practice will depend on the yet-untested real-world performance of algorithms for the write-all problem [Hesselink et al. 2001; Kanellakis and Shvartsman 1997].

1.2. THE LOCK-FREE RESIZING PROBLEM. What is it that makes lock-free extensible hashing hard to achieve? The core problem is that even if individual buckets are lock-free, when resizing the table, several items from each of the “old” buckets must be relocated to a bucket among “new” ones. However, in a single CAS operation, it seems impossible to atomically move even a single item, as this requires one to remove the item from one linked list and insert it in another. If this move is not done atomically, elements might be lost, or to prevent loss, will have to be replicated, introducing the overhead of “replication management”. The lock-free techniques for providing the broader atomicity required to overcome these difficulties imply that processes will have to “help” others complete their operations. Unfortunately, “helping” requires processes to store state and repeatedly monitor other processes’ progress, leading to redundancies and overheads that are unacceptable if one wants to maintain the constant time performance of hashing algorithms.

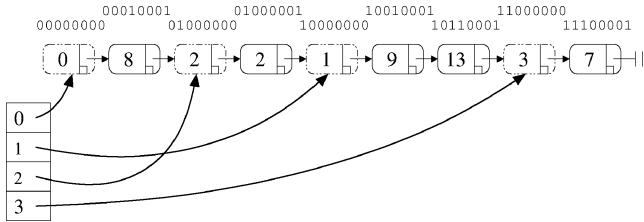


FIG. 1. A split-ordered hash table.

1.3. SPLIT-ORDERED LISTS. To implement our algorithm, we thus had to overcome the difficulty of atomically moving items from old to new buckets when resizing. To do so, we decided to, metaphorically speaking, flip the linear hashing algorithm on its head: our algorithm *will not move the items among the buckets*, rather, it *will move the buckets among the items*. More specifically, as shown in Figure 1, the algorithm keeps all the items in one lock-free linked list, and gradually assigns the bucket pointers to the places in the list where a sublist of “correct” items can be found. A bucket is initialized upon first access by assigning it to a new “dummy” node (dashed contour) in the list, preceding all items that should be in that bucket. A newly created bucket splits an older bucket’s chain, reducing the access cost to its items. Our table uses a modulo 2^i hash (there are known techniques for “pre-hashing” before a modulo 2^i hash to overcome possible binary correlations among values Lea [2003]). The table starts at size 2 and repeatedly doubles in size.

Unlike moving an item, the operation of directing a bucket pointer can be done in a single CAS operation, and since items are not moved, they are never “lost”. However, to make this approach work, one must be able to keep the items in the list sorted in such a way that any bucket’s sublist can be “split” by directing a new bucket pointer within it. This operation must be recursively repeatable, as every split bucket may be split again and again as the hash table grows. To achieve this goal we introduced *recursive split-ordering*, a new ordering on keys that keeps items in a given bucket adjacent in the list throughout the repeated splitting process.

Magically, yet perhaps not surprisingly, recursive split-ordering is achieved by simple *binary reversal*: reversing the bits of the hash key so that the new key’s most significant bits (MSB) are those that were originally its least significant. As detailed below and in the next section, some additional bit-wise modifications must be made to make things work properly. In Figure 1, the split-order key values are written above the nodes (the reader should disregard the rightmost binary digit at this point). For instance, the split-order value of 3 is the bit-reverse of its binary representation, which is 11000000. The dashed-line nodes are the special dummy nodes corresponding to buckets with original keys that are 0, 1, 2, and 3 modulo 4. The split-order keys of regular (nondashed) nodes are exactly the bit-reverse image of the original keys after turning on their MSB (in the example we used 8-bit words). For example, items 9 and 13 are in the “1 mod 4” bucket, which can be recursively split in two by inserting a new node between them.

To *insert* (respectively *delete* or *find*) an item in the hash table, hash its key to the appropriate bucket using recursive split-ordering, follow the pointer to the appropriate location in the sorted items list, and traverse the list until the key’s proper location in the split-ordering (respectively, until the key or a key indicating the item is not in the list) is found. The solution depends on the property that the

items' position is “encoded” in their binary representation, and therefore cannot be generalized to bases other than 2.

As we show, because of the combinatorial structure induced by the split-ordering, this will require traversal of no more than an expected constant number of items. A detailed proof appears in Section 3.

We note that our design is modular: to implement the ordered items list, one can use one of several non-blocking list-based set algorithms in the literature. Potential candidates are the lock-free algorithms of Harris [2001] or Michael [2002a], or the obstruction-free algorithms of Valois²[1995] or Luchangco et al. [2003]. We chose to base our presentation on the algorithm of Michael [2002a], an extension of the Harris algorithm [Harris 2001] that fits well with memory management schemes [Herlihy et al. 2002; Michael 2002b] and performs well in practice.

1.4. COMPLEXITY. When analyzing the complexity of concurrent hashing schemes, there are two adversaries to consider: one controlling the distribution of item keys, the other controlling the scheduling of thread operations. The former appears in all hash table algorithms, sequential or concurrent, while the latter is a direct result of the introduction of concurrency. We use the term *expected time* to refer to the expected number of machine instructions per operation in the worst case scheduling scenario, assuming (as is standard in the literature [Cormen et al. 2001]) a hash function of uniform distribution. We use the term *average time* to refer to the number of machine instructions per operation averaged over all executions, also assuming a uniform hash function. It follows that constant expected time implies constant average time.

As we show in Section 3, if we make the standard assumption of a hash function with a uniform distribution, then under any scheduling adversary our new algorithm provides a lock-free extensible hash table with $O(1)$ average cost per operation.

The complexity improves to expected constant time if we assume a *constant extendibility rate*, meaning that the table is never extended (doubled in size) a non-constant number of times while a thread is delayed by the scheduler. Constant expected time is an improvement over average expected time since it means that given a good hash function, the adversary cannot cause any single operation to take more than a constant number of steps.

One feature in which the new algorithm is similar in flavor to sequential linear hashing algorithms [Litwin 1980] (in contrast to all the above algorithms [Gao et al. 2004; Greenwald 2002; Lea 2003]) is that resizing is done incrementally and only bad distributions (ones that have very low probability given a uniform hash function) or extreme scheduling scenarios can cause the cost of an operation to exceed constant time. This possibly makes the algorithm a better fit for soft real-time applications [Buttazzo et al. 2005] where relaxable timing deadlines need to be met.

1.5. PERFORMANCE. We tested our new *split-ordered list* hash algorithm against the most-efficient known lock-based implementation due to Lea [2003]. We created an optimized C++ based version of the algorithm and compared it to split-ordered lists using a collection of tests executed on a 72-node shared memory machine. We present experiments in Section 4 that show that split-ordered lists

²Valois' algorithm was labeled “lock-free” by mistake. It is livelock-prone.

perform as well as Lea’s algorithms, even in nonmultiprogrammed cases, although lock-free algorithms are expected to benefit systems mainly in multiprogrammed environments. Under high loads, they significantly outperform Lea’s algorithm, exhibiting up to four times higher throughput. They also exhibit greater robustness, for example in experiments where the hash function is biased to create nonuniform distributions.

The remainder of this article is organized as follows: In the next section, we describe the background and the new algorithm in depth. In Section 3, we present the full correctness proof. In Section 4, the empirical results are presented and discussed.

2. The Algorithm in Detail

Our hash table data structure consists of two interconnected substructures (see Figure 1): A linked list of nodes containing the stored items and keys, and an expanding array of pointers into the list. The array entries are the logical “buckets” typical of most hash tables. Any item in the hash table can be reached by traversing down the list from its head, while the bucket pointers provide shortcuts into the list in order to minimize the search cost per item.

The main difficulty in maintaining this structure is in managing the continuous coverage of the full length of the list by bucket pointers as the number of items in the list grows. The distribution of bucket pointers among the list items must remain dense enough to allow constant time access to any item. Therefore, new buckets need to be created and assigned to sparsely covered regions in the list.

The bucket array initially has size 2, and is doubled every time the number of items in the table exceeds $\text{size} \cdot L$, where L is a small integer denoting the *load factor*, the maximum number of items one would expect to find in each logical bucket of the hash table. The initial state of all buckets is *uninitialized*, except for the bucket of index 0, which points to an empty list, and is effectively the head pointer of the main list structure. Each bucket goes through an initialization procedure when first accessed, after which it points to some node in the list.

When an item of key k is inserted, deleted, or searched for in the table, a hash function modulo the table size is used, that is, the bucket chosen for item k is $k \bmod \text{size}$. The table size is always equal to some power 2^i , $i \geq 1$, so that the bucket index is exactly the integer represented by the key’s i least significant bits (LSBs). The hash function’s dependency on the table *size* makes it necessary to take special care as this size changes: an item that was inserted to the hash table’s list before the resize must be accessible, after the resize, from both the buckets it already belonged to and from the new bucket it will logically belong to given the new hash function.

2.1. RECURSIVE SPLIT-ORDERING. The combination of a modulo-size hash function and a 2^i table size is not new. It was the basis of the well known sequential extensible Linear Hashing scheme proposed by Litwin [1980], was the basis of the two-level locking hash scheme of Ellis [1983], and was recently used by Lea [2003] in his concurrent extensible hashing scheme. The novelty here is that we use it as a basis for a combinatorial structure that allows us to repeatedly “split” all the items among the buckets without actually changing their position in the main list.

When the table size is 2^i , a logical table bucket b contains items whose keys k maintain $k \bmod 2^i = b$. When the size becomes 2^{i+1} , the items of this bucket are split into two buckets: some remain in the bucket b , and others, for which $k \bmod 2^{i+1} = b + 2^i$, migrate to the bucket $b + 2^i$. If these two groups of items were to be positioned one after the other in the list, splitting the bucket b would be achieved by simply pointing bucket $b + 2^i$ after the first group of items and before the second. Such a manipulation would keep the items of the second group accessible from bucket b as desired.

Looking at their keys, the items in the two groups are differentiated by the i 'th binary digit (counting from right, starting at 0) of their items' key: those with 0 belong to the first group, and those with 1 to the second. The next table doubling will cause each of these groups to split again into two groups differentiated by bit $i + 1$, and so on. For example, the elements 9 ($1001_{(2)}$) and 13 ($1101_{(2)}$) share the same two least significant bits (01). When the table size is 2^2 , they are both in the same bucket, but when it grows to 2^3 , having a different third bit will cause them to be separated. This process induces *recursive split-ordering*, a complete order on keys, capturing how they will be repeatedly split among logical buckets. Given a key, its order is completely defined by its bit-reversed value.

Let us now return to the main picture: an exponentially growing array of (possibly uninitialized) buckets maps to a linked list ordered by the split-order values of inserted items' keys, values that are derived by reversing the bits of the original keys. Buckets are initialized when they are accessed for the first time. List operations such as `insert`, `delete` or `find` are implemented via a linearizable lock-free linked list algorithm. However, having additional references to nodes from the bucket array introduces a new difficulty: it is nontrivial to manage deletion of nodes pointed to by bucket pointers. Our solution is to add an auxiliary dummy node per bucket, preceding the first item of the bucket, and to have the bucket pointer point to this dummy node. The dummy nodes are not deleted, which helps keep things simple.

In more detail, when the table size is 2^{i+1} , the first time bucket $b + 2^i$ is accessed, a dummy node is created, holding the key $b + 2^i$. This node is inserted to the list via bucket b , the *parent* bucket of $b + 2^i$. Under split-ordering, $b + 2^i$ precedes all keys of bucket $b + 2^i$, since those keys must end with $i + 1$ bits forming the value $b + 2^i$. This value also succeeds all the keys of bucket b that do not belong to $b + 2^i$: they have identical i LSBs, but their bit numbered i is “0”. Therefore, the new dummy node is positioned in the exact location in the list that separates the items that belong to the new bucket from other items of bucket b . In the case where the parent bucket b is uninitialized, we apply the initialization procedure on it recursively before inserting the dummy node. In order to distinguish dummy keys from regular ones we set the most significant bit of regular keys to “1”, and leave the dummy keys with “0” at the MSB. Figure 2 defines the complete split-ordering transformation using the functions `so_regularkey` and `so_dummykey`. The former, reverses the bits after turning on the MSB, and the latter simply performs the bit reversal.³

Figure 3 describes a bucket initialization caused by an insertion of a new key to the set. The insertion of key 10 is invoked when the table size is 4 and buckets 0,1 and 3 are already initialized.

³ An efficient implementation of the REVERSE function utilizes a 2^8 or 2^{16} lookup table holding the bit-reversed values of $[0..2^8 - 1]$ or $[0..2^{16} - 1]$ respectively.

```

so_key_t so_regularkey(key_t key) {
    return REVERSE(key OR 0x8000...0000);
}

so_key_t so_dummykey(key_t key) {
    return REVERSE(key);
}

```

FIG. 2. The Split-Ordering Transformation. The function `so_regularkey` computes the split-order value for regular nodes, where the MSB is set before reversing the bits. The split-order value of dummy nodes is the exact bit reverse of the key.

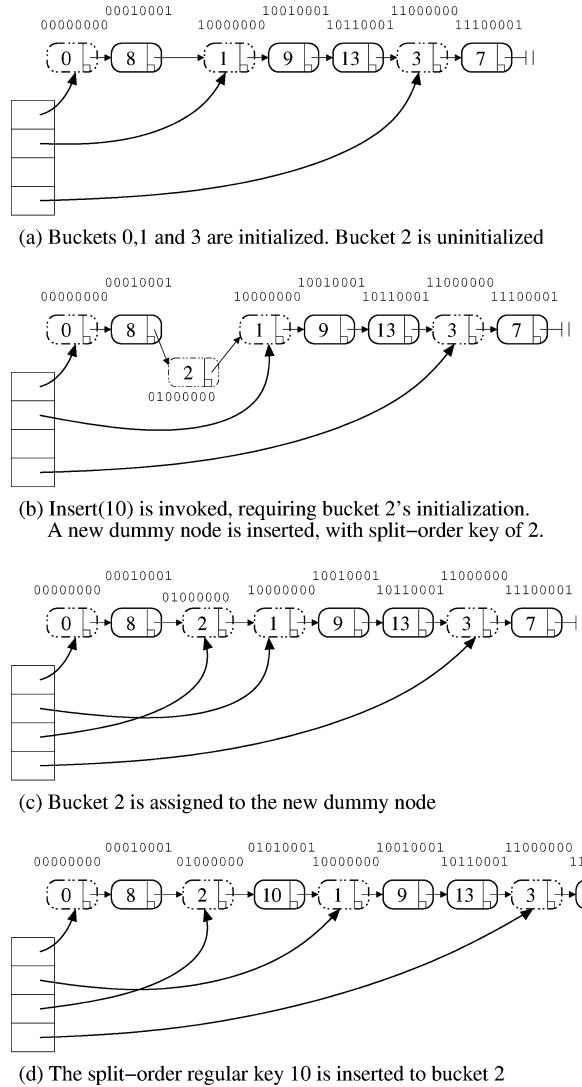


FIG. 3. Insertion into the split-ordered list.

Since the bucket array is growing, it is not guaranteed that the parent bucket of an uninitialized bucket is initialized. In this case, the parent has to be initialized (recursively) before proceeding. Though the total complexity in such a series of recursive calls is potentially logarithmic, our algorithm still works. This is because given a uniform distribution of items, the chances of a logarithmic-size series of recursive initialization calls are low, and in fact, the expected length of such a bad sequence of parent initializations is constant.

2.2. THE CONTINUOUSLY GROWING TABLE. We can now complete the presentation of our algorithm. We use the lock-free ordered linked-list algorithm of Michael [2002a] to maintain the main linked list with items ordered based on the split-ordered keys. This algorithm is an improved variant, including improved memory management, of an algorithm by Harris [2001]. Our presentation will not discuss the various memory reclamation options of such linked-list schemes, and we refer the interested reader to Harris [2001], Herlihy et al. [2002], and Michael [2002a, 2002b]. To keep our presentation self contained, we provide in Appendix A the code of Michael’s linked list algorithm. This implementation is linearizable, implying that each of these operations can be viewed as happening atomically at some point within its execution interval.

Our algorithm decides to double the table size based on the average bucket load. This load is determined by maintaining a shared counter that tracks the number of items in the table. The final detail we need to deal with is how the array of buckets is repeatedly extended. To simplify the presentation, we keep the table of buckets in one continuous memory segment as depicted in Figure 4. This approach is somewhat impractical, since table doubling requires one process to reallocate a very large memory segment while other processes may be waiting. The practical version of this algorithm, which we used for performance testing, actually employs an additional level of indirection in accessing buckets: a main array points to segments of buckets, each of which is a bucket array. A segment is allocated only upon the first access to some bucket within it. The code for this dynamic allocation scheme appears in Section 2.4.

2.3. THE CODE. We now provide the code of our algorithm. Figure 4 specifies some type definitions and global variables. The accessible shared data structures are the array of buckets `T`, a variable `size` storing the current table size, and a counter `count` denoting the number of `regular` keys currently inside the structure.⁴ The counter is initially 0, and the buckets are set as *uninitialized*, except the first one, which points to a node of key 0, whose `next` pointer is set to `NULL`. Each thread has three private variables `prev`, `cur`, and `next`, that point at a currently searched node in the list, its predecessor, and its successor. These variables have the same functionality as in Michael’s algorithm [Michael 2002a]: they are set by `list_find` to point at the nodes around the searched key, and are subsequently used by the same thread to refer to these nodes in other functions. In Figure 5, we show the implementation of the `insert`, `find` and `delete` operations. The `fetch-and-inc` operation can be implemented in a lock-free manner via a simple repeated loop of

⁴Though for the sake of brevity, we do not mention it in the presented code, to reduce contention, we have threads accumulate updates locally and update the shared counter `count` only periodically. We included this optimization in the code used in our benchmarks.

```

struct MarkPtrType {      // Markable pointer type
    <mark, next>; <bool, NodeType* >;
};

struct NodeType {         // Node: contains key and next pointer
    so_key_t key;
    MarkPtrType <mark, next>;
};

/* shared variables */
MarkPtrType* T[ ];          // buckets
uint count;                 // total item count
uint size;                  // current table size

/* thread-private variables */

MarkPtrType *prev;          /* prev */
MarkPtrType <pmark, cur>;  /* curr */
MarkPtrType <cmark, next>; /* next */

```

FIG. 4. Types and Structures. The angular brackets notation denotes a single word type divided to the two fields `mark` and `next`. `mark` is a single bit, while the size of `next` is the rest.

CAS operations, which as we show, given the low access rates, has a negligible performance overhead.

The function `insert` creates a new node and assigns it a split-order key. Note that the keys are stored in the nodes in their split-order form. The bucket index is computed as `key mod size`. If the bucket has not been initialized yet, `initialize_bucket` is called. Then, the node is inserted to the bucket by using `list_insert`. If the insertion is successful, one can proceed to increment the item count using a `fetch-and-inc` operation. A check is then performed to test whether the load factor has been exceeded. If so, the table size is doubled, causing a new segment of uninitialized buckets to be appended.

The function `find` ensures that the appropriate bucket is initialized, and then calls `list_find` on `key` after marking it as regular and inverting its bits. `list_find` ceases to traverse the chain when it encounters a node containing a higher or equal (split-ordered) key. Notice that this node may also be a dummy node marking the beginning of a different bucket.

The function `delete` also makes sure that the `key`'s bucket is initialized. Then it calls `list_delete` to delete `key` from its bucket after it is translated to its split-order value. If the deletion succeeds, an atomic decrement of the total item count is performed.

The role of `initialize_bucket` is to direct the pointer in the array cell of the index bucket. The value assigned is the address of a new dummy node containing the dummy key bucket. First, the dummy node is created and inserted to an existing bucket, `parent`. Then, the cell is assigned the node's address. If the parent bucket is not initialized, the function is called recursively with `parent`. In order to control the recursion, we maintain the invariant that `parent < bucket`, where “`<`” is the regular order among keys. It is also wise to choose `parent` to be as close as possible to `bucket` in the list, but still preceding it. Formally, the following constraints define

```

int insert(so_key_t key) {
I1:  node = new node(so_regularkey(key));
I2:  bucket = key % size;
I3:  if (T[bucket] == UNINITIALIZED)
I4:    initialize_bucket(bucket);
I5:  if (!list_insert(&(T[bucket]), node)) {
I6:    delete_node(node);
I7:    return 0;
}
I8:  csize = size;
I9:  if (fetch-and-inc(&count) / csize > MAX_LOAD)
I10:   CAS(&size, csize, 2 * csize);
I11:  return 1;
}

int find(so_key_t key) {
S1:  bucket = key % size;
S2:  if (T[bucket] == UNINITIALIZED)
S3:    initialize_bucket(bucket);
S4:  return list_find(&(T[bucket]),
                     so_regularkey(key));
}

int delete(so_key_t key) {
D1:  bucket = key % size;
D2:  if (T[bucket] == UNINITIALIZED)
D3:    initialize_bucket(bucket);
D4:  if (!list_delete(&(T[bucket]),
                     so_regularkey(key)))
D5:    return 0;
D6:  fetch-and-dec(&count);
D7:  return 1;
}

void initialize_bucket(uint bucket) {
B1:  parent = GET_PARENT(bucket);
B2:  if (T[parent] == UNINITIALIZED)
B3:    initialize_bucket(parent);
B4:  dummy = new node(so_dummykey(bucket));
B5:  if (!list_insert(&(T[parent]), dummy)) {
B6:    delete dummy;
B7:    dummy = cur;
}
B8:  T[bucket] = dummy;
}

```

FIG. 5. Our split-order-based hashing algorithm.

our the algorithm's choice of *parent* uniquely, where “ $<$ ” is the regular order and “ \prec ” is the split-order among keys:

$$\begin{aligned} \forall k \prec \text{bucket}, (k = \text{parent} \vee k \prec \text{parent}) \\ \text{parent} \prec \text{bucket} \\ \text{parent} < \text{bucket}. \end{aligned}$$

This value is achieved by calling the *GET_PARENT* macro that unsets *bucket*'s most significant turned-on bit. If the exact dummy key already exists in the list, it may

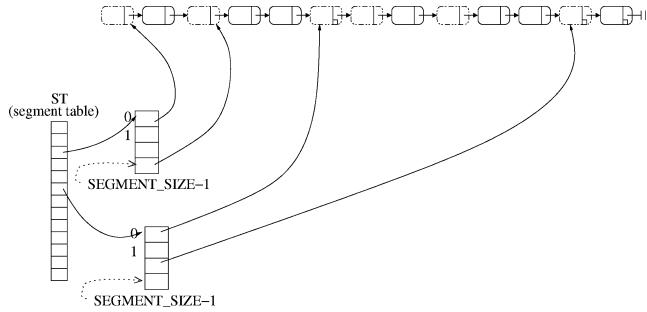


FIG. 6. Structure of the dynamic-sized table.

be the case that some other process tried to initialize the same bucket, but for some reason has not completed the second step. In this case, `list_insert` will fail, but the private variable `cur` will point to the node holding the dummy key. The newly created dummy node can be freed and the value of `cur` used. Note that when line B8 is executed concurrently by multiple threads, the value of `dummy` is the same for all of them.

As we will show in the proof, traversing the list through the appropriate bucket and dummy node will guarantee the node matching a given key will be found, or declared not-found in an expected constant number of steps.

2.4. DYNAMIC-SIZED ARRAY. Our presentation so far simplified the algorithm by keeping the buckets in one continuous memory segment. This approach is somewhat impractical, since table doubling requires one process to reallocate a very large memory segment while other processes may be waiting. In practice, we avoid this problem by introducing an additional level of indirection for accessing buckets: a “main” array points to segments of buckets, each of which is a bucket array. A segment is allocated only on the first access to some bucket within it. The structure of the dynamic-sized hash table is illustrated in Figure 6.

Applying this variation is done by replacing the array of buckets `T` by `ST`, an array of bucket segments, and accessing the table via calls to `get_bucket` and `set_bucket` as defined in Figure 7. Referring to the code of Figure 5, the lines I3, S2, D2, D4, B2, and B5 will use `get_bucket` to access the bucket, and in line B8 `set_bucket` will be called instead of the assignment. Accessing a bucket involves calculating the segment index and then the bucket index within the segment. In `get_bucket`, if the segment has not been allocated yet, it is guaranteed that the bucket was never accessed, and we can return `UNINITIALIZED`. When setting a bucket, in `set_bucket`, if the segment does not exist we have to allocate it and set its pointer in the segment table.

Asymptotically, introducing additional levels of indirection makes the cost of a single access $O(\log n)$. However, one should view the asymptotic in the context of overall memory size, which is bounded. In our case, each level extends the range exponentially with a very high constant, reaching the maximum integer value using a very shallow hierarchy. A level-4 hierarchy can exhaust the memory of a 64-bit machine. Therefore, taking memory size into consideration, the overhead of our construction can be considered as constant.

```

typedef MarkPtrType[SEGMENT_SIZE] segment_t;
segment_t ST[ ]; // the segment table

MarkPtrType * get_bucket(uint bucket) {
    segment = bucket / SEGMENT_SIZE;
    if (ST[segment] == NULL)
        return UNINITIALIZED;
    return &ST[segment][bucket % SEGMENT_SIZE];
}

void set_bucket(uint bucket, NodeType *head) {
    segment = bucket / SEGMENT_SIZE;
    if (ST[segment] == NULL) {
        new_segment = new segment_t;
        new_segment[0..SEGMENT_SIZE-1] =
            UNINITIALIZED;
        if (!CAS(&ST[segment], NULL, new_segment))
            free(new_segment);
    }
    ST[segment][bucket % SEGMENT_SIZE] = head;
}

```

FIG. 7. Dynamic sized array.

3. Correctness Proof

This section contains a formal proof that our algorithm has the desired properties of a resizable hash table. Our model of multiprocessor computation follows [Herlihy and Wing 1990], though for brevity, we will use operational style arguments.

Our linearizable hash table data structure implements an abstract *set* object in a lock-free way so that all operations take an expected constant number of steps on average. Our correctness proof will thus have to prove that our concurrent implementation is linearizable to a sequential set specification, that it is lock-free, and that given a “good” class of hash functions, all operations take an expected constant number of steps on average.

3.1. CORRECT SET SEMANTICS. We begin by proving that the algorithm complies with the abstract set semantics. We use the sequential specification of a “dynamic set with dictionary operations” as defined in Cormen et al. [2001], including the three functions *insert*, *delete* and *find*. The *insert* operation returns 1 if the key was successfully inserted into the set, and 0 if that key already existed in the table. The *find* operation returns 1 if the key is in the set, 0 otherwise. The *delete* operation returns 1 if the key was successfully deleted from the set and 0 if it was not found.

Given a sequential specification of a set, our proof will provide specific linearization points mapping operations in our concurrent implementation to sequential operations so that the histories meet the specification.

Let *list* refer to the non-blocking ordered linked list of all items, pointed to by the buckets of the hash table. Execution histories of our algorithm include sequences of *list_find*, *list_insert*, and *list_delete* operations on this list. Though we argue about these as operations on the shared *list* and not as abstract set operations, our proof will treat these operations as atomic operations. This is a valid approach since they are linearizable by definition of the list-based set algorithms [Harris 2001; Michael 2002a]. We do however need to make additional claims about properties of

operations on the *list*, since we will apply them to various “midpoints” pointed to by buckets, and not only to the start of the list as in the original use of these algorithms of Harris [2001] and Michael [2002a]. To this end, we present the following invariant, which refers to the structure of the list in any state in the execution history of our algorithm.

INVARIANT 1. *In any state:*

- all keys in the list starting at T[0] are sorted in an ascending order.*
- for every $0 \leq i < \text{size}$ if $T[i]$ is initialized, then the node pointed by $T[i]$ holds the key $\text{so_dummykey}[i]$ and is reachable from $T[0]$ by traversing the list following the nodes’ next pointers.*

PROOF. Initially, the invariant holds. We will show that every operation that modifies the data structure preserves the invariant. Lines I9 and D6 manipulate the shared counter, but have no impact on the invariant. Line I10 doubles `size`, which adds new buckets, but since `size` only grows, those new buckets are uninitialized, and the invariant is unaffected.

Assuming that the invariant is true just before line I5, we will show that it is preserved. If `list_insert` fails, the shared state has not changed. Otherwise, we use the induction assumption that $T[\text{bucket}]$ points to a node holding the key $\text{so_dummykey}(\text{bucket})$, and that node is in the list beginning at $T[0]$. The procedure `list_insert` inserts node to the list $T[\text{bucket}]$. This trivially preserves the second condition of the invariant for the bucket. The new node’s key is the bit reverse of key OR $0 \times 800\dots 0$. The array index `bucket` and the value of `key` share the same $\log \text{size}$ least significant bits, while the rest of `bucket`’s bits are 0. Therefore, the new node’s key is ordered after the first node of $T[\text{bucket}]$, whose key is the bit reverse of `bucket`. The first part is also preserved, that is, the list reachable from $T[0]$ remains sorted since all keys before $T[\text{bucket}]$ are by the inductive assumption ordered and have lower keys than $\text{so_dummykey}(\text{bucket})$ and so are properly positioned before the new node, and all other keys are positioned properly by the inductive assumption and the correctness of the `list_insert` operation, since they are a part of the list pointed to by $T[\text{bucket}]$.

The `list_delete` operation of line D4 only deletes a key, and thus cannot affect the order. The deleted node cannot be the first node of $T[\text{bucket}]$, since the least significant bit of its key is 0 and the deleted key’s least significant bit is 1.

The function `list_insert` in line B5 inserts a node with key $\text{so_dummykey}(\text{bucket})$ to the sublist $T[\text{parent}]$, starting with a node holding $\text{so_dummykey}(\text{parent})$. The key `parent` is defined by turning off the index `bucket`’s most significant “1” bit, so the insertion is not before the first node of the sublist starting at $T[\text{parent}]$, and as in the above proof for the case of I5, the invariant is preserved.

Finally, the assignment in B8 sets $T[\text{bucket}]$ to either the dummy node created at B4, or the one assigned at B7. In the first case, since a dummy node created in line B4 is inserted, the second condition of the invariant follows immediately from the correctness of the `list_insert` operation. The first condition follows since the dummy node is inserted in order *after* its parent node which is necessarily ordered before it. In the second case, `list_insert` failed because the key $\text{so_dummykey}(\text{bucket})$ was in the list and `cur` was by the definition of `list_insert` set to the node holding that key, so both parts of the invariant follow. \square

We now define the set H of keys whose items are in the hash table in any given state.

Definition 3.1. For any pointer p , let $S(p)$ be the set of keys in the sorted linked list beginning with the pointer p . Let the *hash table set*

$$H = \{k \mid \text{so_regularkey}(k) \in S(T[0])\}.$$

The set H defines the abstract state of the table. For each one of the hash table operations, we will now show that one can pick a linearization point within its execution interval, so that at this point it has modified the abstract state, that is, the set H , according to the specified operation's semantics. Specifically, we will choose the following linearization points:

- the *insert* operation is linearized in line I5, at the *list_insert* operation,
- the *find* operation is linearized in line S4, at the *list_find* operation, and
- the *delete* operation is linearized in line D4, at the *list_delete* operation.

We start with the following helpful lemma:

LEMMA 3.2. *In lines I5, S4, and D4, $T[\text{bucket}]$ is already initialized, and at B5 $T[\text{parent}]$ is already initialized.*

PROOF. All of the lines above follow a validation that $T[\text{bucket}]$ is initialized. If $T[\text{bucket}]$ is not initialized, *initialize_bucket* is called and the bucket is initialized in B8. \square

Note that, in the proof above, we were not interested in whether the initialization sequence (where initializing a bucket causes initialization of the parent) actually terminates, but rather that if it did terminate then all parents of a bucket were initialized.

LEMMA 3.3. *If key is in H in line I5, then insert fails, and if it is not, insert succeeds and key joins H .*

PROOF. When key is in H , $\text{so_regularkey}(\text{key}) \in S(T[0])$. According to Lemma 3.2, $T[\text{bucket}]$ is initialized, and using Invariant 1, we conclude that the node pointed by $T[\text{bucket}]$ has the key $\text{so_dummykey}(\text{bucket})$ and it is a part of the list. The list is sorted, and

$$\begin{aligned} \text{so_dummykey}(\text{bucket}) &= \text{REVERSE}(\text{bucket}) = \\ \text{REVERSE}(\text{key mod size}) &< \text{REVERSE}(\text{key OR } 0x800..0) = \\ &\quad \text{so_regularkey}(\text{key}). \end{aligned} \tag{1}$$

Thus, the searched key is in the sublist, $S(T[\text{bucket}])$. The *list_insert* at I5 will fail and so will *insert*. If key is not in H , it is also not in $S(T[\text{bucket}])$, and *list_insert* inserts $\text{so_regularkey}(\text{key})$ in the bucket's sublist. From that state on, $\text{so_regularkey} \in S(T[0])$, that is, key is in H . \square

LEMMA 3.4. *If key is in H at line S4, the find succeeds, and otherwise the find fails.*

PROOF. If line S4 is executed when key is in H , then $\text{so_regularkey}(\text{key})$ is in $S(T[0])$. $T[\text{bucket}]$ is assigned to a node in that list, holding the key $\text{so_dummykey}(\text{bucket})$. Using Eq. (1), we conclude that the searched key is in

$S(T[\text{bucket}])$, so `list_find` succeeds and so does `find`. If in line S4 key is not in H , it cannot be in $S(T[\text{bucket}])$, so `list_find` fails. \square

LEMMA 3.5. *If key is in H in line D4, `delete` succeeds and removes key from H , and otherwise `delete` fails.*

PROOF. If key is in H , then `so_regularkey(key)` is in $S(T[0])$. $T[\text{bucket}]$ is assigned to a node inside that list, where the key of that node is `so_dummykey(bucket)`. Using Eq. (1), we conclude that the searched key is in $S(T[\text{bucket}])$, so `list_delete` removes it. If key is not in H , it cannot be in $S(T[\text{bucket}])$, so `list_delete` fails. \square

From Lemma 3.3, Lemma 3.4, and Lemma 3.5, it follows that:

THEOREM 3.6. *The split-ordered list algorithm of Figure 5 is a linearizable implementation of a set object.*

3.2. LOCK FREEDOM. Our algorithm uses loads and stores together with implementations of a list-based set, a shared counter, and memory allocation routines as primitive objects/operations. As we will show, in terms of these primitive operations the algorithm's implementation is wait-free, that is, each thread always completes in a finite number of operations. This implies that its overall progress condition in terms of primitive machine operations will be exactly that of the underlying implementation of those objects. Since we used the lock-free list-based sets of Harris [2001] and Michael [2002a] and a lock-free shared counter as building blocks in this presentation, our implementation will also be lock-free. As noted in the introduction, in some cases, there are advantages in using the obstruction free list-based set algorithm of Luchangco et al. [2003]. If Luchangco et al. [2003] is used together with a lock-free shared counter, our hash table will be obstruction-free [Herlihy et al. 2003].

THEOREM 3.7. *The split-ordered list algorithm of Figure 5 is a wait-free implementation of a set object in terms of load, store, fetch-and-inc, fetch-and-dec, `list_find`, `list_insert` and `list_delete` operations.*

PROOF. The functions `insert`, `find`, `delete` and `initialize_bucket` all take a finite number of steps, each of which is a machine level load or store operation or an operation on the list based set object or the shared counter. The `initialize_bucket` procedure is the only one with a recursive call. However, the recursion of `initialize_bucket` is limited, since each step is executed on the parent of a bucket, which satisfies $\text{parent} < \text{bucket}$. Since bucket 0 is initialized from the start, the recursion is finite, and the implementation is wait-free. \square

The lock-freedom property means that a thread executing the hash table operation completes in a finite number of steps unless other threads are infinitely making progress. Thus, it is a weaker requirement than wait-freedom, and by combining implementations the following is a corollary of Theorem 3.7:

COROLLARY 3.8. *The split-ordered list algorithm of Figure 5 with lock-free implementations of `list_find`, `list_insert`, `list_delete`, `fetch-and-inc`, and the `fetch-and-dec` operations is lock-free.*

COROLLARY 3.9. *The split-ordered list algorithm of Figure 5 with obstruction-free implementations of fetch-and-inc, fetch-and-dec, list_find, list_insert and list_delete operations is obstruction-free.*

The fetch-and-inc and fetch-and-dec operations have known lock-free implementations [Michael and Scott 1998].

3.3. COMPLEXITY. The most important property of a hash table is its expected constant time performance. When analyzing the complexity of hashing in a concurrent environment there are two adversaries one needs to consider: one controlling the distribution of hash values of keys by the hash function (i.e., how good is the hash), the other controlling the scheduling of thread operations. We will follow the standard practice of modelling the hash function as a uniform distribution over keys [Cormen et al. 2001]. The uniformity of keys we assume is global, that is, it extends across all threads in a given execution (A simple way to think of this is that we apply the standard uniform distribution assumption [Cormen et al. 2001] on the linearization of any given execution). We will use the term *expected time* (or *expected number of steps*) to refer to the expected number of machine instructions per operation in the worst case scheduling scenario, assuming a hash function of uniform distribution. We will use the term *average time* (or *average number of steps*) to refer to the number of machine instructions per operation averaged over all executions, also assuming a uniform hash function. It follows that constant expected time implies constant average time.

In our complexity analysis, we assume that loops within the underlying linked list code involve no more than a constant number of retries. This assumption is realistic since a nonconstant number of retry loops implies *Compare & Swap* failures caused by contention within a single bucket, which cannot occur due to the global uniformity of the hash function.

We will show that under any scheduling adversary, our algorithm performs all hash table operations in constant average time. The complexity improves to constant expected time if we assume a *constant extendibility rate*. This is a restriction on the scheduler that requires that the table is never forced to extend a nonconstant number of times while a thread is delayed by the scheduler. It means that given a good hash function, the adversary cannot cause any single operation to take more than a constant number of steps unless it delays its progress through more than a constant number of global resize operations. Formally, when there are n items in the data structure, a thread must complete a single operation before $n \cdot 2^c$ successful insertions of elements by other threads were completed, where $c \in O(1)$. We believe this is the common situation in practice.

Two algorithmic issues require a detailed proof: one is the complexity of list operations, which is essentially the complexity of executing a `list_find`, and the other is the complexity of `initialize_bucket`, which involves recursive calls.

Denote by n the total number of items in the set, and by s the number of buckets. For the complexity analysis, we are not interested in the cases where the table is small, so we make the assumption that s is greater than the number of threads. Let L denote the load factor `MAX_LOAD` in our code, typically a small constant.

LEMMA 3.10. *For any number p of threads, at all times the following condition holds:*

$$\frac{n - p}{s} \leq L.$$

PROOF. Focus on the successful completed `insert` and `delete` operations. Each successful insertion incremented `count` by 1, and each successful deletion decremented it. In any state, there are no more than p concurrent operations. Every one of the “already completed” `insert` operations checked, when executing line I9, that the ratio of `count` and `csize` is not more than L , and doubled the `size` if the gap was exceeded. At all times, there are no more than p currently executing `insert` operations. Therefore, when $n/s > L$ and a resize is needed, no more than p new keys can be inserted to the data structure before the resize takes place. \square

LEMMA 3.11. *Assuming a hash function of uniform distribution, the probability that a bucket is not accessed during the time where the table size is s , is asymptotically bounded by $\exp(-L/2)$.*

PROOF. Focus on a growing table from size $s/2$ to s and then to $2s$. According to Lemma 3.10, in the state in which line I10 doubled the table from $s/2$ to s , the number of items in the table was less or equal to $p + Ls/2$. When later in line I10 the table doubled in size to $2s$, the condition of line I9 implies that the number of items was at least Ls . The last two observations imply that during the set of states in which `size` was s , the item count increased by at least $Ls/2 - p$, that is, line I9 was executed at least $Ls/2 - p$ times. When we consider at most p processes that may have begun the `insert` operation when `size` was less than s , we get that line I2 was executed at least $Ls/2 - 2p$ times.

Assuming a uniform distribution of the keys, the probability that a bucket b was not accessed during this period is at most $(\frac{s-1}{s})^{Ls/2-2p}$. When p is significantly smaller than s , as assumed, the last expression is asymptotically equal to $\exp(-L/2)$. \square

LEMMA 3.12. *For any key k , when the table size is s and the bucket $k \bmod \text{size}$ is initialized, there is no dummy node with key d such that $k \bmod \text{size} < d < k$, that is, d 's split-order value is between those of $k \bmod \text{size}$ and k .*

PROOF. Assume by way of contradiction that d is the key of a node such that: $k \bmod \text{size} < d < k$. It is the case that $d < \text{size}$ because d is in the list, and bucket indices are always smaller than the table size. Therefore, d has less than $\log_2(\text{size})$ non-zero bits. The keys k and $k \bmod \text{size}$ have at least $\log_2(\text{size}) - 1$ identical less significant bits. The split-order value of d is between them, so it must have the same low $\log_2(\text{size}) - 1$ bits, that actually constitute all of its non-zero bits. This implies that $d = k \bmod \text{size}$ under the split-order, a contradiction to the assumption that $d > k \bmod \text{size}$. \square

LEMMA 3.13. *If the hash function distributes the keys uniformly then:*

- In any execution history, the list traversal of `list_find` takes constant time on average.
- Under the constant extendibility rate assumption, the traversal of `list_find` takes expected constant time.

PROOF. For a table of size s , the expected number of uninitialized buckets among the first $s/2$ buckets is no more than $s/2 \cdot \exp(-L/2)$, by Lemma 3.11. For each of the initialized buckets, there is a dummy node in the list holding the bucket index as the split-order value. Therefore, there are at least $s/2 \cdot (1 - \exp(-L/2))$ dummy nodes with keys from $0..s/2 - 1$. Those values divide the integer range into $s/2$ equal segments, while the missing items are distributed evenly. Using Lemma 3.10, there are on average less than

$$\begin{aligned} \frac{n}{s/2 \cdot (1 - \exp(-L/2))} &\leq \frac{Ls + p}{s/2 \cdot (1 - \exp(-L/2))} \\ &= \frac{2L + 2p/s}{1 - \exp(-L/2)} \end{aligned} \quad (2)$$

nodes between every two dummy nodes. The operation `list_find` is called to search for a key k from the bucket $k \bmod \text{size}$, so, using Lemma 3.12, we conclude that in the state in which it was called there were no dummy nodes between the bucket's dummy node and the node at which the search would be completed. We have just computed that dummy nodes are distributed in intervals of less than

$$\frac{2L + 2p/s}{1 - \exp(-L/2)}$$

nodes, implying that if the table size does not change, the search will take no more than a constant expected number of steps.

We will now show that if the search took more than constant time, there were enough successful inserts to maintain a constant number of steps on average. If `list_find` took $\Omega(r)$ steps, $\Omega(r)$ dummy nodes must have been traversed, since at any time the expected distance between them is constant. All of these dummy nodes were inserted to the list after `list_find` started. The number of dummy nodes in the original bucket doubles each time the table is extended, so there were $\Omega(\log r)$ table resize events. Since there were exactly n items in the table when the `list_find` operation started, the number of items had to rise by $\Omega(rn)$, that is, $\Omega(rn)$ successful insertions to the list. There were no more than p threads that successfully executed `list_insert` but then were delayed before completing the `insert` routine. Therefore, we can consider only $\Omega(rn - p)$ as complete hash table insertions. According to the constant extensibility rate assumption, a thread must complete a single operation within $n \cdot 2^c$ successful insertions. Looking at the single operation that took $\Omega(r)$ steps, we now know that during that time there were at least $\Omega(rn - p)$ successful inserts, but we also know that the operation lasted less than $n \cdot 2^c$ successful operations. We get that $\log(r - p/n) \in O(1)$, and thus $r \in O(1)$. \square

LEMMA 3.14. *Given a hash function with an expected uniform distribution, the number of steps performed by the function `initialize_bucket` is constant on average. Under the constant extendibility rate assumption, the number of expected steps in the worst case execution is constant.*

PROOF. A recursive call to `initialize_bucket` terminates when the parent bucket is initialized. To have m recursive calls, m uninitialized ancestor buckets are needed. Applying Lemma 3.11, this may happen with probability less than $\exp(-L(m - 1)/2)$. The number of m -deep executions among m calls to

`initialize_bucket` is $m \cdot \exp(-L(m-1)/2) \in O(1)$, implying that the expected number of recursive calls is constant. By Lemma 3.13, the `list_insert` call inside `initialize_bucket` costs a constant number of steps on average. If we assume constant extendibility rate (threads are not delayed while the table is doubled a nonconstant number of times), a recent ancestor of every bucket is always initialized, and the recursion depth is constant. Also, according to Lemma 3.13, the execution of `list_insert` is of expected constant time. \square

THEOREM 3.15. *Given a hash function with expected uniform distribution, all hash table operations complete within a constant number of steps on average. Assuming a constant extendibility rate, all hash table operations complete within expected constant number of steps.*

PROOF. Beside executing a constant number of simple instructions, all hash operations call a list traversing routine twice at most (actually, only `hash_delete` may cause `list_find` to run twice). By Lemma 3.13, the list traversals cost a constant average number of steps, and by Lemma 3.14, the `initialize_bucket` operation also completes within a constant average number of steps. Both of the above lemmas imply that under the constant extendibility rate assumption, the number of steps is constant in the worst case execution assuming a uniform distribution. \square

4. Performance

We ran a series of tests to evaluate the performance of our lock-free algorithm. Since our algorithm is the first lock-free extensible hash table, it needs to be proven efficient in comparison to existing lock-based extensible hash table algorithms. We have thus chosen to compare our algorithm to the resizable hash table algorithm of Lea [2003] (revision 1.3), originally suggested as a part of *util.concurrent.ConcurrentHashMap*, the proposed JavaTM Concurrency Package, JSR-166.

Lea's algorithm is based on an exponentially growing table of buckets, doubled when the average bucket load exceeds a given load factor. Access to the table buckets is synchronized by 64 locks, dividing the bucket range to 64 interleaved regions, that is, lock i is obtained when bucket b is accessed if $b \bmod 64 = i$. `Insert` and `delete` operations always acquire a lock, but `find` operations are first attempted without locking, and retried with locking upon failure. When a process decides to resize the table, it locks all 64 locks, allocates a larger array and rehashes the buckets' items to their new buckets, utilizing the simplicity of power-of-two hashing. This scheme offers good performance, in comparison to simpler schemes that separately lock each bucket, by significantly reducing the number of locks that need to be acquired when resizing. Figure 8 illustrates the effect of different concurrency levels on Lea's algorithm performance.

We translated the JavaTM code by Lea to C++ and simplified it to handle integer keys that also serve as values, exactly as in our new algorithm's code. There is a trade-off in this algorithm: the more locks used, the lower the contention on them, but the higher the global delay when resizing. We thus ran an experiment to confirm that in the translated algorithm there is no significant advantage to using more or less than 64 locks.

We compared our split-ordered hashing algorithm to Lea's algorithm using a collection of experiments on a 30-processor Sun Enterprise 6000, a cache-coherent

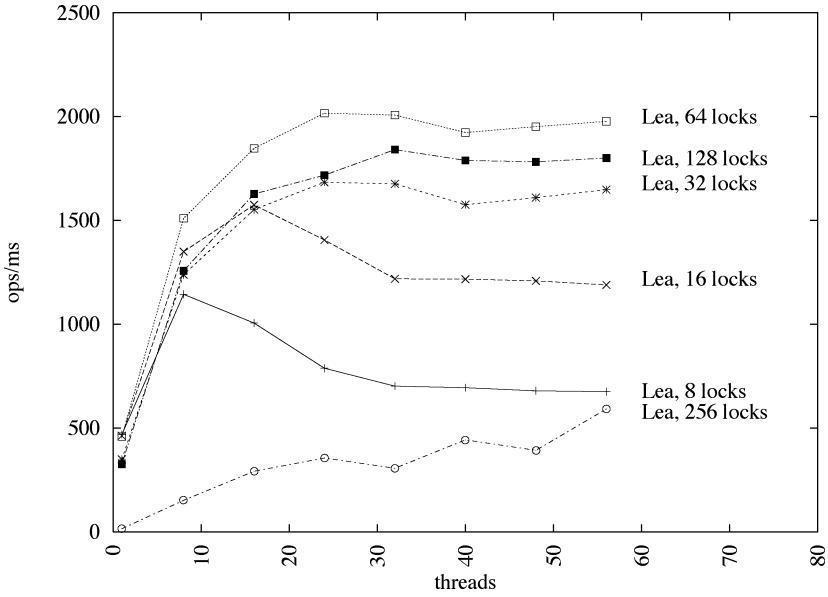


FIG. 8. Lea's algorithm with different concurrency levels.

NUMA machine formed from 15 boards of two 300 MHz UltraSPARC® II processors and 2 GB of RAM on each. The C/C++ code was compiled with a Sun *cc* compiler 5.3, with the flags *-x05* and *-xarch=v8plusa*. We executed each experiment three times to lower the effect of temporary scheduling anomalies.

Lea's algorithm has significant vulnerability in multiprogrammed environments since whenever the resizing processor is swapped out or delayed, the algorithm as a whole grinds to a halt. The significant latency overhead while resizing would also make it less of a fit for real-time environments. However, our tests here are designed to compare the performance of the algorithms in the currently more common environments without multiprogramming or real-time requirements.

Since Lea's algorithm behaves differently when hash table operations fail rather than succeed, we also tested the algorithms in scenarios where they begin after a significant amount of elements have been inserted. Since the range from which the elements are selected is limited, the more we pre-insert, the more chances are that an element is already in the table when search for it. Additionally, we ran a series of experiments measuring the change in throughput as a function of concurrency under various synthetic distributions of *insert*, *delete* and *find*.

To capture performance under typical hash-table usage patterns [Lea (personal communication, 2003)], we first look at a mix that consists of about 88% *find* operations, 10% *inserts* and 2% *deletes*. Our first graph, in Figure 9, shows the results of comparing the algorithms under such a pattern. The hash table load factor (the number of items per bucket) for both tested algorithms was chosen as 3. In the presented graph we show the change in throughput as a function of concurrency. As can be seen, at high loads the lock-free split-ordered hashing algorithm significantly outperforms Lea's when the concurrency level goes beyond eight threads.

The first data point, corresponding to the throughput when executed by a single thread, is a measure for the overhead cost of the new algorithm. According to this data point, the new algorithm is 23% slower than the lock-based algorithm when

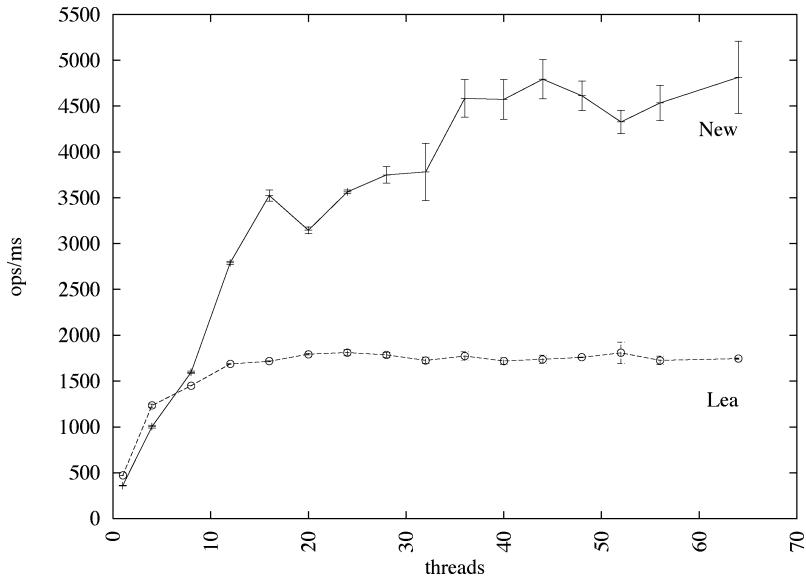


FIG. 9. Throughput of both algorithms. Standard deviation is denoted by vertical bars.

run by a single thread.

- Lea’s algorithm reaches peak performance at about 24 threads and at the same concurrency level, our new algorithm has two times higher throughput.
- Our algorithm reaches peak performance at 44 threads, where it is almost three times faster than Lea’s.
- Our algorithm’s performance fluctuates after reaching peak performance because it involves significantly higher concurrent communication and is thus much more sensitive to the specific layout of threads on the machine and to the load on the shared crossbar.
- Lea’s algorithm suffers a much milder deterioration caused by the architectural critical paths because it never reaches high concurrency levels and its overall performance is limited by the bottlenecks introduced by the shared locks.

Figure 10 shows the results of an experiment varying the chosen distribution of inserts, deletes, and finds. Note that our algorithm consistently outperforms Lea’s algorithm throughout the full range of tested distributions. We also ran an experiment that varies the load factor in our algorithm. As seen in Figure 11, the load factor does not affect the performance significantly, and its effect is in any case minimal when compared to those of the thread layout and the overall communication overhead.

Figure 12 shows the throughput of both algorithms when the amount of pre-insertions varied among 0, 300 K, 600 K, and 900 K. The range from which elements were selected was $[0, 1e + 6]$, so pre-insertions affected significantly the success rate of the hash table operations. The performance of Lea’s algorithm slightly improves on lower concurrency levels, but from 12 threads and on the new algorithm is faster.

We also tested the robustness of the algorithms under a biased hash function, mimicking conditions in case of a bad choice of a hash function relative to the

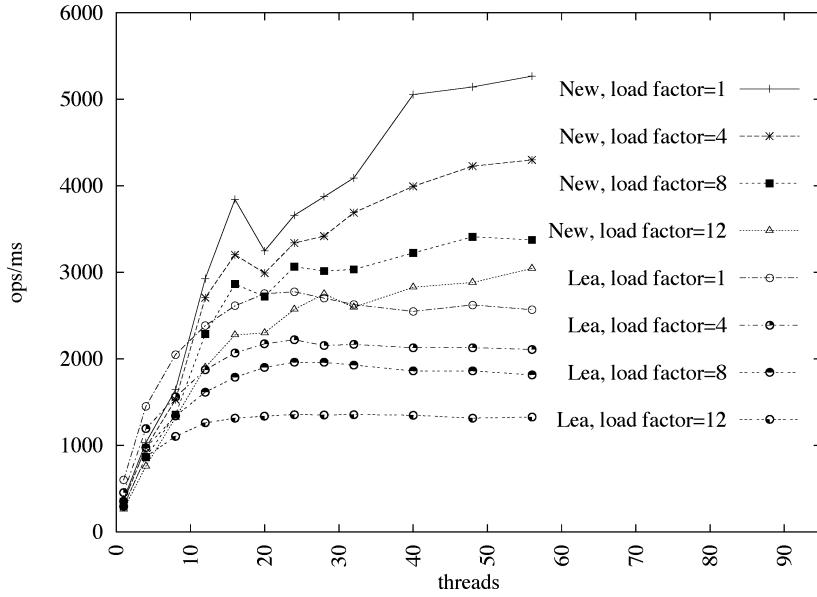


FIG. 10. Varying operation distribution.

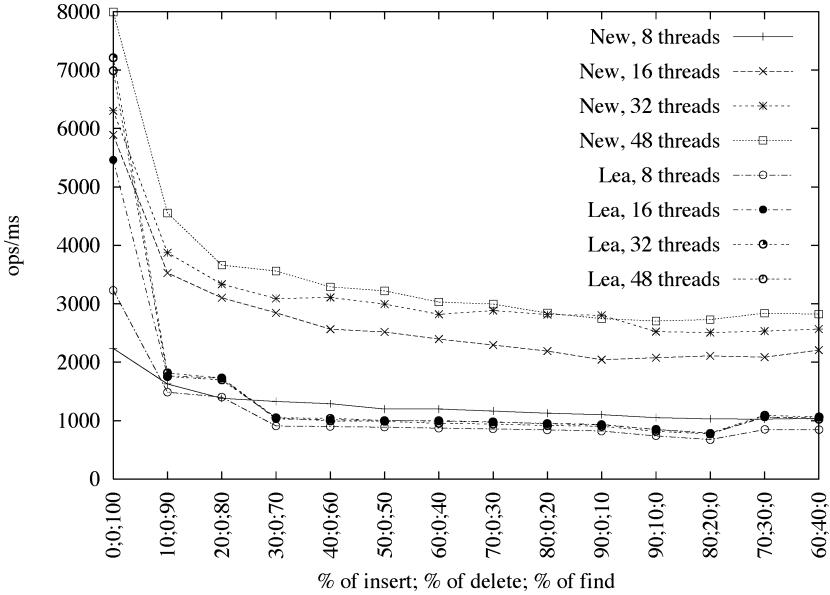


FIG. 11. Varying load factor.

given data. To do so we generated keys in a nonuniform distribution by randomly turning off 0 to 3 LSBs of randomly chosen integers. Our empirical data shows that our algorithm shows greater robustness: it was slowed down by approximately 7%, while Lea's algorithm's performance decreased by more than 30%. The reason for this is that a biased hash function causes some number of buckets to have many more items than the average load. The locks controlling these buckets in Lea's

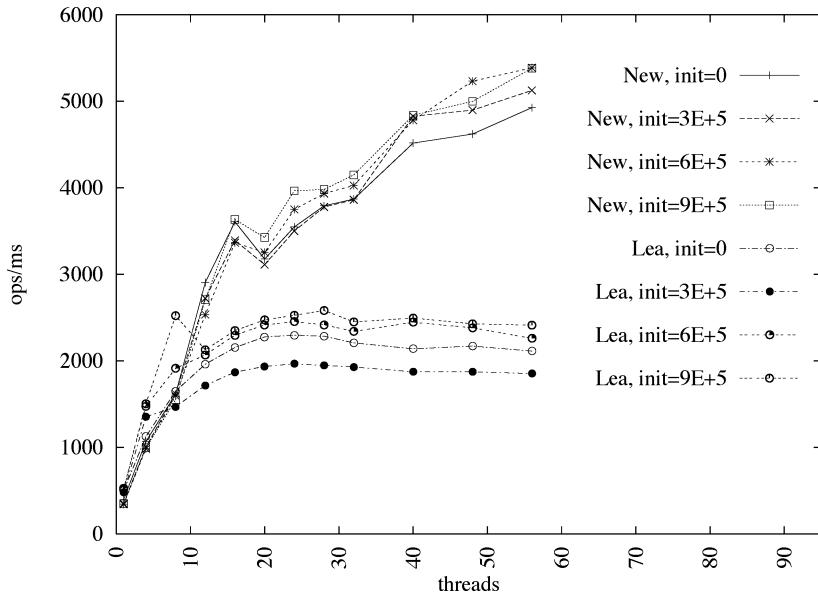


FIG. 12. Varying amount of pre-insertions.

algorithm are thus contended, causing a performance degradation. This does not happen in the lock-free list used by the new algorithm.

Based on the above results, we conclude that in low-load nonmultiprogrammed environments both algorithms offer comparable performance, while under medium to high loads, split-ordered hashing scales better than Lea's algorithm and is thus the algorithm of choice.

5. Conclusion

Our article introduced split-ordered lists and showed how to use them to build resizable concurrent hash tables. We believe the split-order list structure may have broader applications, and in particular it might be interesting to test empirically if a purely sequential variation of split-ordered hashing will offer an improvement over linear hashing in the sequential case. This follows since splitting buckets in split-ordered hash tables does not require redistribution of individual items among buckets, but rather only the insertion of a dummy node, and in the sequential case the need for the dummy nodes might be avoidable altogether.

Appendix

A. Additional Code

For the purpose of being self contained, we provide in Figures 13 and 14 the code for the lock-free CAS-based ordered list algorithm of Michael [2002a].

The difficulty in implementing a lock-free ordered linked list is in ensuring that during an insertion or deletion, the adjacent nodes are still valid, that is, they are still in the list and are still adjacent. Both the implementation of Harris [2001] and that of Michael [2002a] do so by “stealing” one bit from the pointer to mark a node as deleted, and performing the deletion in two steps: first marking the node,

```

struct MarkPtrType {
    <mark, next>: <bool, NodeType *>
};

struct NodeType {
    key_t key;
    MarkPtrType <mark, next>;
};

/* thread-private variables */
MarkPtrType *prev;
MarkPtrType <pmark, cur>;
MarkPtrType <cmark, next>;

int list_insert(MarkPtrType *head,
                NodeType *node) {
    key = node->key;
    while (1) {
        if (list_find(head, key)) return 0;
        node-><mark,next> = <0,cur>;
        if (CAS(prev, <0,cur>, <0,node>))
            return 1;
    }
}

int list_delete(MarkPtrType *head,
                so_key_t key) {
    while (1) {
        if (!list_find(head, key))
            return 0;
        if (!CAS(&(cur-><mark,next>), <0,next>,
                 <1,next>))
            continue;
        if (CAS(prev, <0,cur>, <0,next>))
            delete_node(cur);
        else list_find(head, key);
        return 1;
    }
}

```

FIG. 13. Michael's lock free list based sets.

and then deleting it. This bit and the *next* pointer are set atomically by the same CAS operation.⁵ The *list_find* operation is the most complicated: it traverses through the list, and stops when it reaches an item that is equal-to or greater-than the searched item. If a marked-for-deletion node is encountered, the deletion is completed and the traversal continues. The *list_find* in Michael's scheme thus improves on that of Harris since by completing the deletion immediately when a marked node is encountered it prevents other operations from traversing over marked nodes, that is, ones that have been logically deleted.

⁵Stealing one bit in a pointer in such a manner is straightforward assuming properly aligned memory, and can be achieved with indirection using a “dummy bit node” [Agesen et al. 2000] in languages like Java™ where stealing a bit in a pointer is a problem. The new Java™ Concurrency Package proposes to eliminate this drawback by offering “tagged” atomic variables.

```

int list_find(NodeType **head, so_key_t key) {
try_again:
    prev = head;
    <pmark,cur> = *prev;
    while (1) {
        if (cur == NULL) return 0;
        <cmark,next> = cur-><mark,next>;
        ckey = cur->key;
        if (*prev != <0,cur>)
            goto try_again;
        if (!cmark) {
            if (ckey >= key)
                return ckey == key;
            prev = &(cur-><mark,next>);
        }
        else {
            if (CAS(prev, <0,cur>, <0,next>))
                delete_node(cur);
            else goto try_again;
        }
        <pmark,cur> = <cmark,next>;
    }
}

```

FIG. 14. Michael's lock free list based sets—continued.

```

int fetch-and-inc(int *p) {
    do {
        old = *p;
    } while (!CAS(p, old, old+1));
    return old;
}

int fetch-and-dec(int *p) {
    do {
        old = *p;
    } while (!CAS(p, old, old-1));
    return old;
}

```

FIG. 15. Lock free atomic counter implementation.

Figure 15 depicts a simple lock-free implementation of a shared incrementable (or decrementable) counter using CAS.

ACKNOWLEDGMENTS. We thank Mark Moir, Victor Luchangco and Paul Martin for their help and patience in accessing and running our tests on several of Sun's large multiprocessor machines. This paper could not have been completed without them. We also thank Victor Luchangco, Mark Moir, Maged Michael, Sivan Toledo, and the anonymous PODC 2003 referees for their helpful comments and insights. We thank Doug Lea for his constructive skepticism and for sharing with us real-world data on the growth characteristics of dynamic hash tables. Finally, the comments of the anonymous referees assisted greatly in improving this manuscript.

REFERENCES

- AGESEN, O., DETLEFS, D., FLOOD, C., GARTHWAITE, A., MARTIN, P., SHAVIT, N., AND STEELE, G. 2000. DCAS-based concurrent deques. In *Proceedings of the 12th Annual ACM Symposium on Parallel Algorithms and Architectures*. ACM, New York.

- BUTTAZZO, G., LIPARI, G., ABENI, L., AND CACCAMO, M. 2005. *Soft Real-Time Systems: Predictability vs. Efficiency*. Series: Series in Computer Science. Springer-Verlag, New York.
- CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. 2001. *Introduction to Algorithms, Second Edition*. MIT Press, Cambridge, MA.
- ELLIS, C. S. 1983. Extendible hashing for concurrent operations and distributed data. In *Proceedings of the 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*. ACM, New York, 106–116.
- ELLIS, C. S. 1987. Concurrency in linear hashing. *ACM Trans. Database Syst.* 12, 2, 195–217.
- GAO, H., GROOTE, J., AND HESSELINK, W. 2004. Almost wait-free resizable hashtables. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPOPS)*.
- GREENWALD, M. 1999. Non-blocking synchronization and system design. Ph.D. dissertation. Stanford University Tech. Rep. STAN-CS-TR-99-1624, Palo Alto, CA.
- GREENWALD, M. 2002. Two-handed emulation: How to build non-blocking implementations of complex data-structures using DCAS. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing*. ACM, New York, 260–269.
- HARRIS, T. L. 2001. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of 15th International Symposium on Distributed Computing (DISC 2001)*, 300–314.
- HERLIHY, M., LUCHANGCO, V., AND MOIR, M. 2002. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures. In *Proceedings of 16th International Symposium on Distributed Computing (DISC 2002)*, 339–353.
- HERLIHY, M., LUCHANGCO, V., MOIR, M., AND SCHERER, III, W. N. 2003. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing*. ACM, New York, 92–101.
- HERLIHY, M. P., AND WING, J. M. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 3, 463–492.
- HESSELINK, W., GROOTE, J., MAUW, S., AND VERMEULEN, R. 2001. An algorithm for the asynchronous write-all problem based on process collision. *Distrib. Comput.* 14, 2, 75–81.
- HSU, M., AND YANG, W. 1986. Concurrent operations in extendible hashing. In *Proceedings of the 12th International Conference on Very Large Data Bases (VLDB'86)* (Kyoto, Japan, Aug. 25–28). W. W. Chu, G. Gardarin, S. Ohsuga, and Y. Kambayashi, Eds. Morgan-Kaufmann, San Francisco, CA, 241–247.
- KANELAKIS, P. C., AND SHVARTSMAN, A. 1997. *Fault-Tolerant Parallel Computation*. Kluwer Academic Publishers.
- LEA, D. 2003. Hash table util.concurrent.ConcurrentHashMap, revision 1.3, in JSR-166, the proposed Java Concurrency Package. <http://gee.cs.oswego.edu/cgi-bin/viewcvs.cgi/jsr166/src/main/java/util/concurrent/>.
- LITWIN, W. 1980. Linear hashing: A new tool for file and table addressing. In *Proceedings of the 6th International Conference on Very Large Data Bases (VLDB'80)* (Montreal, Que., Canada, Oct. 1–3). IEEE Computer Society, Press, Los Alamitos, CA, 212–223.
- LUCHANGCO, V., MOIR, M., AND SHAVIT, N. 2003. Nonblocking k -compare single swap. In *Proceedings of the 15th Annual ACM Symposium on Parallel Algorithms and Architectures*. ACM, New York.
- MELLOR-CRUMMEY, J. M., AND SCOTT, M. L. 1991. Scalable reader-writer synchronization for shared-memory multiprocessors. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 106–113.
- MICHAEL, M. M. 2002a. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures*. ACM, New York, 73–82.
- MICHAEL, M. M. 2002b. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the 21st Annual Symposium on Principles of Distributed Computing*. ACM, New York, 21–30.
- MICHAEL, M. M., AND SCOTT, M. L. 1998. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared-memory multiprocessors. *J. Parall. Distrib. Comput.* 51, 1, 1–26.
- MOIR, M. 1997. Practical implementations of non-blocking synchronization primitives. In *Proceedings of the 15th Annual ACM Symposium on the Principles of Distributed Computing*. ACM, New York.
- VALOIS, J. D. 1995. Lock-free linked lists using compare-and-swap. In *Proceedings of the Symposium on Principles of Distributed Computing*. ACM, New York, 214–222.

RECEIVED MARCH 2004; REVISED SEPTEMBER 2005; ACCEPTED FEBRUARY 2006

Lock-free Cuckoo Hashing

Nhan Nguyen, Philippas Tsigas
*Chalmers University of Technology
Gothenburg, Sweden*
Email: {nhann, tsigas}@chalmers.se

Abstract—This paper presents a lock-free cuckoo hashing algorithm; to the best of our knowledge this is the first lock-free cuckoo hashing in the literature. The algorithm allows mutating operations to operate concurrently with query ones and requires only single word compare-and-swap primitives. Query of items can operate concurrently with others mutating operations, thanks to the two-round query protocol enhanced with a logical clock technique. When an insertion triggers a sequence of key displacements, instead of locking the whole cuckoo path, our algorithm breaks down the chain of relocations into several single relocations which can be executed independently and concurrently with other operations. A fine tuned synchronization and a helping mechanism for relocation are designed. The mechanisms allow high concurrency and provide progress guarantees for the data structure’s operations. Our experimental results show that our lock-free cuckoo hashing performs consistently better than two efficient lock-based hashing algorithms, the chained and the hopscotch hash-map, in different access pattern scenarios.

I. OVERVIEW

A hash table is a fundamental data structure which offers rapid storage and retrieval operations. Hash tables are widely used in many computer systems and applications. Papers in the literature have studied several hashing schemes which differ mainly in their methods to resolve hash conflicts. As multi-core computers become ubiquitous, many works have also targeted the parallelization of hash tables to achieve high performance and scalable concurrent ones.

Cuckoo hashing [1] is an open address hashing scheme which has a simple conflict resolution. It uses two hash tables that correspond to two hash functions. A key is stored in one of the tables but not in both. The addition of a new key is made to the first hash table using the first hash function. If a collision occurs, the key currently occupying the position is “kicked out”, leaving the space for the new key. The “nestless” key is then hashed by the second function and is inserted to the second table. The insertion process continues until no key is “nestless”. Searching for a key involves examining two possible slots in two tables. Deletion is performed in the table where the key is stored. Search and delete operations in cuckoo hashing have constant worst case cost. Meanwhile, insertion operations with the cuckoo approach have been also proven to work well in practice. Cuckoo hashing has been shown to be very efficient for small hash tables on modern processors [2].

In cuckoo hashing, the sequence of the evicted keys is usually referred to as “cuckoo path”. It might happen that the process of key evictions is a loop, causing the insertion to fail. If this happens, the table needs to be expanded or rehashed with two new hash functions. The probability of such insertion failure is low when the load factor¹ is lower than 0.49 but increases significantly beyond that [1]. Recent improvements address this issue by either using more hash functions [3] or storing more than one key in a bucket - known as *bucketized cuckoo hashing* [4] [5].

A great effort has been made to build high performance concurrent hash tables running on multi-core systems. Lea’s hash table from Java Concurrency Package [6] is an efficient one. It is a closed address hash table based on chain hashing and uses a small number of locks to synchronize concurrent accesses. Hopscotch hashing [7] is an open address algorithm which combines linear probing with the cuckoo hashing technique. It offers a constant worst case look-up but insertion might require a sequence of relocation similar to the cuckoo hashing. The concurrent hopscotch hashing synchronizes concurrent accesses using locks, one per bucket. A concurrent version of cuckoo hashing found in [8] is a bucketized cuckoo hash table using a stripe of locks for synchronization. As lock-free programming has been proved to achieve high performance and scalability [9] [10], a number of lock-free hash tables have also been introduced in the literature. Micheal, M. [11] presented an efficient lock-free hash table with separated chaining using linked lists. Shalev O. and Shavit N. [12] designed another high performance lock-free closed address resizable hash table. In [13], a lock-free/wait-free hash table is introduced, which does not physically delete an item. Instead, all the live items are moved to a new table when the table is full.

To the best of our knowledge, there has not been any lock-free cuckoo hashing introduced in the literature. There are several reasons which can explain this fact. Because a key can be stored in two possible independent slots in two tables, synchronization of different operations becomes a hurdle to overcome when using lock-freedom. As an example, two insertion operations of a key with different data can simultaneously and independently succeed; this can

¹Load factor: the ratio between the total number of elements currently in the table over its capacity.

cause both of them to co-exist, which is not aligned with the common semantics of hash tables in the literature. In addition, a relocation of a key from one table to another is a combination of one remove and one insert operations, which need to be combined in a lock-free way. While taking care of that, the relocation of a key when it is being looked up can cause the look-up operation to miss the key, though it is just relocated between tables.

In this work, we address these challenges and present a lock-free cuckoo hashing algorithm. We do not consider bucketized cuckoo hashing. To the best of our knowledge, this is the first lock-free cuckoo hashing algorithm in the literature. Our algorithm tolerates any number of process failures. The algorithm offers very high query throughput by optimizing the synchronization between look-up and modification operations. Concurrency among insertions is also high thanks to a carefully designed relocation operation. The sequence of relocations during insertion is broken down into several single relocations to allow higher concurrency among operations. In addition, a fine tuned helping mechanism for relocation operations is designed to guarantee progress. Our evaluation results show that the new cuckoo hashing outperforms the state-of-the-art hopscotch and lock-based chained hash tables [14].

The rest of this paper is organized as follows. Section II introduces our algorithm in a nutshell. The full design together with a pseudo-code description is presented in Section III. Section IV provides the proof of correctness of the algorithm. Experiments and evaluation results are presented in V. Finally, Section VI concludes our paper.

II. LOCK-FREE CUCKOO HASHING ALGORITHM

Our concurrent cuckoo hashing contains two hash tables, hereafter called sub-tables, which correspond to two independent hash functions. Each key can be stored at one of its two possible positions, one in each sub-table. To distinguish the two sub-tables, we refer to one as the primary and to the other as the secondary. The look-up operation always starts searching in the primary sub-table and then in the secondary one. Because there are different use cases of looking-up operations, we divide them into two types. One, *search*, is a query-only one which asks for the existence of a key without modifying the hash table. The other one is a query as a part of another operation such as a deletion or an insertion. We refer to it as *find* to distinguish it from the “real” *search*.

A *search* operation starts by examining the possible slots in the primary sub-table first, and then in the secondary one, and reports if the searched key is found in one of them. Such a simple search, however, can miss an existing key and report the key as not found. The reason is that the reading from two slots is not performed in one atomic step and a relocation operation might interleave in between. The searched key can be relocated from the secondary to the primary sub-table but is missed by the above reading operations. We

call such key a “moving key”. To deal with this issue, we design a two round query protocol enhanced with a logical clock based counter technique. Each hash table slot has a counter attached to it to count the number of relocations at the slot. The first round of the two round query reads from the two possible slots and check for the existence of the searched key like the mentioned simple search does. In addition, it records the slot’s counter values. If the key is not found, the second round does similar readings and examination. The second round can discover the key if it was relocated from the secondary to the primary sub-table, and was missed by the first round query. However, it might also miss the key if it has been relocated back and forth between sub-tables several times and interleaved with the readings. Therefore, the second round also records the counter values and compares them with the values of the first round. If the new values are at least two units higher than the previous ones, there is a possibility that even the two round query misses the key because two or more interleaving relocations have happened. In this case, the search is reexecuted.

The *insert* operation of a key starts by invoking *find* to examine if the key exists. If it does not, the insertion is made to the primary sub-table first and, only if a collision occurs, to the secondary sub-table. If both positions are occupied, a relocation process is triggered to displace the existing key to make space for the new key. The original cuckoo approach [1] inserts the new key to the primary sub-table by evicting a collided key and re-inserting it to the other sub-table, as described in Section I. This approach, however, causes the evicted key to be “nestless”, i.e. absent from both sub-tables, until the re-insertion is completed. This might be an issue in concurrent environments: the “nestless” key is unreachable by other concurrent operations which want to operate on it. One way to deal with this issue is to make the whole relocation process atomic, which is not efficient and scalable since it is going to result in coarse grained synchronization. We approach the relocation process differently. If an insertion requires relocation, an empty slot is created before the new key is actually added to the table. Our approach contains two steps. First, we search for the cuckoo path, to find a vacant slot. Thereafter, the vacant slot is “moved” backwards to the beginning of the cuckoo path by “swapping” with the last key in the cuckoo path, then the second last key and so forth. The new key is then inserted to the empty slot using an atomic primitive. Each “swap” step involves modifications of two slots in two sub-tables. We can design a fine tuned synchronization for the “swap” using single word Compare-And-Swap (CAS)² primitives by using the pointer’s Least-Significant-Bit (LSB) marking technique. This technique, which takes advantages of aligned memory addresses and is widely used in the literature,

²CAS is a synchronization primitive available in most modern processors. It compares the content of a memory word to a given value and, only if they are the same, modifies the content of that word to a given new value.

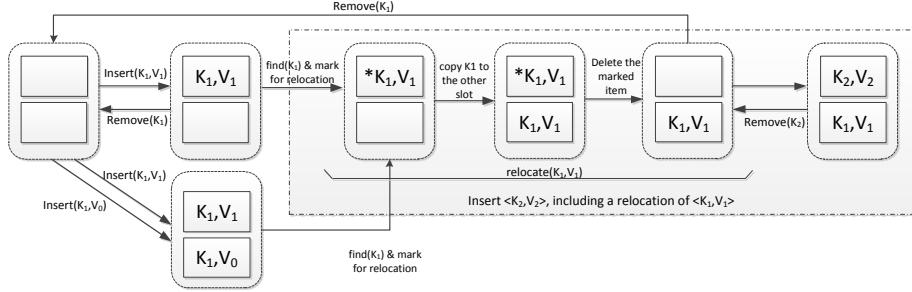


Figure 1: State transition of two possible positions of a key in the primary (upper) and secondary sub-tables

set the unused LSB for certain purposes. In our case, it indicates one thread’s intention to relocate a table’s entry. A helping mechanism is also designed so that other concurrent operations can help finishing an on-going relocation.

Another issue with insertion operations is that a key can be inserted to two sub-tables simultaneously. Since concurrent insertions can operate independently on two sub-tables, they can both succeed. This results in two existing instances of one key, possibly mapped to different values. To prevent such a case to happen, one can use a lock during insertion to lock both slots so that only one instance of a key is successfully added to the table at a time. As we aim for high concurrency and strong progress guarantees, we solve this problem differently. Our table allows, in some special cases, two instances of a key to co-exist in two sub-tables. The special cases are when the two instances have been inserted to two sub-tables simultaneously. (It is noticed that, concurrent insertions to the same sub-table operate and linearize normally, which guarantees that only one instance of a key exists in one sub-table). We design a mechanism to delete one of the instances internally and as soon as possible so that our table still provides the conventional semantic in which one key is mapped to one value. The question is: which instance between the two is to be deleted? Since the two successful insertions which lead to the co-existence must be concurrent, it is always possible to order them so that the insertion to the secondary sub-table is linearized before the insertion to the primary one. In this way, the latter insertion “overwrites” the data inserted by the former one. As a result, if the same key is found in both sub-tables at a certain time, the key in the primary table is the only valid one. Our mechanism to realize such duplication and remove the overwritten key, i.e the one in the secondary sub-table, will be described after discussing below the consequence of such co-existence to the hash table’s operations.

We now analyze the consequence of the co-existence of two instances of a key to the operations of the hash table, beginning with the look-up operations. The query-only *search* first examines the primary hash table, and can report if the searched key is stored there immediately. In the cases that two instances of a key co-exist in two sub-tables, *search* always returns the one in the primary sub-table, which agrees

with the semantics described in the previous paragraph. The *find* operation, on the other hand, is used at the beginning of any *insert* or *remove* operation. The result of the invoked *find*, i.e the key exists in one or both sub-tables, affects the way the invoker behaves. Therefore, if *find* discovers that two instances of a key exist, it deletes the one in the secondary sub-table, as being described in detail in the next section. When an *insert* or a *remove* proceeds after *find* returns, there is going to be one instance of the key in the table.

The remove operation of a key starts by invoking *find* to locate the table’s slot where the key is being stored. Then the key is removed by means of a CAS primitive.

Figure 1 shows the states of two possible positions on two sub-tables where a key K_1 can be hashed to. The states change according to the operations performed. The operations are, for example, an *insert(K_1, V_1)*, a *remove(K_1, V_1)*, two concurrent *insert(K_1, V_0)* and *insert(K_1, V_1)*, and an *insert(K_2, V_2)* (in the dashed rectangle) which requires a relocation of $\langle K_1, V_1 \rangle$. When *find()* is invoked, it can also delete the duplicated key, i.e $\langle K_1, V_0 \rangle$ stored in the second sub-table.

III. DETAILED ALGORITHMIC DESCRIPTION

We are now presenting the detailed description and pseudo-code for the functions of our cuckoo hash table. The pseudo-code follows the C/C++ language conventions.

A. Search operation

Searching for a key in our lock-free cuckoo hash table includes querying for the existence of the key in two sub-tables, as in Program 2. A key is available if it is found in one of them. A search operation starts with the first query round by reading from the possible slots in the primary sub-table $\text{table}[0]$ (lines 24-26), and then in the secondary sub-table $\text{table}[1]$ (lines 27-29). If the key is found in one of them, the value mapped to key is returned.

As mentioned in section II, the above query can miss the searched key which happens to be a “moving key”. The key present in $\text{table}[1]$ is relocated to $\text{table}[0]$, meanwhile *search* reads from $\text{table}[0]$ and then $\text{table}[1]$. To avoid such missing, *search* performs the second round query (lines 32-34).

Program 1 Data structure and support functions

```
1 HashEntry:
2   word key,
3   word value;
4
5 CountPtr:
6   <HashEntry*, int> <entry, counter>
7
8 int hash1(key)
9 int hash2(key)
10 bool checkCounter(int ts1, int ts2, int ts1x, int ts2x)
11 /*check the counter values to see if 2 relocations may
12 have taken place*/
13 class CuckooHashTable
14   EntryType *table[2][] // 2 sub-tables
15   word find(word key, CountPtr &<e1,ts1>, CountPtr &<e2,
16             ts2>)
17   word search(word key)
18   insert(word key, word value)
19   remove(word key)
20   relocate(int which, int index)
21   help_relocate(int which, int index, bool initiator)
22   del_dup(idx1, <e1,ts1>, idx2, <e2,ts2>)
```

Program 2 word search (word key)

```
22 //h1=hash1(key) and h2=hash2(key)
23 while (true)
24   <e1,ts1> ← table[0][h1] //read the element \& counter
25   if (e1≠NULL ∧ e1→key = key)
26     return e1→value
27   <e2,ts2> ← table[1][h2]
28   if (e2≠NULL ∧ e2→key = key)
29     return e2→value
30
31 //second round query
32 <e1x,ts1x> ← table[0][h1]
33 ...
34 <e2x,ts2x> ← table[1][h2]
35 ...
36
37 if (checkCounter(ts1, ts2, ts1x, ts2x))
38   continue
39 else
40   return NIL
```

This two-round query, however, still can miss an existing key if the key is relocated back and forth between $\text{table}[1]$ and $\text{table}[0]$ repetitively and alternatively when search reads from each sub-table. The possibility of such continuously relocation of a key is very rare but can not be ruled out. To deal with that, we employ a technique based on Lamport's logical clocks [15]. The idea of this technique is to attach a counter to each slot of the hash table to record the number of relocations happening at that slot. Similar to a logical clock whose value changes when a local event happens or when a message is received, the value of the counter is changed on the event of relocations. The counter is initialized to 0. When an element stored in a slot is relocated, the slot's counter is incremented. When a slot serves as the destination of a relocation, its counter is updated with the maximum of its current counter value and the source's counter value, plus 1. The counter value of a slot remains even when the element stored in that slot is deleted or relocated to the

Program 3 word find(word key, CountPtr& <e1,ts1>, CountPtr& <e2,ts2>)

```
41 //h1=hash1(key) and h2=hash2(key)
42 word result; int counter
43
44 while (true)
45   <e1,ts1> ← table[0][h1]
46   if (e1 ≠ NULL)
47     if (e1 is marked)
48       help_relocate(0, h1, false)
49       continue
50     if (e1→key = key)
51       result ← FIRST
52
53 <e2,ts2> ← table[1][h2]
54 if (e2 ≠ NULL)
55   if (e2 is marked)
56     help_relocate(1, h2, false)
57     continue;
58   if (e2→key = key)
59     if (result = FIRST)
60       del_dup(h1, <e1, ts1>, h2, <e2, ts2>)
61     else
62       result ← SECOND
63
64 if (result=FIRST ∨ result=SECOND)
65   return result
66 /*second round query*/
67 <e1,ts1x> ← table[0][h1]
68 ...
69 <e2,ts2x> ← table[1][h2]
70 ...
71 if (checkCounter(ts1, ts2, ts1x, ts2x))
72   continue
73 else
74   return NIL
```

other sub-table. For example, consider a key associated with counter value t is stored at $\text{table}[1][\text{h}2]$. When key is relocated to $\text{table}[0][\text{h}1]$ which has counter t_1 , the new counter value of $\text{table}[0][\text{h}1]$ is $\max(t, t_1) + 1$ and that of $\text{table}[1][\text{h}2]$ is incremented to $t + 1$.

By examining the above counters after the second round query, a search can detect if it might have missed an existing key. Such missing happens if: (i) Before the execution of line 24, the key is stored in the secondary table at $\text{table}[1][\text{h}2]$, then (ii) the key is relocated from $\text{table}[1]$ to $\text{table}[0]$ before the second read at line 27, then (iii) relocated back to $\text{table}[1]$ before the next read at line 32, and finally (iv) relocated again back to $\text{table}[0]$ before line 34. If it is so, the counter value read at line 32 should be at least two units higher than the one read at line 24. Similar condition is applied for the counter values read at line 34 and line 27. In addition, the counter value read at line 34 is at least 3 units higher than the one read at line 24 because the counter increases its value like a logical clock when a relocation happens. If these conditions are satisfied, i.e checkCounter returns true , the two-round query probably misses an item because of alternative relocations, so the search restarts.

In practice, as each slot in the hash table is a pointer to a table element, we can use the unused bits of pointer values on x86_64 to store the counter value. Currently pointers on x86_64 use only 48 lower bits of the available 64 bits. We can use the 16 highest bits of the 64-bit pointer to store the

Program 4 *insert*(word key, word value)

```
75 //h1=hash1(key); h2=hash2(key)
76 HashEntry *newNode(key,value)
77 CountPtr *<ent1,ts1>, *<ent2,ts2>
78 start_insert:
79     int result ← find(key, <ent1,ts1>, <ent2,ts2>)
80     if (result=FIRST V result=SECOND)
81         Update the current entry with new value
82         return
83
84     if (ent1=NULL)
85         if (¬CAS(&table[0][h1],<ent1,ts1>,<newNode,ts1>))
86             goto start_insert
87         return
88     if (ent2=NULL)
89         if (¬CAS(&table[1][h2],<ent2,ts2>,<newNode,ts2>))
90             goto start_insert
91         return
92
93     result ← relocate(0, h1)
94     if (result=true) goto start_insert
95     else
96         //rehash()
```

counter value, an approach which has been used in literature [16]. This approach is efficient as the pointer to an element and its counter can be loaded in one read operation. The disadvantage of this technique is that the counter which has been increased by $2^{16} + k$ can be misinterpreted to be increased by just k , where k is any counter value. However, such increment of $2^{16} + k$ can only be made by many thousands of relocation operations happening at the same slot. Moreover, it must have happened in a very short period of time of a search operation to cause such misinterpretation. With a good choice of hash functions, the possibility that such misinterpretation happens is practically impossible. Therefore, 16 bits are sufficient to store the counter value.

B. Find operation

Program 3 shows the pseudo code of the *find* operation, which functions similar to the *search*. The *find* takes an argument key and answers if, and in which sub-table, the key exists. In addition, it also reports the current values (and their associated counter values) stored at the two possible positions of key. The logic flow of the *find* is similar to that of the *search*, in the sense that it also uses a two round query. However, it has 3 main differences compared to the *search*. First, if it reads an entry who LSB is marked, indicating an on-going relocation operation, it helps the operation (lines 47-48, and 55-56). Secondly, it examines both sub-tables instead of returning immediately when key is found. This is to discover if two instances of the key exist in two sub-tables. When the same key is found on both sub-tables, the one in the secondary sub-table table[1] is deleted (line 60), as described in Section II. Finally, *find* returns also current items which are stored at two possible slots where key should be hashed to. This information is used by the invokers, i.e *insert* or *delete* operations, as described in next subsections.

Program 5 *remove*(word key)

```
97 //h1=hash1(key); h2=hash2(key)
98 CountPtr *<ent1,ts1>, *<ent2,ts2>
99 while (true)
100     ret ← find(key, <ent1,ts1>, <ent2,ts2>)
101     if (ret = NULL) return
102     if (ret = FIRST)
103         if (CAS(&table[0][h1],<ent1,ts1>,<NULL,ts1>))
104             return
105     else if (ret = SECOND)
106         if (table[0][h1]≠<ent1,ts1>)
107             continue
108         if (CAS(&table[1][h2],<ent2,ts2>,<NULL,ts2>))
109             return
```

C. Insert operation

The insertion of a key, Program 4, works as follows. First, it invokes the *find* at line 79 to examine the state of the key: if it exists in the sub-tables and what are the current entries stored at the slots where the key can be hashed to. If the key already exists, the current value associated with it is updated with the new value and the *insert* returns (line 82). Otherwise, the *insert* operation proceeds to store the new key. If one of the two slots is empty (lines 84 and 88), the new entry is inserted with a CAS. If both slots are occupied by other keys, relocation process is triggered at line 93 to create an empty slot for the new key. The relocation operation is described in detail in Section III-E. If the relocation succeeds to create an empty slot for the new key, the insertion retries. Otherwise, which means the length of the relocation chain exceeds the THRESHOLD, the insertion fails. In this case, typical approaches in the literature of cuckoo hashing such as a rehash with two new hash functions or an extension of the size of the table can be used.

D. Remove operation

The *remove* operation also starts by invoking *find* at line 100. If the key is found, it is removed by a CAS, either at line 103 or 108.

E. Relocation operation

When both slots which can accommodate a newly inserted key are occupied by existing keys, one of them is relocated to make space for the new key. This can trigger a sequence of relocations as the other slot might be occupied too. The *relocate* method presented in Program 6 performs such a relocation process. As mentioned earlier, we use a relocation strategy which can retain the presence of a relocated key in the table without the need for expensive atomicity of the whole relocation process. First, the cuckoo path is discovered, lines 113-135. Then, the empty slot is moved backwards to the beginning of the path, where the new key is to be inserted, lines 137-154.

The path discovery starts from a slot index of one of the sub-tables identified by *which* and runs at most THRESHOLD steps along the path. If table[*which*][*index*] is an empty slot, the discovery finishes (line 134). Otherwise, i.e the slot is

Program 6 *int relocate(int which, int index)*

```

110 int route[THRESHOLD] //storing cuckoo path
111 int start_level=0, tbl=which, idx=index

113 path_discovery:
114     bool found ← false
115     int depth ← start_level
116     do {
117         <e1,ts1> ← table[tbl][idx];
118         while (e1 is marked)
119             help_relocate(tbl, idx, false)
120         <e1,> ← table[tbl][idx]
121         if (<pre,tsp>=<e1,ts1> ∨ pre→key=e1→key)
122             if (tbl = 0)
123                 del_dup(idx,<e1,ts1>,pre_idx,<pre,tsp>)
124             else
125                 del_dup(pre_idx,<pre,tsp>,idx,<e1,ts1>)
126             if (e1 ≠ NULL)
127                 route[depth] = idx
128             key ← e1→key;
129             <pre,tsp> ← <e1,ts1>
130             pre_idx ← idx
131             tbl ← 1 - tbl
132             idx ← tbl = 0 ? hash1(key) : hash2(key)
133         else
134             found ← true
135     } while (!found ∧ ++depth<THRESHOLD)

137 if (found)
138     tbl ← 1 - tbl;
139     for (i ← depth-1; i>=0; --i, tbl ← 1-tbl)
140         idx ← route[i].index
141         <e1,ts1> ← table[tbl][idx]
142         if (e1 is marked)
143             help_relocate(tbl, idx, false)
144             <e1,ts1> ← table[tbl][idx]
145         if (e1 = NULL)
146             continue
147         dest_idx ← tbl=0?hash2(e1→key):hash1(e1→key)
148         <e2,ts2> ← table[1-tbl][dest_idx]
149         if (e2 ≠ NULL)
150             start_level ← i+1
151             idx ← dest_idx
152             tbl ← 1 - tbl
153             goto path_discovery
154         help_relocate(tbl, idx, false)
155     return found

```

occupied by a key (line 126), the key should be relocated to its other slot in the other sub-table. The discovery then continues with the other slot of key. If this slot is empty, the discovery finishes. Otherwise, the discovery continues similarly as before. Each element along the path is identified by a sub-table and an index on that sub-table. Along the path, the sub-tables that elements belong to alternatively change between the primary and secondary ones. Therefore, the data of the path which need to be stored are the indexes of the elements along the path and the sub-table of the last element.

Once the cuckoo path is found, the empty slot is moved backwards along the path by a sequence of “swaps” with the respective preceding slot in the path. Each swap is actually a relocation of the key in the latter slot, a.k.a the source, to the empty slot, a.k.a the destination. Because of the concurrency, the entry stored in the source might have changed. Thus, the relocation operation needs to update the destination and check for its emptiness (lines 148-149), and retry the path discovery if the destination is no longer empty (line 153). If the destination is empty, the relocation is performed in three

Program 7 Help relocation and delete duplication operations

```

156 void help_relocate(int which, int index, bool initiator)
157     while (true)
158         <src,ts1> ← table[which][index]
159         while (initiator && src is not marked)
160             if (src = NULL) return
161             CAS(&table[which][index]),<src,ts1>,<src|1,ts1>
162             <src,ts1> ← table[which][index]
163         if (src is not marked) return
164         /*hd=hash(src.key) where hash is hash function used
           for table (1-which)*/
165         <dst,ts2> ← table[1-which][hd])
166         if (dst = NULL)
167             nCnt ← ts1>ts2 ? ts1 + 1 : ts2 + 1
168             if (<src,ts1> ≠ table[which][index])
169                 continue
170             if (CAS(&table[1-which][hd],<dst,ts2>,<src,nCnt>))
171                 CAS(&table[which][index],<src,ts1>,<NULL,ts1+1>)
172             return
173         //dst is not null
174         if (src = dst)
175             CAS(&table[which][index]),<src,ts1>,<NULL,ts1+1>
176             return
177         CAS(&table[which][index],<src,ts1>,<src&~1,ts1+1>)
178     return false;

180 void del_dup(idx1, <e1,ts1>, idx2 , <e2,ts2>)
181     if (<e1,ts1>≠table[0][idx1] ∧
182         <e2,ts2>≠table[1][idx2])
183     return
184     if (e1→key ≠ e2→key)
185     return
186     CAS(&table[1][idx2],<e2,ts2>,<NULL,ts2>)

```

steps in the *help_relocate* operation presented in Program 7. First, the source entry’s LSB is marked to indicate the relocation intention (line 161). Then, the entry is copied to the destination slot (line 170), which has been made empty. Finally, the source is deleted (line 171). Marking the LSB allows other concurrent threads to help the on-going relocation, for example at lines 119 and 143.

After a slot is marked in *help_relocation*, the destination of the relocation might have been changed and is no longer empty. This can be because either other threads successfully help relocating the marked entry, or a concurrent insertion has inserted a new key to that destination slot. If it is the former case (line 174), the source of the relocation is deleted either by that helping thread (line 171) or by the current thread (line 175). If it is the latter case, the *help_relocation* fails, unmarks the source (line 177) and returns. The relocation process then continues but might need to retry the path discovery in the next loop.

IV. PROOF OF CORRECTNESS

In this section, we are going to prove that our cuckoo hash table is linearizable and lock-free. At first, we prove the linearizability under the assumption that a key exists in only one sub-table. Later on, we prove that if there are two instances of a key in two sub-tables, the linearizability is not violated. Then we continue to prove the lock-freedom. We provides proofs related to the *search* operation and the full proofs can be found in the technical report [17]. In the followings, we assume that a key can be stored in two possible positions: *table[0][h1]* and *table[1][h2]*.

Lemma 1: When *help_relocation*(which, index) is invoked, either it succeeds relocating the element pointed to by $\text{table}[\text{which}][\text{index}]$ to the other table, i.e $\text{table}(1-\text{which})$ and unmarks the source slot; or if it fails doing that, the source slot, i.e $\text{table}[\text{which}][\text{index}]$ is unmarked.

Corollary 1: If *help_relocation* succeeds relocating entry in $\text{table}[\text{which}][\text{index}]$ to the other sub-table, the counter values of both source and destination increases by at least 1.

Following Corollary 1, it is easy to see that if there were two relocations which had happened at one slot, the slot's counter would have been increased by at least 2 units.

Lemma 2: The *search* is linearizable.

Proof: The *search* operation returns a value only when one of the keys in e_1, e_2, e_{1x}, e_{2x} (lines 24, 27, 32 or 34) matches the searched key. In this case, *search* is linearized at the respective line. We now consider the loop of the *search* to see when it returns NIL.

As we discussed in Section III-A, during a search for key, the two round query protocol misses the searched key only when there is a sequence of relocations of key from $\text{table}[1]$ to $\text{table}[0]$, back to $\text{table}[1]$ and again to $\text{table}[0]$, as described in subsection III-A. Condition at line 40 can detect if such a scenario may happen by examining the counter values of $\text{table}[0][h_1]$ and $\text{table}[1][h_2]$, as described also in subsection III-A. If the condition is satisfied, the *search* restarts. We note that the condition is also satisfied if there are two independent relocations of other keys which are stored in the same slots; or if there is only one relocation at each slot but the relocation increases the counter by two or more units. In these cases, the *search* might restart unnecessarily but its correctness is not violated.

Therefore, the *search* returns NIL when the key is not found in any of the read entries: e_1, e_2, e_{1x}, e_{2x} and there is no possibility that the key is relocated which forces the *search* to reexecute the loop. Even though, there are cases that key appears in one of the sub-tables at the time *search* performs a reading from the other sub-table. In such cases, *search* might still return NIL, but we can argue that it is totally correct. We consider, as an example, a key exist in one sub-table as *search* performs reading for the second round query at lines 32 and 34, and show the correct linearization points. Other readings, e.g lines 24 and 27, can be argued in a similar manner. If the key exists in the table when *search* executes lines 32 and 34, and the *search* returns NIL:

- key must be inserted to $\text{table}[0]$ after the reading from that sub-table at line 32. The *search* can be linearized at line 32 where key has not been inserted to the table.
- Or key exists in $\text{table}[1]$ before the *search* starts and is deleted before the *search* reads from it (line 34). The *search* can then be linearized to line 34, when key has been deleted.
- Or key exists in $\text{table}[1]$ when the *search* reads from $\text{table}[0]$ (line 32), is deleted and then re-inserted to $\text{table}[0]$ before the *search* reads from $\text{table}[1]$ (line 34).

In such scenario, neither line 32 or line 34 can be the correct linearization point of the *search*. Because key exists in the table at those points of time, in particular, in the other sub-table than the one *search* reads from. Even though, we notice that there is an interval between when key is deleted from $\text{table}[1]$ and when it is inserted to $\text{table}[0]$ (if these operations overlap, the re-insertion would have failed), this period of time is inside the duration that *search* executes line 32 to line 34. In this interval, key does not exist anywhere in the table. Therefore, we can always linearize *search* to a point of time in that interval. This satisfies the requirement that the linearization point must be between the time when *search* is invoked and when it responds. ■

Henceforth, *search* is linearizable. ■

Lemma 3: The *find* operation is linearizable.

Lemma 4: The *remove* operation is linearizable.

Lemma 5: The *insert* operation is linearizable.

Now we consider the scenario where two instances of a key co-exist in the table. When two concurrent insertions try to insert the same key to two sub-tables, they might both succeed and store it in two possible positions of that key. As described in Section II, our hash table allows such physical co-existence and then removes the instance in the second sub-table before any mutating operations can operate on these two positions. The subsequence operations can only see one valid instance, i.e the one in the primary table.

Lemma 6: The correctness of the *search* operation is immune of the concurrent physical existence of key in both sub-tables.

The below propositions can be derived from the pseudo code of *find* and *relocate*.

Proposition 1: If two instances of key co-exist in the table when a *find* on key is invoked, it removes the instance of the key in the second sub-table.

Proposition 2: If two instances of key co-exist in the table when a *relocate* of key happens, it removes the instance of the key in the second sub-table.

Now, we examine the effect of the existence of a duplicated key to the correctness of *insert/remove* operations.

Lemma 7: The correctness of a *remove* and *insert* operation of a key in Lemma 4 and 5 holds even when two instances of a key exist concurrently.

Theorem 1: The hash table algorithm is linearizable.

Proof: Each operation of the hash table is linearizable follows Lemmas 2, 4, 5, and Lemmas 6, 7. ■

Now, we are ready to prove that the algorithm is lock-free.

Lemma 8: Either the *help_relocation* operation finishes after a finite number of steps or another operation must have finished in a finite number of steps.

Lemma 9: If a *help_relocation* operation finishes but fails to relocate $\text{table}[\text{which}][\text{index}]$, there must be another operation making progress during that execution of the *help_relocation*.

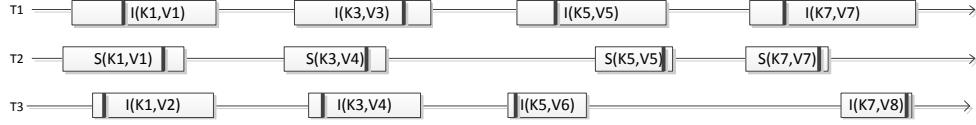


Figure 2: Concurrent inserts can create the existence of two instances of a key in the table

We observe that *help_relocation* can encounter an ABA problem³ [8], even when we have a proper memory management to handle the hash table's element. This scenario can take place when there are threads executing line 166 to relocate the same `table[which][index]`. Meanwhile, a new key can be inserted to `dst`, and then deleted from `dst`. Some of the threads doing relocation can observe that `dst` is pointing to the inserted element, which leads to the CAS at line 170 to fail and the source of the relocation to be unmarked at line 177. Meanwhile, other threads doing relocation might perform the CAS at line 170 after the deletion, and therefore succeed to copy `table[which][index]` to `dst`. As a result, the key exists in two sub-tables. However, such co-existence caused by the ABA does not hurt the correctness of our algorithm. This is because our algorithm is capable of tolerating such co-existence and can soon remove the one in the second sub-table. Such removal is done by the calling thread performing this *help_relocation*, or any other thread doing *find* or *relocate* which involves the slots storing the duplicated key (at line 60, 123 or 125, as discussed in Section II).

Lemma 10: The *search* operation finishes after a finite number of steps, or another operation must have finished after a finite number of steps.

Proof: The *search* operation has only one `while` loop which, according to the proof of Lemma 2, repeats only when there are relocations of keys stored in `table[0][h1]` and `table[1][h2]`. Such relocations mean progress of the respective operations performing them. This observation holds even when the relocations move key(s) back and forth between two sub-tables. In this case, the *search* might not make progress but the relocation progresses towards the THRESHOLD number of relocation steps. When it reaches the THRESHOLD and returns false, the insertion calling such a relocation fails and proceeds with rehashing or resizing the hash table. ■

Lemma 11: A *find* operation finishes after a finite number of steps, or another operation must have finished after a finite number of steps.

Lemma 12: An *insert* operation finishes after a finite number of steps or another operation must have finished after a finite number of steps.

Lemma 13: A *remove* operation finishes after a finite number of steps or another operation must have finished after a finite number of steps.

Theorem 2: The hash table algorithm is lock-free.

³ABA problem happens when an operation succeeds because the memory location it read has not changed; but in fact, it has changed its value from A to B and then back to A.

Proof: According to Lemmas 10, 12, 13, our cuckoo hash table always makes progress after a finite number of steps. ■

V. EXPERIMENTAL EVALUATION

This section evaluates the performance of our lock-free cuckoo hash table and compares it with current efficient hash tables. We use micro-benchmarks with several concurrent threads performing hash table's operations, a standard evaluation approach taken in the literature.

A. The Experimental Setup

We compare our lock-free cuckoo hashing with:

- A lock-based chained one: that uses a linked-list to store keys hashed to the same bucket. A number of locks, equal to the number of table segments, are used [14].
- Hopscotch hashing: a concurrent version of hopscotch hashing, with each lock for a segment [7]. Thanks to the kindness of the authors, we could obtain the original source code of hopscotch hashing.
- LF Cuckoo: our new lock-free cuckoo hashing.

All the algorithms were implemented in C++ and compiled with the same flags. No customized memory management was used. In all the algorithms, each bucket contains either two pointers to a key and a value, or an entry to a hash element, which contains a key and a value.

The experiments were performed on a platform of two 8-core Xeon E5-2650 at 2GHz with HyperThreading, 64GB DDR3 RAM. In our evaluation, we sampled each test point 5 times and plotted the average. To make sure the tables did not fit into the cache, we used a table-size of 2^{23} slots. Each test used the same set of keys for all the hash tables.

B. Results

Figure 3 presents the throughput result of the hash tables in different distributions of actions. The commonly found distribution is $90s/5i/5r$, i.e 90% *search*, 5% *insert* and 5% *remove*. Other less common distributions were also evaluated. One with more query-only operations: $94s/3i/3r$. Two others with more mutating operations: $80s/10i/10r$ and $60s/20i/20r$. As it is commonly known that original cuckoo hashing works with load factors lower than 49%, we used the load-factor of 40%. The concurrency increased up to 32 threads, the maximum number of concurrent hardware threads supported by the machines.

Our lock-free cuckoo hashing performs consistently better than both the lock-based chained and the hopscotch hashing

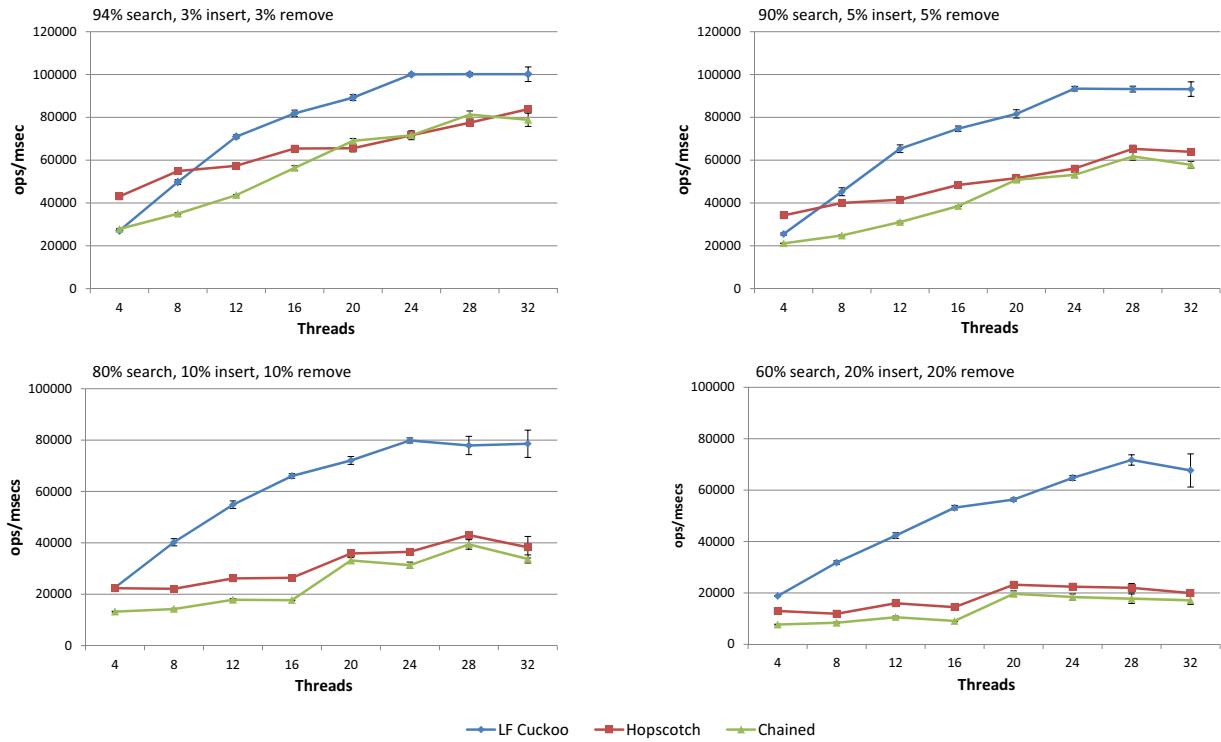


Figure 3: Throughput as a function of concurrency at load factor 40%

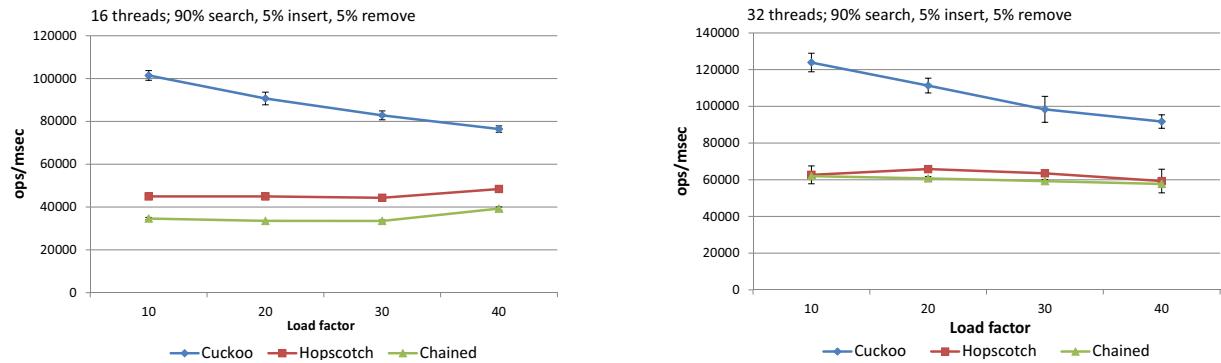


Figure 4: Throughput as a function load factor at 16 and 32 threads

in all the access distribution patterns. This is because positive searches need to examine either one or two table’s slots, and negative searches need 3 read operations in most cases. Cases that *search* might need to perform more read operations do happen but not often. This is because the possibility that a relocation of a key happens concurrently as the key is being queried is not high. In addition, our algorithm is designed so that the search operations still make progress concurrently with any other mutating operations. In contrast, the lock-based chained and the hopscotch hashing lock the bucket during insertion or removal.

Our lock-free cuckoo hashing maintains very high throughput in scenarios with more mutating operations, i.e $10i/10r$ or $20i/20r$, respectively. The insert operation in cuckoo hashing might require relocations of existing keys

to make space for the new key. The algorithm, however, has been fine designed to allow high concurrency between relocation operations and other operations. Meanwhile, the lock-based chained and the hopscotch hashing degrade quickly when the percentage of mutating operations increases, mainly because of their blocking designs.

Figure 4 presents throughput results as a function of load factor. In cuckoo hashing, higher load factor means more relocations of existing keys during insertion. Therefore, we can observe that the throughput of our lock-free cuckoo hashing decreases when the load factor increases. Nevertheless, our cuckoo hashing algorithm always achieve throughput 1.5–2 times as much as other algorithms, in both cases of 16 and 32 concurrent threads.

We now analyze the cache behavior of our lock-free

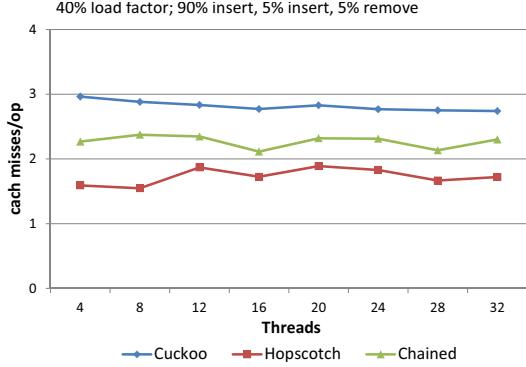


Figure 5: The number of cache-misses per operation.

cuckoo hashing. A positive search operation usually requires reading 1 or 2 references and a negative one often requires reading 3 references in the two-round query protocol, if no concurrent relocation happens at the read slots. Otherwise, search operations might need to perform more read operations, which might cause more cache misses. A removal of a key often needs one additional CAS compared to a search operation to delete the found element from the tables. Insertion operations are more complicated. If an insertion does not trigger any relocation, its behavior is similar to the deletion operation. Otherwise, it can cause more cache misses. We have recorded that the number of relocations is approximately 2% of the total performed operations, in the distribution of $90s/5i/5r$. Therefore, we expect a higher number of cache misses in our cuckoo hashing compared to other hash tables. A measurement of number of cache misses of our lock-free cuckoo hashing is presented in Figure 5. Our lock-free cuckoo table triggers about 3 cache misses per operation, a bit higher than hopscotch hashing and lock-based chained hashing. Regardless of a slightly higher number of cache misses, our lock-free cuckoo hashing has maintained a good performance over the other algorithms, thanks to the fine designed mechanism to handle concurrency.

VI. CONCLUSION

We have presented a lock-free cuckoo hashing algorithm which, to the best of our knowledge, is the first lock-free cuckoo hashing in the literature. Our algorithm uses atomic primitives which are widely available in modern computer systems. We have performed experiments that compares our algorithm with the efficient parallel hashing algorithms from the literature, in particular hopscotch hashing and optimized lock-based chained hashing. The experiments show that our implementation is highly scalable and outperform the other algorithms in all the access pattern scenarios.

ACKNOWLEDGMENT

The authors would like to acknowledge the support of the European Union Seventh Framework Programme (FP7/2007-2013) through the EXCESS Project (www.excess-project.eu) under grant agreement 611183.

REFERENCES

- [1] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122 – 144, 2004.
- [2] M. Zukowski, S. Héman, and P. Boncz, "Architecture-conscious hashing," in *Proceedings of the 2nd International Workshop on Data Management on New Hardware*. ACM, 2006.
- [3] D. Fotakis, R. Pagh, P. Sanders, and P. Spirakis, "Space efficient hash tables with worst case constant access time," in *Proceedings of the 20th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, ser. LNCS, vol. 2607. Springer Berlin Heidelberg, 2003, pp. 271–282.
- [4] K. Ross, "Efficient hash probes on modern processors," in *Proceedings of the IEEE 23rd International Conference on Data Engineering (ICDE)*, 2007, pp. 1297–1301.
- [5] A. Kirsch, M. Mitzenmacher, and U. Wieder, "More robust hashing: Cuckoo hashing with a stash," *SIAM J. Comput.*, vol. 39, no. 4, pp. 1543–1561, Dec. 2009.
- [6] D. Lea, "Hash table util.concurrent.concurrenthashmap," <http://gee.cs.oswego.edu/cgi-bin/viewcvs.cgi/jsr166/src/main/java/util/concurrent/>, 2003.
- [7] M. Herlihy, N. Shavit, and M. Tzafrir, "Hopscotch hashing," in *The Proceedings of the 22nd International Symposium on Distributed Computing (DISC)*, ser. LNCS, vol. 5218. Springer Berlin Heidelberg, 2008, pp. 350–364.
- [8] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.
- [9] P. Tsigas and Y. Zhang, "Evaluating the performance of non-blocking synchronization on shared-memory multiprocessors," *SIGMETRICS Perform. Eval. Rev.*, vol. 29, no. 1, pp. 320–321, Jun. 2001.
- [10] D. Cederman, A. Gidenstam, P. H. Ha, H. Sundell, M. Papatriantafilou, and P. Tsigas, "Lock-free concurrent data structures," in *Programming Multi-Core and Many-Core Computing Systems*, S. Pllana and F. Xhafa, Eds. Wiley-Blackwell, 2014.
- [11] M. M. Michael, "High performance dynamic lock-free hash tables and list-based sets," in *Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*. ACM, 2002, pp. 73–82.
- [12] O. Shalev and N. Shavit, "Split-ordered lists: Lock-free extensible hash tables," *Journal of the ACM*, vol. 53, no. 3, pp. 379–405, May 2006.
- [13] C. Click, "A lock-free wait-free hash table," http://www.stanford.edu/class/ee380/Abstracts/070221_LockFreeHash.pdf, accessed: 2013-11-14.
- [14] D. E. Knuth, *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1997.
- [15] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.
- [16] M. Brunink, M. Susskraut, and C. Fetzer, "Boundless memory allocations for memory safety and high availability," in *Proceedings of the 41st International Conference on Dependable Systems Networks (DSN)*, 2011, pp. 13–24.
- [17] N. Nguyen and P. Tsigas, "Lock-free cuckoo hashing," Chalmers University of Technology, Department of Computer Science and Engineering, Tech. Rep. 2014:03, January 2014.

Mathematizing C++ Concurrency

Mark Batty Scott Owens Susmit Sarkar Peter Sewell Tjark Weber
University of Cambridge

Abstract

Shared-memory concurrency in C and C++ is pervasive in systems programming, but has long been poorly defined. This motivated an ongoing shared effort by the standards committees to specify concurrent behaviour in the next versions of both languages. They aim to provide strong guarantees for race-free programs, together with new (but subtle) relaxed-memory atomic primitives for high-performance concurrent code. However, the current draft standards, while the result of careful deliberation, are not yet clear and rigorous definitions, and harbour substantial problems in their details.

In this paper we establish a mathematical (yet readable) semantics for C++ concurrency. We aim to capture the intent of the current ('Final Committee') Draft as closely as possible, but discuss changes that fix many of its problems. We prove that a proposed x86 implementation of the concurrency primitives is correct with respect to the x86-TSO model, and describe our CPPMEM tool for exploring the semantics of examples, using code generated from our Isabelle/HOL definitions.

Having already motivated changes to the draft standard, this work will aid discussion of any further changes, provide a correctness condition for compilers, and give a much-needed basis for analysis and verification of concurrent C and C++ programs.

Categories and Subject Descriptors C.1.2 [*Multiple Data Stream Architectures (Multiprocessors)*]: Parallel processors; D.1.3 [*Concurrent Programming*]: Parallel programming; F.3.1 [*Specifying and Verifying and Reasoning about Programs*]

General Terms Documentation, Languages, Reliability, Standardization, Theory, Verification

Keywords Relaxed Memory Models, Semantics

1. Introduction

Context Systems programming, of OS kernels, language runtimes, etc., commonly rests on shared-memory concurrency in C or C++. These languages are defined by informal-prose standards, but those standards have historically not covered the behaviour of concurrent programs, motivating an ongoing effort to specify concurrent behaviour in a forthcoming revision of C++ (unofficially, C++0x) [AB10, BA08, Bec10]. The next C standard (unofficially, C1X) is expected to follow suit [C1X].

The key issue here is the multiprocessor relaxed-memory behaviour induced by hardware and compiler optimisations. The design of such a language involves a tension between usability and performance: choosing a very strong memory model, such as se-

quential consistency (SC) [Lam79], simplifies reasoning about programs but at the cost of invalidating many compiler optimisations, and of requiring expensive hardware synchronisation instructions (e.g. fences). The C++0x design resolves this by providing a relatively strong guarantee for typical application code together with various *atomic* primitives, with weaker semantics, for high-performance concurrent algorithms. Application code that does not use atomics and which is race-free (with shared state properly protected by locks) can rely on sequentially consistent behaviour; in an intermediate regime where one needs concurrent accesses but performance is not critical one can use *SC atomics*; and where performance is critical there are *low-level atomics*. It is expected that only a small fraction of code (and of programmers) will use the latter, but that code —concurrent data structures, OS kernel code, language runtimes, GC algorithms, etc.— may have a large effect on system performance. Low-level atomics provide a common abstraction above widely varying underlying hardware: x86 and Sparc provide relatively strong TSO memory [SSO⁺10, Spa]; Power and ARM provide a weak model with cumulative barriers [Pow09, ARM08, AMSS10]; and Itanium provides a weak model with release/acquire primitives [Int02]. Low-level atomics should be efficiently implementable above all of these, and prototype implementations have been proposed, e.g. [Ter08].

The current draft standard covers all of C++ and is rather large (1357 pages), but the concurrency specification is mostly contained within three chapters [Bec10, Chs.1, 29, 30]. As is usual for industrial specifications, it is a prose document. Mathematical specifications of relaxed memory models are usually either operational (in terms of an abstract machine or operational semantics, typically involving explicit buffers etc.) or axiomatic, defining constraints on the relationships between the memory accesses in a complete candidate execution, e.g. with a happens-before relation over them. The draft concurrency standard is in the style of a prose description of an axiomatic model: it introduces various relationships, identifying when one thread *synchronizes* with another, what a *visible side effect* is, and so on (we explain these in §2), and uses them to define a happens-before relation. It is obviously the result of extensive and careful deliberation. However, when one looks more closely, it is still rather far from a clear and rigorous definition: there are points where the text is unclear, places where it does not capture the intent of its authors, points where a literal reading of the text gives a broken semantics, several substantial omissions, and some open questions. Moreover, the draft is very subtle. For example, driven by the complexities of the intended hardware targets, the happens-before relation it defines is intentionally non-transitive. The bottom line is that, given just the draft standard text, the basic question for a language definition, of what behaviour is allowed for a specific program, can be a matter for debate.

Given previous experience with language and hardware memory models, e.g. for the Java Memory Model [Pug00, MPA05, CKS07, SA08, TVD10] and for x86 multiprocessors [SSZN⁺09, OSS09, SSO⁺10], this should be no surprise. Prose language definitions leave much to be desired even for sequential languages; for relaxed-memory concurrency, they almost inevitably lead to ambiguity, error and confusion. Instead, we need rigorous (but readable)

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

POPL'11, January 26–28, 2011, Austin, Texas, USA.
Copyright © 2011 ACM 978-1-4503-0490-0/11/01...\$10.00

mathematical semantics, with tool support to explore the consequences of the definitions on examples, proofs of theoretical results, and support for testing implementations. Interestingly, the style of semantics needed is quite different from that for conventional sequential languages, as are the tools and theorems.

Contributions In this paper we establish a mathematically rigorous semantics for C++ concurrency, described in Section 2 and with further examples in Section 3. It is *precise*, formalised in Isabelle/HOL [Isa], and is *complete*, covering essentially all the concurrency-related semantics from the draft standard, without significant idealisation or abstraction. It includes the data-race-freedom (DRF) guarantee of SC behaviour for race-free code, locks, SC atomics, the various flavours of low-level atomics, and fences. It covers initialisation but not allocation, and does not address the non-concurrent aspects of C++. Our model builds on the informal-mathematics treatment of the DRF guarantee by Boehm and Adve [BA08]. We have tried to make it as *readable* as possible, using only minimal mathematical machinery (mostly just sets, relations and first-order logic with transitive closure) and introducing it with a series of examples. Finally, wherever possible it is a *faithful* representation of the draft standard and of the intentions of its authors, as far as we understand them.

In developing our semantics, we identified a number of issues in several drafts of the C++0x standard, discussed these with members of the concurrency subgroup, and made suggestions for changes. These are of various kinds, including editorial clarifications, substantive changes, and some open questions. We discuss a selection of these in Section 4. The standards process for C++0x is ongoing: the current version is at the time of writing the ‘final committee draft’, leaving a small window for further improvements. That for C1X is at an earlier stage, though the two should be compatible.

As a theoretical test of our semantics, we prove a correctness result (§5) for the proposed prototype x86 implementation of the C++ concurrency primitives [Ter08] with respect to our x86-TSO memory model [SSO⁺10, OSS09]. We show that any x86-TSO execution of a translated C++ candidate execution gives behaviour that the C++ semantics would admit, which involves delicate issues about initialisation. This result establishes some confidence in the model and is a key step towards a verified compilation result about translation of programs.

Experience shows that tool support is needed to work with an axiomatic relaxed memory model, to develop an intuition for what behaviour it admits and forbids, and to explore the consequences of proposed changes to the definitions. At the least, such a tool should take an example program, perhaps annotated with constraints on the final state or on the values read from memory, and find and display all the executions allowed by the model. This can be combinatorially challenging, but for C++ it turns out to be feasible, for typical test examples, to enumerate the possible witnesses. We have therefore built a CPPMEM tool (§6) that exhaustively considers all the possible witnesses, checking each one with code automatically generated from the Isabelle/HOL axiomatic model (§6). The front-end of the tool takes a program in a fragment of C++ and runs a symbolic operational semantics to calculate possible memory accesses and constraints. We have also explored the use of a model generator (the SAT-solver-based Kodkod [TJ07], via the Isabelle Nitpick interface [BN10]) to find executions more efficiently, albeit with less assurance. All of the examples in this paper have been checked (and their executions drawn) using CPPMEM.

Our work provides a basis for improving both standards, both by the specific points we raise and by giving a precisely defined checkpoint, together with our CPPMEM tool for exploring the behaviour of examples in our model and in variants thereof. The C and C++ language standards are a central interface in today’s computational infrastructure, between what a compiler (and hardware) should im-

plement, on the one hand, and what programmers can rely on, on the other. Clarity is essential for both sides, and a mathematically precise semantics is a necessary foundation for any reasoning about concurrent C and C++ programs, whether it be dynamic analysis, model-checking, static analysis and abstract interpretation, program logics, or interactive proof. It is also a necessary precondition for work on compositional semantics of such programs.

2. C++0x Concurrency, as Formalised

Here we describe C++ concurrency incrementally, starting with single-threaded programs and then adding threads and locks, SC atomics, and low-level atomics (release/acquire, relaxed, and release/consume). Our model also covers fences, but we omit the details here. In this section we do not distinguish between the C++ draft standard, which is the work of the Concurrency subcommittee of WG21, and our formal model, but in fact there are substantial differences between them. We highlight some of these (and our rationale for various choices) in Section 4. Our memory model is expressed as a standalone Isabelle/HOL file and the complete model is available online [BOS]; here we give the main definitions, automatically typeset (and lightly hand-edited in a few cases) from the Isabelle/HOL source.

The semantics of a program p will be a set of allowed executions X . Some C++ programs are deemed to have *undefined behaviour*, meaning that an implementation is unconstrained, e.g. if any execution contains a data race. Accordingly, we define the semantics in two phases: first we calculate a set of pre-executions which are admitted by the operational semantics and are *consistent* (defined in the course of this section). Then, if there is a pre-execution in that set with a race of some kind, the semantics indicates undefined behaviour by giving NONE, otherwise it gives all the pre-executions. In more detail, a candidate execution X is a pair $(X_{\text{opsem}}, X_{\text{witness}})$, where the first component is given by the operational semantics and the second is an existential witness of some further data; we introduce the components of both as we go along. The top-level definition of the memory model, then, is:

```
cpp-memory-model opsem (p : program) =
  let pre_executions = {(Xopsem, Xwitness) .
    opsem p Xopsem ∧
    consistent_execution (Xopsem, Xwitness)} in
  if ∃X ∈ pre_executions .
    (indeterminate_reads X ≠ {}) ∨
    (unsequenced_races X ≠ {}) ∨
    (data_races X ≠ {})
  then NONE
  else SOME pre_executions
```

2.1 Single-threaded programs

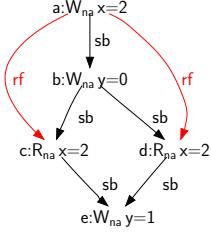
We begin with the fragment of the model that deals with single-threaded programs, which serves to introduce the basic concepts and notation we use later.

As usual for a relaxed memory model, different threads can have quite different views of memory, so the semantics cannot be expressed in terms of changes to a monolithic memory (e.g. a function from locations to values). Instead, an execution consists of a set of *memory actions* and various relations over them, and the memory model axiomatises constraints on those.

For example, consider the program on the left below. This has only one execution, shown on the right. There are five actions, labelled (a)–(e), all by the same thread (their thread ids are elided). These are all non-atomic memory reads (R_{na}) or writes (W_{na}), with their address (x or y) and value (0,1, or 2). Actions (a) and (b) are the initialisation writes, (c) and (d) are the reads of the operands of the $==$ operator, and (e) is a write of the result of $==$. The evaluations of the arguments to $==$ are *unsequenced* in C++ (as are arguments

to functions), meaning that they could be in either order, or even overlapping. Evaluation order is expressed by the *sequenced-before* (sb) relation, a strict preorder over the actions, that here does not order (c) and (d). The two reads both *read from* the same write (a), indicated by the *rf* relation.

```
int main() {
    int x = 2;
    int y = 0;
    y = (x == x);
    return 0;
}
```



The set of actions and the sequenced-before relation are given by the operational semantics (so are part of the X_{opsem}); the *rf* relation is existentially quantified (part of the X_{witness}), as in general there may be many writes that each read might read from.

In a non-SC semantics, the constraint on reads cannot be simply that they read from the ‘most recent’ write, as there is no global linear time. Instead, they are constrained here using a *happens-before* relation, which in the single-threaded case coincides with sequenced-before. Non-atomic reads have to read from a *visible side effect*, a write to the same location that happens-before the read but is not happens-before-hidden, i.e., one for which there is no intervening write to the location in happens-before. We define the visible-side-effect relation below, writing it with an arrow. The auxiliary functions *is_write* and *is_read* pick out all actions (including atomic actions and read-modify-writes but not lock or unlock actions) that write or read memory.

$$\begin{aligned} a \xrightarrow{\text{visible-side-effect}} b &= \\ a \xrightarrow{\text{happens-before}} b \wedge & \\ \text{is_write } a \wedge \text{is_read } b \wedge \text{same_location } a \ b \wedge & \\ \neg(\exists c. (c \neq a) \wedge (c \neq b)) \wedge & \\ \text{is_write } c \wedge \text{same_location } c \ b \wedge & \\ a \xrightarrow{\text{happens-before}} c \xrightarrow{\text{happens-before}} b & \end{aligned}$$

The constraint on the values read by nonatomic reads is in two parts: the reads-from map must satisfy a well-formedness condition (not shown here), saying that reads cannot read from multiple writes, that they must be at the same location and have the same value as the write they read from, and so on. More interestingly, it must respect the visible side effects, in the following sense.

$$\begin{aligned} \text{consistent_reads_from_mapping} = & \\ (\forall b. (\text{is_read } b \wedge \text{is_at_non_atomic_location } b) \implies & \\ (\text{if } (\exists a_{\text{use}}, a_{\text{use}} \xrightarrow{\text{visible-side-effect}} b) & \\ \text{then } (\exists a_{\text{use}}, a_{\text{use}} \xrightarrow{\text{visible-side-effect}} b \wedge a_{\text{use}} \xrightarrow{\text{rf}} b) & \\ \text{else } \neg(\exists a. a \xrightarrow{\text{rf}} b))) \wedge & \\ [...] & \end{aligned}$$

If a read has no visible side effects (e.g. reading an uninitialized variable), there can be no *rf* edge. This is an *indeterminate read*, and the program is deemed to have undefined behaviour.

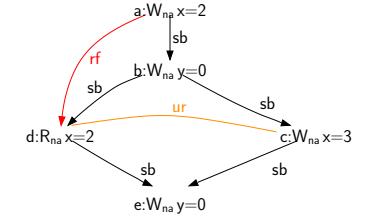
$$\text{indeterminate_reads} = \{b. \text{is_read } b \wedge \neg(\exists a. a \xrightarrow{\text{rf}} b)\}$$

A pre-execution has an *unsequenced-race* if there is a write and another access to the same location, on the same thread that are unsequenced.

$$\begin{aligned} \text{unsequenced_races} = & \{(a, b). \\ (a \neq b) \wedge \text{same_location } a \ b \wedge (\text{is_write } a \vee \text{is_write } b) \wedge & \\ \text{same_thread } a \ b \wedge & \\ \neg(a \xrightarrow{\text{sequenced-before}} b \vee b \xrightarrow{\text{sequenced-before}} a)\} & \end{aligned}$$

Programs with an execution that contains an unsequenced race (ur), like the one below, have undefined behaviour.

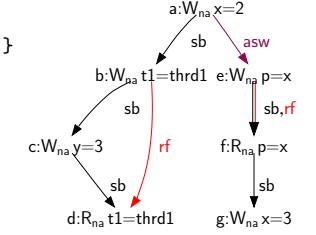
```
int main() {
    int x = 2;
    int y = 0;
    y = (x == (x=3));
    return 0;
}
```



2.2 Threads, Data Races, and Locks

We now integrate C++0x threads into the model. The following program spawns a thread that writes 3 to x and concurrently writes 3 into y in the original thread.

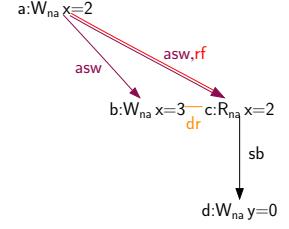
```
void foo(int* p) {*p = 3;}
int main() {
    int x = 2;
    int y;
    thread t1(foo, &x);
    y = 3;
    t1.join();
    return 0;
}
```



The thread creation gives rise to *additional-synchronizes-with* (asw) edges (here $a \xrightarrow{\text{asw}} e$) from sequenced-before-maximal actions of the parent thread before the thread creation to sequenced-before-minimal edges of the child. As we shall see, these edges are also incorporated, indirectly, into happens-before. They are generated by the operational semantics, so are another component of an X_{opsem} .

Thread creation gives rise to many memory actions (for passing function arguments and writing and reading the thread id) which clutter examples, so for this paper we usually use a more concise parallel composition, written $\{\{\dots\} \parallel \{\dots\}\}$:

```
int main() {
    int x = 2;
    int y;
    \{\{\ x = 3;
    \|\| y = (x==3);
    \}\};
    return 0;
}
```



This example exhibits a *data race* (dr): two actions at the same location, on different threads, not related by happens-before, at least one of which is a write.

$$\begin{aligned} \text{data_races} = & \{(a, b). \\ (a \neq b) \wedge \text{same_location } a \ b \wedge (\text{is_write } a \vee \text{is_write } b) \wedge & \\ \neg \text{same_thread } a \ b \wedge & \\ \neg(\text{is_atomic_action } a \wedge \text{is_atomic_action } b) \wedge & \\ \neg(a \xrightarrow{\text{happens-before}} b \vee b \xrightarrow{\text{happens-before}} a)\} & \end{aligned}$$

If there is a pre-execution of a program that has a data-race, then, as with unsequenced-races, that program has undefined behaviour.

Data races can be prevented by using mutexes, as usual. These give rise to *lock* and *unlock* memory actions on the mutex location, and a pre-execution has a relation, *sc*, as part of X_{witness} that totally orders such actions. A *consistent_locks* predicate checks that lock and unlock actions are appropriately alternating. Moreover, these actions on each mutex create *synchronizes-with* edges from every

unlock to every lock that is ordered after it in sc . The synchronizes-with relation is a derived relation, calculated from a candidate execution, which contains mutex edges, the additional-synchronizes-with edges (e.g. from thread creation), and other edges that we will come to.

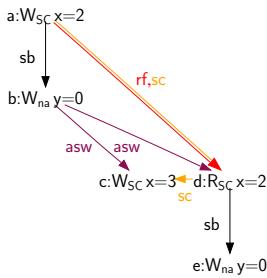
$$a \xrightarrow{\text{synchronizes-with}} b = \\ (* - \text{additional synchronisation, from thread create etc.} - *) \\ a \xrightarrow{\text{additional-synchronizes-with}} b \vee \\ (\text{same_location } a \ b \wedge a \in \text{actions} \wedge b \in \text{actions} \wedge (\\ (* - \text{mutex synchronization} - *) \\ (\text{is_unlock } a \wedge \text{is_lock } b \wedge a \xrightarrow{sc} b) \vee \\ [...]))$$

For multi-threaded programs with locks but without atomicics, happens-before is the transitive closure of the union of the sequenced-before and synchronizes-with relations. The definition of a visible side effect and the conditions on the reads-from relation are unchanged from the single-threaded case.

2.3 SC Atomics

For simple concurrent accesses to shared memory that are not protected by locks, C++0x provides *sequentially consistent atomics*. Altering the racy example from above to use an atomic object x and SC atomic operations, we have the following, in which the concurrent access to x is not considered a data race, and so the program does not have undefined behaviour.

```
int main() {
    atomic_int x;
    x.store(2);
    int y = 0;
    {{ x.store(3);
    ||| y = ((x.load())==3);
    }};
    return 0;
}
```



Semantically, this is because SC atomic operations are totally ordered by sc , and so can be thought of as interleaving with each other in a global time-line. Their semantics are covered in detail in [BA08] and we will describe their precise integration into happens-before in the following section.

Initialisation of an atomic object is by non-atomic stores (to avoid the need for a hardware fence for every such initialisation), and those non-atomic stores *can* race with other actions at the location unless the program has some synchronisation. Non-initialisation SC-atomic accesses are made with atomic read, write and read-modify-write actions that do not race with each other.

2.4 Low-level Atomics

SC atomics are expensive to implement on most multiprocessors, e.g. with the suggested implementations for an SC atomic load being `LOCK XADD(0)` on x86 [Ter08] and `hwsync; ld; cmp; bc; isync` on Power [MS10]; the `LOCK'd` instruction and the `hwsync` may take 100s of cycles. They also provide more synchronisation than needed for many concurrent idioms. Accordingly, C++0x includes several weaker variants: atomic actions are parametrised by a *memory order*, mo , that specifies how much synchronisation and ordering is required. The strongest ordering is required for `MO_SEQ_CST` actions (which is the default, as used above), and the weakest for `MO_RELAXED` actions. In between there are `MO_RELEASE/MO_ACQUIRE` and `MO_RELEASE/MO_CONSUME` pairs, and `MO_ACQ_REL` with both acquire and release semantics.

2.5 Types and Relations

Before giving the semantics of low-level atomics, we summarise the types and relations of the model. There are base types of action ids aid , thread ids tid , locations l , and values v . As we have seen already, actions can be non-atomic reads or writes, or mutex locks or unlocks. Additionally, there are atomic reads, writes, and read-modify-writes (with a memory order parameter mo) and fences (also with an mo parameter). We often elide the thread ids.

action =	
$aid, tid:R_{na} l = v$	non-atomic read
$aid, tid:W_{na} l = v$	non-atomic write
$aid, tid:R_{mo} l = v$	atomic read
$aid, tid:W_{mo} l = v$	atomic write
$aid, tid:RMW_{mo} l = v_1/v_2$	atomic read-modify-write
$aid, tid:L l$	lock
$aid, tid:U l$	unlock
$aid, tid:F_{mo}$	fence

The `is_read` predicate picks out non-atomic and atomic reads and atomic read-modify-writes; the `is_write` predicate picks out non-atomic and atomic writes and atomic read-modify-writes.

Locations are subject to a very weak type system: each location stores a particular kind of object, as determined by a *location-kind* map. The atomic actions can only be performed on ATOMIC locations. The non-atomic reads and writes can be performed on either ATOMIC or NON_ATOMIIC locations. Locks and unlocks are *mutex* actions and can only be performed on MUTEX locations. These are enforced (among other sanity properties) by a `well_formed_threads` predicate; we elide the details here.

The X_{opsem} part of a candidate execution X consists of a set of thread ids, a set of actions, a location typing, and three binary relations over its actions: *sequenced-before* (sb), *additional-synchronized-with* (asw), and *data-dependency* (dd). We have already seen the first two: *sequenced-before* contains the intra-thread edges imposed by the C++ evaluation order, and *additional-synchronized-with* contains additional edges from thread creation and thread join (among others). Data dependence will be used for release/consume atomics (in §2.8). These are all relations that are decided by the syntactic structure of the source code and the path of control flow, and so the set of possible choices for an X_{opsem} can be calculated by the operational semantics without reference to the memory model (with reads taking unconstrained values).

The X_{witness} part of a candidate execution X consists of a further three binary relations over its actions: rf , sc , and *modification order* (mo). The rf reads-from map is a relation containing edges to read actions from the write actions whose values they take, and edges to each lock action from the last unlock of its mutex. The sequentially consistent order sc is a total order over all actions that are `MO_SEQ_CST` and all mutex actions. The modification order (mo) is a total order over all writes at each atomic location (leaving writes at different locations unrelated), and will be used to express coherence conditions. These relations are existentially quantified in the definition of `cpp_memory_model`, and for each X_{opsem} admitted by the operational semantics there may be many choices of an X_{witness} that give a consistent execution (each of which may or may not have a data race, unsequenced race, or indeterminate read).

The happens-before relation, along with several others, are derived from those in X_{opsem} and X_{witness} .

2.6 Release/Acquire Synchronization

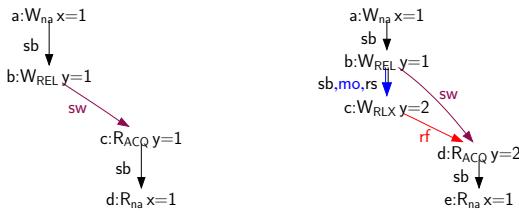
An atomic write or fence is a *release* if it has the memory order `MO_RELEASE`, `MO_ACQ_REL` or `MO_SEQ_CST`. Atomic reads or fences with order `MO_ACQUIRE`, `MO_ACQ_REL` or `MO_SEQ_CST`, and fences with order `MO_CONSUME`, are *acquire* actions.

Pairs of a write-release and a read-acquire support the following programming idiom. Here one thread writes some data x (perhaps spanning multiple words) and then sets a flag y while the other spins until the flag is set and then reads the data.

```
// sender           // receiver
x = ...           while (0 == y);
y = 1;             r = x;
```

The desired guarantee here is that the receiver must see the data writes of the sender (in more detail, that the receiver cannot see any values of data that precede those writes in modification order). This can be achieved with an atomic store of y , annotated MO_RELEASE, and an atomic load of y annotated MO_ACQUIRE. The reads and writes of x can be nonatomic.

In the model, any instance of a read-acquire that reads from a write-release gives rise to a *synchronizes-with* edge, e.g. as on the left below (where the *rf* edges are suppressed).



For such programs (in fact for any program without release/consume atomics), happens-before is still the transitive closure of the union of the sequenced-before and synchronizes-with relations, so here $a \xrightarrow{\text{happens-before}} d$ and (d) is obliged to read from (a).

In this case, the read-acquire synchronizes with the write-release that it reads from. More generally, the read-acquire can synchronize with a write-release (to the same location) that is before the write that it reads from. To define this precisely, we need to use the modification order of a candidate execution and to introduce the derived notion of a *release sequence*, of writes that follow (in some sense) a write-acquire.

For example, in the fragment of an execution on the right above, the read-acquire (d) synchronizes with the write-release (b) by virtue of the fact that (d) reads from another write to the same location, (c), and (b) precedes (c) in the modification order (mo) for that location.

The modification order of a candidate execution (here $b \xrightarrow{\text{modification-order}} c$) totally orders all of the write actions on each atomic location, in this case y . It must also be consistent with happens-before, in the sense below.

```
consistent_modification_order =
  ( $\forall a. \forall b. a \xrightarrow{\text{modification-order}} b \implies \text{same\_location } a b$ )  $\wedge$ 
  ( $\forall l \in \text{locations\_of } \text{actions}. \text{case } \text{location\_kind } l \text{ of}$ 
    ATOMIC  $\rightarrow$  (
      let  $\text{actions\_at\_l} = \{a. (\text{location } a = \text{SOME } l)\}$  in
      let  $\text{writes\_at\_l} = \{a \in \text{actions\_at\_l}. (\text{is\_store } a \vee$ 
        strict_total_order_over  $\text{writes\_at\_l}$ 
         $\xrightarrow{\text{modification-order}} | \text{actions\_at\_l}\}) \wedge$ 
        (* happens-before at the writes of  $l$  is a subset of mo for  $l$  *)
         $\frac{\text{happens-before}}{| \text{writes\_at\_l}} \subseteq \xrightarrow{\text{modification-order}} \wedge$ 
        [...])
    )  $\parallel - \rightarrow$  (
      let  $\text{actions\_at\_l} = \{a. (\text{location } a = \text{SOME } l)\}$  in
       $\xrightarrow{\text{modification-order}} | \text{actions\_at\_l} = \{\})$ 
```

In the example, the release action (b) has a release sequence [(b),(c)], a contiguous sub-sequence of modification order on the location of the write-release. The release sequence is headed by the release and can be followed by writes from the same thread or read-modify-writes from any thread; other writes by other threads break the sequence. We represent a release sequence not by the list of actions but by a relation from the head to all the elements, as the order is given by modification order. In figures we usually suppress the reflexive edge from the head to itself.

$\text{rs_element } rs_head a =$
 $\text{same_thread } a rs_head \vee \text{is_atomic_rmw } a$

$a_{\text{rel}} \xrightarrow{\text{release-sequence}} b =$
 $\text{is_at_atomic_location } b \wedge$
 $\text{is_release } a_{\text{rel}} \wedge ($
 $(b = a_{\text{rel}}) \vee$
 $(\text{rs_element } a_{\text{rel}} b \wedge a_{\text{rel}} \xrightarrow{\text{modification-order}} b \wedge$
 $(\forall c. a_{\text{rel}} \xrightarrow{\text{modification-order}} c \xrightarrow{\text{modification-order}} b \implies$
 $\text{rs_element } a_{\text{rel}} c))$

A write-release *synchronizes-with* a read-acquire if both act on the same location and the release sequence of the release contains the write that the acquire reads from. In the example $b \xrightarrow{\text{release-sequence}} c \xrightarrow{rf} d$, so we have $b \xrightarrow{\text{synchronizes-with}} d$. The definition below covers mutexes and thread creation (in additional-synchronizes-with) but elides the effects of fences.

$a \xrightarrow{\text{synchronizes-with}} b =$
(* – additional synchronization, from thread create etc. – *)
 $a \xrightarrow{\text{additional-synchronizes-with}} b \vee$
(same_location $a b \wedge a \in \text{actions} \wedge b \in \text{actions} \wedge$
(* – mutex synchronization – *)
(is_unlock $a \wedge \text{is_lock } b \wedge a \xrightarrow{\text{sc}} b$) \vee
(* – release/acquire synchronization – *)
(is_release $a \wedge \text{is_acquire } b \wedge \neg \text{same_thread } a b \wedge$
 $(\exists c. a \xrightarrow{\text{release-sequence}} c \xrightarrow{rf} b)) \vee$
[...]))

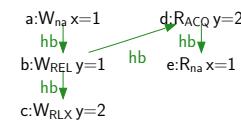
The modification order and the sc order we saw earlier must also be consistent, in the following sense:

consistent_sc_order =

```
let sc_happens_before =  $\xrightarrow{\text{happens-before}} | \text{all_sc_actions}$  in
let sc_mod_order =  $\xrightarrow{\text{modification-order}} | \text{all_sc_actions}$  in
strict_total_order_over all_sc_actions ( $\xrightarrow{\text{sc}}$ )  $\wedge$ 
 $\frac{\text{sc\_happens\_before}}{\text{sc}} \subseteq \xrightarrow{\text{sc}} \wedge$ 
 $\frac{\text{sc\_mod\_order}}{\text{sc}} \subseteq \xrightarrow{\text{sc}}$ 
```

2.7 Constraining Atomic Read Values

The values that can be read by an atomic action depend on happens-before, derived from sequenced-before and synchronizes-with. We return to the execution fragment shown on the right in the previous subsection, showing a transitive reduction of happens-before that coincides with its constituent orderings.



An atomic action must read a write that is in one of its *visible sequences of side effects*; in this case (d) either reads (b) or (c).

A visible sequence of side effects of a read is a contiguous subsequence of modification order, headed by a visible side effect of the read, where the read does not happen before any member of the sequence. We represent a visible sequence of side effects not as a list but as a set of actions in the tail of the sequence (we are not concerned with their order).

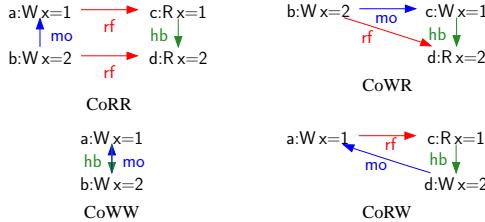
$$\begin{aligned} \text{visible_sequence_of_side_effects_tail } vsse_head b = \\ \{c. vsse_head \xrightarrow{\text{modification-order}} c \wedge \\ \neg(b \xrightarrow{\text{happens-before}} c) \wedge \\ (\forall a. vsse_head \xrightarrow{\text{modification-order}} a \xrightarrow{\text{modification-order}} c \\ \implies \neg(b \xrightarrow{\text{happens-before}} a))\} \end{aligned}$$

We define *visible-sequences-of-side-effects* to be the binary relation relating atomic reads to their visible-side-effect sets (now including the visible side effects themselves). The atomic read must read from a write in one of these sets.

We can now extend the previous definition of the consistent reads-from predicate to be the conjunction of the read-restrictions on nonatomic and atomic actions, and a constraint ensuring read-modify-write atomicity.

$$\begin{aligned} \text{consistent_reads_from_mapping} = \\ (\forall b. (\text{is_read } b \wedge \text{is_at_non_atomic_location } b) \implies \\ (\text{if } (\exists a_{vse}. a_{vse} \xrightarrow{\text{visible-side-effect}} b) \\ \text{then } (\exists a_{vse}. a_{vse} \xrightarrow{\text{visible-side-effect}} b \wedge a_{vse} \xrightarrow{rf} b) \\ \text{else } \neg(\exists a. a \xrightarrow{rf} b)) \wedge \\ (\forall b. (\text{is_read } b \wedge \text{is_at_atomic_location } b) \implies \\ (\text{if } (\exists (b', vsse) \in \text{visible-sequences-of-side-effects}. (b' = b)) \\ \text{then } (\exists (b', vsse) \in \text{visible-sequences-of-side-effects}. \\ (b' = b) \wedge (\exists c \in vsse. c \xrightarrow{rf} b)) \\ \text{else } \neg(\exists a. a \xrightarrow{rf} b))) \wedge \\ (\forall (a, b) \in \xrightarrow{rf}. \text{is_atomic_rmw } b \\ \implies a \xrightarrow{\text{modification-order}} b) \wedge \\ [\dots] \end{aligned}$$

A candidate execution is also required to be free of the following four execution fragments. This property is called coherence.



CoRR Two reads ordered by happens-before may not read two writes that are modification ordered in the other direction.

CoWR It is forbidden to read from a write that is happens-before hidden by a later write in modification order.

CoWW Happens-before and modification-order may not disagree.

CoRW The union of the reads-from map, happens-before and modification-order must be acyclic.

Finally, we restrict SC reads: If there is no preceding write in sc order, then there is no extra restriction. Otherwise, they must read from the last prior write in sc order, from a non-atomic write that follows it in modification order, or from any non-SC atomic write.

2.8 Release/Consume Atomics

On multiprocessors with weak memory orders, notably Power, release/acquire pairs are cheaper to implement than sequentially

consistent atomics but still significantly more expensive than plain stores and loads. For example, the proposed Power implementation of load-acquire, `ld`; `cmp`; `bc`; `isync`, involves an `isync` [MS10]. However, Power (and also ARM) does guarantee that certain dependencies in an assembly program are respected, and in many cases those suffice, making the `isync` sequence unnecessary. As we understand it, this is the motivation for introducing a *read-consume* variant of read-acquire atomics. On a stronger processor (e.g. a TSO x86 or Sparc), or one where those dependencies are not respected, read-consume would be implemented just as read-acquire.

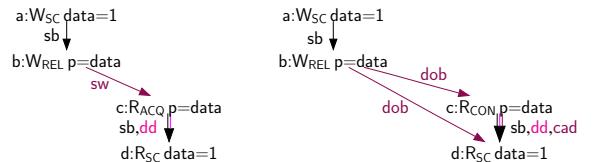
Read-consume enables efficient implementations of algorithms that use pointer reassignment for commits of their data, e.g. read-copy-update [MW]. For example, suppose one thread writes some data (perhaps spanning multiple words) then writes the address of that data to a shared atomic pointer, while the other thread reads the shared pointer, dereferences it and reads the data.

<pre>// sender data = ... p = &data;</pre>	<pre>// receiver r1 = p r2 = *r1; // data</pre>
--	---

Here there is a dependency at the receiver from the read of `p` to the read of `data`. This can be expressed using a write-release and an atomic load of `p` annotated MO_CONSUME:

```
int main() {
    int data; atomic_address p;
    {{ { data=1;
        p.store(&data, mo_release); }
    }|| printf("%d\n", *(p.load(mo_consume)) );
}};
return 0; }
```

As we saw in §2.6, the semantics of release/acquire pairs introduced synchronizes-with edges, and happens-before includes the transitive closure of synchronizes-with and sequenced-before — for a release/acquire version of this example, we would have the edges on the left below, and hence $a \xrightarrow{\text{happens-before}} d$.



For release/consume, the key fact is that there is a *data dependency* (dd) from (c) to (d), as shown on the right. The (dd) edge is provided by the operational semantics and gives rise to a *carries-a-dependency-to* (cad) edge, which extends data dependency with thread-local reads-from relationships:

$$a \xrightarrow{\text{carries-a-dependency-to}} b = \\ a ((\xrightarrow{rf} \cap \xrightarrow{\text{sequenced-before}}) \cup \xrightarrow{\text{data-dependency}})^+ b$$

In turn, this gives rise to a *dependency-ordered-before* (dob) edge, which is the release/consume analogue of the release/acquire synchronizes-with edge. This involves release sequences as before (in the example just the singleton [(b)]):

$$\begin{aligned} a \xrightarrow{\text{dependency-ordered-before}} d = \\ a \in \text{actions} \wedge d \in \text{actions} \wedge \\ (\exists b. \text{is_release } a \wedge \text{is_consume } b \wedge \\ (\exists e. a \xrightarrow{\text{release-sequence}} e \xrightarrow{rf} b) \wedge \\ (b \xrightarrow{\text{carries-a-dependency-to}} d \vee (b = d))) \end{aligned}$$

2.9 Happens-before

Finally, we can define the complete happens-before relation. To accommodate MO_CONSUME, and specifically the fact that release/consume pairs only introduce happens-before relations to dependency-successors of the consume, *not* to all actions that are sequenced-after it, the definition is in two steps. First, we define *inter-thread-happens-before*, which combines synchronizes-with and dependency-ordered-before, allowing transitivity with sequenced-before on the left for both and on the right only for synchronizes-with:

$$\begin{aligned} \text{inter-thread-happens-before} &= \\ \text{let } r &= \xrightarrow{\text{synchronizes-with}} \cup \\ &\quad \xrightarrow{\text{dependency-ordered-before}} \cup \\ &\quad (\xrightarrow{\text{synchronizes-with}} \circ \xrightarrow{\text{sequenced-before}}) \text{ in} \\ &(\xrightarrow{r} \cup (\xrightarrow{\text{sequenced-before}} \circ \xrightarrow{r}))^+ \end{aligned}$$

In any execution, this must be acyclic:

$$\begin{aligned} \text{consistent_inter_thread_happens_before} &= \\ &\text{irreflexive } (\xrightarrow{\text{inter-thread-happens-before}}) \end{aligned}$$

Happens-before (which is thereby also acyclic) is then just the union with sequenced-before:

$$\begin{aligned} \text{happens-before} &= \\ &\text{sequenced-before} \cup \text{inter-thread-happens-before} \end{aligned}$$

2.10 Putting it together

Given a candidate execution $X = (X_{\text{opsem}}, X_{\text{witness}})$, we can now calculate the derived relations:

release-sequence (§2.6), *hypothetical-release-sequence* (a variant of *release-sequence* used in the fence semantics), *synchronizes-with* (§2.2, §2.6), *carries-a-dependency-to* (§2.8), *dependency-ordered-before* (§2.8), *inter-thread-happens-before* (§2.8), *happens-before* (§2.1, §2.2, §2.3, §2.8), *visible-side-effect* (§2.1), and *visible-sequences-of-side-effects* (§2.7).

The definition of *consistent_execution* used at the start of Section 2 is then simply the conjunction of the predicates we have defined:

$$\begin{aligned} \text{consistent_execution} &= \\ &\text{well_formed_threads} \wedge \quad (\text{§2.5, defn. elided}) \\ &\text{consistent_locks} \wedge \quad (\text{§2.2, defn. elided}) \\ &\text{consistent_inter_thread_happens_before} \wedge \quad (\text{§2.8}) \\ &\text{consistent_sc_order} \wedge \quad (\text{§2.6}) \\ &\text{consistent_modification_order} \wedge \quad (\text{§2.6}) \\ &\text{well_formed_reads_from_mapping} \wedge \quad (\text{§2.1, defn. elided}) \\ &\text{consistent_reads_from_mapping} \quad (\text{§2.1, §2.7}) \end{aligned}$$

The acyclicity check on inter-thread-happens-before, and the subtlety of the non-transitive happens-before relation, are needed only for release/consume pairs:

Theorem 1. *For an execution with no consume operations, the consistent_inter_thread_happens_before condition of consistent_execution is redundant.*

Theorem 2. *If a consistent execution has no consume operations, happens-before is transitive.*

The proofs are by case analysis and induction on the size of possible cycles.

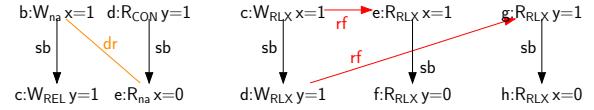
3. Examples

We now illustrate the varying strength of the different memory orders by showing the semantics of some ‘classic’ examples. In

all cases, variants of the examples with SC atomics do not have the weak-memory behaviour. As in our other diagrams, to avoid clutter we only show selected edges, and we omit the C++ sources for these examples, which are available on-line [BOS].

Store Buffering (SB) Here two threads write to separate locations and then each reads from the other location. In Total Store Order (TSO) models both can read from before (w.r.t. coherence) the other write in the same execution. In C++0x this behaviour is allowed if those four actions are relaxed, for release/consume pairs and for release/acquire pairs. This behaviour is not allowed for the same program using sequentially consistent consistent atomics (with non-atomic initialisation).

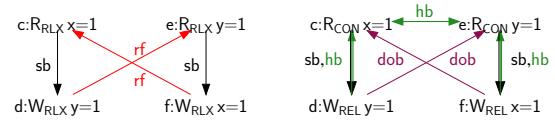
Message Passing (MP) Here one thread (non-atomically) writes data and then an atomic flag while a second thread waits for the flag and then (non-atomically) reads data; the question is whether it is guaranteed to see the data written by the first. As we saw in §2.6, with a release/acquire pair it is. A release/consume pair gives the same guarantee iff there is a dependency between the reads, otherwise there is a consistent execution (on the left) in which there is a data race (here the second thread sees the initial value of x ; the candidate execution in which the second thread sees the write $x=1$ is ruled out as that does not happen-before the read and so is not a visible side effect).



The same holds with relaxed flag operations.

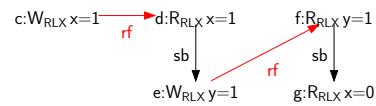
In a variant in which all writes and reads are release/consumes or relaxed atomics, eliminating the race, and there are two copies of the reading thread, the two reading threads can see the two writes of the writing thread in opposite orders (as on the right above) — consistent with what one might see on Power, for example.

Load Buffering (LB) In this dual of the SB example the question is whether the two reads can both see the (sequenced-before) later write of the other thread in the same execution. With relaxed atomics this is allowed, as on the left:



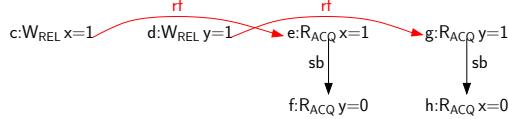
but with release/consumes (with dependencies) it is not (as on the right above), because inter-thread-happens-before would be cyclic. It is not allowed for release/acquire and sequentially consistent atomics (which are stronger than release/consumes with dependencies), because of the cyclic inter-thread-happens-before and stronger inter-thread ordering.

Write-to-Read Causality (WRC) Here the first thread writes to x ; the second reads from that and then (w.r.t. sequenced-before) writes to y ; the third reads from that and then (w.r.t. sequenced-before) reads x . The question is whether it is guaranteed to see the first thread’s write.



With relaxed atomics, this is not guaranteed, as shown above, while with release/acquires it is, as the *synchronizes-with* edges in the inter-thread-happens-before relation interfere with the required read-from map.

Independent Reads of Independent Writes (IRIW) Here the first two threads write to different locations; the question is whether the second two threads can see those writes in different orders. With relaxed, release/acquire, or release/consume atomics, they can.



4. From standard to formalisation and back

We developed the model presented in Section 2 by a lengthy iterative process: building formalisations of various drafts of the standard, and of Boehm and Adve’s model without low-level atomics [BA08]; considering the behaviour of examples, both by hand and with our tool; trying to prove properties of the formalisations; and discussing issues with members of the Concurrency subcommittee of the C++ Standards Committee (TC1/SC22/WG21). To give a flavour of this process, and to explain how our formalisation differs from the current draft (the final committee draft, N3092) of the standard, we describe a selection of debatable issues. This also serves to bring out the delicacy of the standard, and the pitfalls of prose specification, even when carried out with great care. We have made suggestions for technical or editorial changes to the draft for many of these points and it seems likely that they will be incorporated.

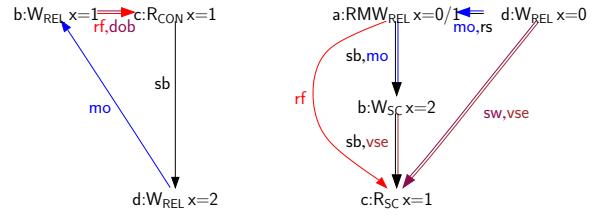
We begin with two straightforward drafting issues, easily fixed. Then there are three substantial semantic problems in N3092 where we have proposed solutions. Finally, there is an outstanding question that warrants further investigation.

‘Subsequent’ in visible sequences of side effects N3092 defines: *The visible sequence of side effects on an atomic object M, with respect to a value computation B of M, is a maximal contiguous sub-sequence of side effects in the modification order of M, where the first side effect is visible with respect to B, and for every subsequent side effect, it is not the case that B happens before it.* However, if every element in a vsse happens-before a read, the read should not take the value of the visible side effect. Following discussion, we formalise this without the *subsequent*.

Additional happens-before edges There are 6 places where N3092 adds happens-before relationships explicitly (in addition to those from sequenced-before and inter-thread-happens-before), e.g. between the invocation of a thread constructor and the function that the thread runs. As happens-before is carefully *not* transitive, such edges would not be transitive with (e.g.) sequenced-before. Accordingly, we instead add them to the synchronizes-with relation; for those within our C++ fragment, our operational semantics introduces them into additional-synchronizes-with.

Acyclicity of happens-before N3092 defines happens-before, making plain that it is not necessarily transitive, but does not state whether it is required to be acyclic (or whether, perhaps, a program with a cyclic execution is deemed to have undefined behaviour). The release/consume LB example of §3 has a cyclic inter-thread-happens-before, as shown there, but is otherwise a consistent execution. After discussion, it seems clear that executions with cyclic inter-thread-happens-before (or, equivalently, cyclic happens-before) should not be considered, so we impose that explicitly.

Coherence requirements The draft standard enforced only two of the four coherence requirements presented in §2.7, CoRR and CoWW. In the absence of CoRW and CoWR, the following executions were allowed.

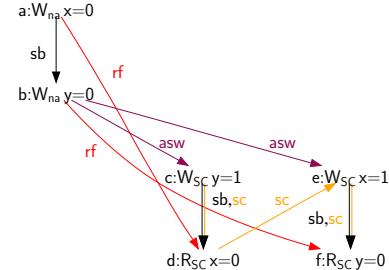


The execution on the left violates CoRW by containing a cycle of happens-before and modification order edges, allowed only due to the lack of transitivity of happens-before. The execution on the right violates CoWR by having a read from a write (the read-modify-write (a)) that is sequenced-before-hidden by (c). Actions (b) and (c) are shown as SC atomics for emphasis.

Furthermore, the draft standard refers to ‘*the*’ visible sequence of side-effects, suggesting uniqueness. Nevertheless, it allows valid executions that have more than one, relying on the lack of transitivity of happens-before as in the CoRW execution above.

These behaviours are surprising and were not intended by the designers.

Sequential consistency for SC atomics The promise of sequential consistency to the non-expert programmer is a central design choice of C++0x and is stated directly by N3092: *memory_order_seq_cst ensures sequential consistency [...] for a program that is free of data races and uses exclusively memory_order_seq_cst operations.* Unfortunately N3092 allows the following non-sequentially consistent execution of the SB example with SC atomics (initialisation writes, such as (a) and (b), are non-atomic so that they need not be compiled with memory fences):



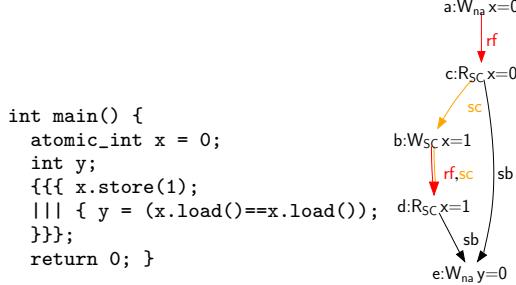
We devised a stronger restriction on the values that may be read by SC atomics, stated in §2.7, that does provide sequential consistency here.

Overlapping executions and thin-air reads In a C++0x program that gives rise to the relaxed LB example in §3, the written value 1 might have been concrete in the program source. Alternatively, one might imagine a *thin-air read*: the program below has the same execution, and here there is *no* occurrence of 1 in the program source.

```
int main() {
    int r1, r2;
    atomic_int x = 0;
    atomic_int y = 0;
    {{ { r1 = x.load(mo_relaxed)); y.store(r1,mo_relaxed); }
      ||| { r2 = y.load(mo_relaxed)); x.store(r2,mo_relaxed); }
    } }
    return 0; }
```

This would be surprising, and in fact would not happen with typical hardware and compilers. In the Java Memory Model [MPA05], much of the complexity of the model arises from the desire to outlaw thin-air reads, which there is essential to prevent forging of pointers. N3092 also attempts to forbid thin air reads, with:

An atomic store shall only store a value that has been computed from constants and program input values by a finite sequence of program evaluations, such that each evaluation observes the values of variables as computed by the last prior assignment in the sequence. This seems to be overly constraining. For example, two subexpression evaluations (in separate threads) can overlap (e.g. if they are the arguments of a function call) and can contain multiple actions. With relaxed atomics there can be consistent executions in which it is impossible to disentangle the two into any sequence, for example as below, where the SC-write of x must be between the two reads of x . In our formalisation we currently do not impose any thin-air condition.



5. Correctness of a Proposed x86 Implementation

The C++0x memory model has been designed with compilation to the various target architectures in mind, and prototype implementations of the atomic primitives have been proposed. For example, the following table presents an x86 prototype by Terekhov [Ter08]:

Operation	x86 Implementation
Load non-SC	mov
Load Seq_cst	lock xadd(0)
Store non-SC	mov
Store Seq_cst	lock xchg
Fence non-SC	no-op
Fence Seq_cst	mfence

This is a simple mapping from individual source-level atomic operations to small fragments of assembly code, abstracting from the vast and unrelated complexities of compilation of a full C++ language (argument evaluation order, object layout, control flow, etc.). Proposals for the Power [MS10] and other architectures follow the same structure, although, as they have more complex memory models than the x86, the assembly code for some of the operations is more intricate.

Verifying that these prototypes are indeed correct implementations of the model is a crucial part of validating the design. Furthermore, as they represent the atomic-operation parts of efficient compilers (albeit without fence optimisations), they can directly form an important part of a verified C++ compiler, or inform the design and verification of a compiler with memory-model-aware optimisations.

Here, we prove a version of the above prototype x86 implementation [Ter08] correct with respect to our x86-TSO semantics [SSZN⁺09, OSS09, SSO⁺10]. Following the prototype, we ignore lock and unlock operations, as well as forks and joins, all of which require significant runtime or operating system support in addition to the x86 hardware. We also ignore sequentially consistent fences, but cover all other fences. We do consider read-modify-write actions, implementing them with x86 LOCK'd read-modify-writes; and we include non-atomic loads and stores, which can map to multiple x86 loads and stores, respectively. The prototype mapping is simple, and x86-TSO is reasonably well-understood, so this should be seen as a test of the C++ memory model.

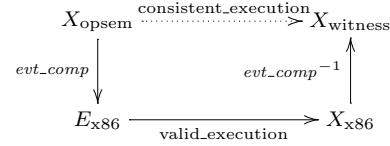
In x86-TSO, an operational semantics gives meaning to an assembly program by creating an *x86 event structure* E_{x86} (analogous to X_{opsem}) comprising a set of events and an intra-thread *program-order* relation (analogous to sequenced-before) that orders events according to the program text. Events can be reads, writes, or fences, and certain instructions (e.g. CMPXCHG) create *locked* sets of events that execute atomically. Corresponding to $X_{witness}$, there are *x86 execution witnesses* X_{x86} which comprise a reads-from mapping and a memory order, which is a partial order over reads and writes that is total on the writes. The remainder of the axiomatisations are very different: x86-TSO has no concept of release, acquire, visible side effect, etc.

Abstracting out the rest of the compiler To discuss the correctness of the proposed mapping in isolation, without embarking on a verification of some particular full compiler, we work solely in terms of candidate executions and memory models.

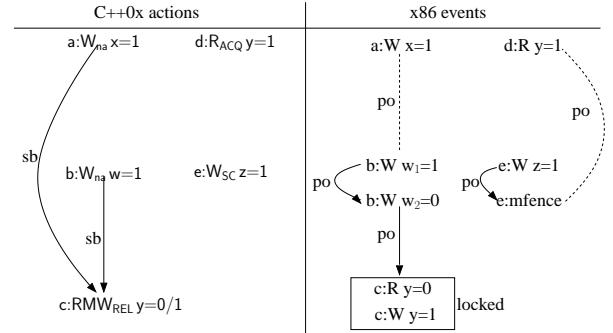
First, we lift the mapping between instructions to a nondeterministic translation *action_comp* from C++ actions to small x86 event structures, e.g. relating an atomic read-modify-write action to the events of the corresponding x86 LOCK'd instruction.

To define what it means for the mapping to be correct, suppose we have a C++ program p with no undefined behaviour and an X_{opsem} which is allowed by its operational semantics. We regard an abstract compiler *evt_comp* as taking such an X_{opsem} and giving an x86 event structure E_{x86} , respecting the *action_comp* mapping but with some freedom in the resulting x86 program order.

We say the mapping is correct if given such an abstract compiler, the existence of a valid x86-TSO execution witness for E_{x86} implies the existence of a consistent C++ execution witness $X_{witness}$ for the original actions X_{opsem} . We prove this by lifting such an x86 execution witness to a C++ consistent execution, as illustrated below.



Below we show an X_{opsem} and E_{x86} that could be related by *evt_comp*. The dotted lines indicate some of the x86 program ordering decisions that the compiler must make, but which *evt_comp* does not constrain.



In more detail, we use two existentially quantified helper functions *locn_comp* and *tid_comp* to encapsulate the details of a C++ compiler's data layout, its mapping of C++ locations to x86 addresses, and the mapping of C++ threads to x86 threads.

Given a C++ location and value, *locn_comp* produces a finite mapping from x86 addresses to x86 values. The domain of the finite map is the set of x86 addresses that corresponds to the C++ location, and the mapping itself indicates how a C++ value is laid out across the x86 addresses. A well-formed *locn_comp* has the following properties: it is injective; the address calculation

cannot depend on the value; each C++ location has an x86 address; different C++ locations have non-overlapping x86 address sets; and an atomic C++ location has a single x86 address, although a non-atomic location can have several addresses (e.g. for a multi-word object).

Finally, the *evt_comp* relation specifies valid translations, applying *action_comp* with a well-formed *locn_comp* and also constraining how events from different actions relate: no single x86 instruction instance can be used by multiple C++ actions, and the x86 *program-order* relation must respect C++'s *sequenced-before*. The detailed definitions, and the proof of the following theorem, are available online [BOS].

Theorem 3. *Let p be a C++ program that has no undefined behaviour. Suppose also that p contains no SC fences, forks, joins, locks, or unlocks. Then the x86 mapping is correct in the sense above. That is, if actions, sequenced-before, and location-kind are members of the X_{opsem} part of a candidate execution resulting from the operational semantics of p, then the following holds:*

$$\begin{aligned} \forall \text{comp } & \text{ locn_comp } \text{ tid_comp } X_{\text{x86}}. \\ & \text{evt_comp } \text{ comp } \text{ locn_comp } \text{ tid_comp } \text{ actions} \\ & \quad \text{sequenced-before } \text{ location-kind} \wedge \\ & \quad \text{valid_execution } (\cup_{a \in \text{actions}} (\text{comp } a)) X_{\text{x86}} \Rightarrow \\ & \quad \exists X_{\text{witness}}. \text{ consistent_execution } (X_{\text{opsem}}, X_{\text{witness}}) \end{aligned}$$

Proof outline. X_{x86} includes a reads-from map and a memory ordering relation that is total on all memory writes. To build X_{witness} , we lift a C++ reads-from map and modification order from these through *comp* (e.g., $a \xrightarrow{rf} b$ iff $\exists (e_1 \in \text{comp } a) (e_2 \in \text{comp } b)$: $e_1 \xrightarrow{\text{x86-}rf} e_2$). We create an *sc* ordering by restricting the X_{x86} memory ordering to the events that originate in sequentially consistent atomics, and linearising it using the proof technique from our previous triangular-race freedom work for x86-TSO [Owe10]. We then lift that through *comp*. The proof now proceeds in three steps:

- 1) We first show that if $a \xrightarrow{\text{happens-before}} b$ and there are x86 events e_1 and e_2 such that $e_1 \in \text{comp } a$ and $e_2 \in \text{comp } b$, then e_1 precedes e_2 in either X_{x86} 's memory order or program order. We have machine-checked this step in HOL-4 [HOL].¹

This property establishes that, in some sense, x86-TSO has a stronger memory model than C++, and so any behaviour allowed by the former should be allowed by the latter. However, things are not quite so straightforward.

- 2) Check that X_{witness} is a consistent_execution. Most cases are machine checked in HOL; some are only pencil-and-paper. Many rely upon the property from 1. For example, in showing that (at a non-atomic location) if $a \xrightarrow{rf} b$ then $a \xrightarrow{\text{visible-side-effect}} b$, we note that if there were a write c to the same location such that $a \xrightarrow{\text{happens-before}} c \xrightarrow{\text{happens-before}} b$, then using the property from 1, there is an x86 write event in *comp* c that would come between the events of *comp* a and *comp* b in X_{x86} , thus meaning that they would not be in X_{x86} 's reads-from map, contradicting the construction of X_{witness} 's reads-from map.
- 3) In some cases, some of the properties required for 2 might be false. For example, in showing that $a \xrightarrow{rf} b$ implies $a \xrightarrow{\text{visible-side-effect}} b$, we need to show that $a \xrightarrow{\text{happens-before}} b$. Even though there is such a relationship at the x86 level, it does not necessarily exist in C++. In general, x86 executions can establish reads-from relations

¹ The C++ model is in Isabelle/HOL, but x86-TSO is in HOL-4. We support the proof with a semi-automated translation from Isabelle/HOL to HOL-4.

that are prohibited in C++. Similarly, for non-atomic accesses that span multiple x86 addresses, the lifted reads from-map might not be well-formed.

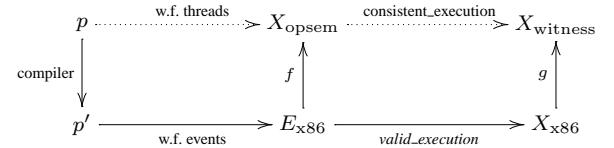
We show that if one of these violations of 2 arises, then the original C++ program has a data race. We find a minimum violation in X_{x86} , again using techniques from our previous work [Owe10]. Next we can remove the violation, resulting in a consistent X_{witness} for a prefix of the execution, then we add the bad action, note that it creates a data race, and allow the program to complete in any way. The details of this part are by pencil-and-paper proof. \square

Sequentially consistent atomics The proposal above includes two implementations of sequentially consistent atomic reads and writes; one with the x86 locked instructions, and the other with fence instructions on both the reads and writes. However, we can prove that it suffices either to place an mfence before every sc read, or after every sc write, but that it is not necessary to do both. In practice, placing the fence after the sc writes is expected to yield higher performance.

This optimisation is a direct result of using triangular-race freedom (TRF) [Owe10] to construct the *sc* ordering in proving Theorem 3. Roughly, our TRF theorem characterises when x86-TSO executions are not sequentially consistent; it uses a pattern, called a triangular race, involving an x86-level data race combined with a write followed, on the same thread, by a read without a fence (or locked instruction) in between. If no such pattern exists, then an execution X_{x86} can be linearised such that each read reads from the most recent preceding write.

Although the entirety of an execution witness X_{x86} might contain triangular races and therefore not be linearisable, by restricting attention to only sc reads and writes we get a subset of the execution that is TRF, as long as there is a fence between each sc read and write on the same thread. Linearising this subset guarantees the relevant property of X_{witness} 's *sc* ordering: that if a and b are sequentially consistent atomics and $a \xrightarrow{rf} b$, then a immediately precedes b in *sc* restricted to that address.

Compiler correctness Although we translate executions instead of source code, Theorem 3 could be applied to full source-to-assembly compilers that follow the prototype implementation. The following diagram presents the overall correctness property.



If, once we use f , we can then apply *evt_comp* to get the same event set back, i.e., informally, $\text{evt_comp}(f(E)) = E$, then Theorem 3 ensures that the compiler respects the memory model, and so we only need to verify that it respects the operational semantics. Thus, our result applies to compilers that do not optimise away any instructions that *evt_comp* will produce. These restrictions apply to the code generation phase; the compiler can perform any valid source-to-source optimisations before generating x86 code.

6. Tool support for exploring the model

Given a relatively complex axiomatic memory model, as we presented in Section 2, it is often hard to immediately see the consequences of the axioms, or what behaviour they allow for particular programs. Our CPPMEM tool takes a program in a fragment of C++0x and calculates the set of its executions allowed by the memory model, displaying them graphically.

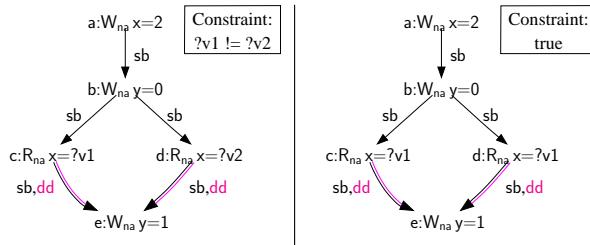
The tool has three main components: an executable symbolic operational semantics to build the X_{opsem} parts of the candidate executions X of a program; a search procedure to enumerate the possible X_{witness} for each of those; and a checking procedure to calculate the derived relations and predicates of the model for each $(X_{\text{opsem}}, X_{\text{witness}})$ pair, to check whether it is consistent and whether it has data races, unsequenced races or indeterminate reads.

Of these, the checker is the most subtle, since the only way to intuitively understand it is to understand the model itself (which is what the tool is intended to aid with), and thus bugs are hard to catch. It also has to be adapted often as the model is developed. We therefore use Isabelle/HOL code generation [Haf09] to build the checker directly from our Isabelle/HOL axiomatisation, to keep the checker and our model in exact correspondence and reduce the possibility for error.

The operational semantics Our overall semantics is stratified: the memory model is expressed as a predicate on the actions and relations of a candidate execution. This means we need an operational semantics of an unusual form to generate all such candidates. In a setting with a global SC memory, the values read by loads can be determined immediately, but here, for example for a program with a single load, in principle we have to generate a large set of executions, each with a load event with one of the possible values. We make this executable by building a symbolic semantics in which the values in actions can be either concrete values or unification variables (shown as $?v$). Control flow can depend on the values read, so the semantics builds a set of these actions (and the associated relations), together with constraints on the values, for each control-flow path of the program. For each path, the associated constraint is solved at the end; those with unsatisfiable constraints (indicating unreachable execution paths) are discarded.

The tool is designed to support litmus test examples of the kind we have seen, not arbitrary C++ code. These do not usually involve many C++ features, and the constraints required are propositional formulae over equality and inequality constraints over symbolic and concrete values. It is not usually important in litmus tests to do more arithmetic reasoning; one could imagine using an SMT solver if that were needed, but for the current constraint language, a standard union-find unifier suffices. The input program is processed by the CIL parser [NMRW02], extended with support for atomics. We use Graphviz [GN00] to generate output. We also allow the user to add explicit constraints on the value read by a memory load in a C++ source program, to pick out candidate executions of interest; to selectively disable some of the checks of the model; and to de-clutter the output by suppressing actions and edges.

As an example, consider the first program we saw, in §2.1. There are two possibilities: the reads of x either read the same value or different values, and hence the operational semantics gives the two candidate executions and constraints below:



Later, the memory model will rule out the left execution, since there is no way to read anything but 2 at x .

The semantics maintains an environment mapping identifiers to locations. For loads, the relevant location is found in that, and a fresh variable $?v$ is generated to represent the value read.

Other constructs typically combine the actions of their subterms and also build the relations (sequenced-before, data-dependency, etc.) of X_{opsem} as appropriate. For example, for the `if` statement, the execution path splits and two execution candidates will be generated. The one for the true branch has an additional constraint, that the value returned by the condition expression is true (in the C/C++ sense, i.e. different from 0), and the candidate for the false branch constrains the value to be false. There are also additional *sequenced-before* and *control-dependency* edges from the actions in the condition expression to actions in the branch.

Choosing instantiations of existential quantifiers Given the X_{opsem} part of a finite candidate execution, the X_{witness} part is existentially quantified over a finite but potentially large set. In the worst case, with m reads and n writes, all sequentially consistent (atomic), to the same location, and with the same value, there might be $O(m^{(n+1)} \cdot m! \cdot (m+n)!)$ possible choices of an *rf*, *modification-order* and *sc* relation. In practice, though, litmus tests are much simpler: there are typically no more than 2 or 3 writes to any one location, so we avoid coding up a sophisticated memory-model-aware search procedure in favour of keeping this part of the code simple. For the examples shown here, the tool has to check at most a few thousand alternatives, and takes less than 0.2 seconds. The most complex example we tested (IRIW with all SC) had 162,000 cases to try, and the overall time taken was about 5 minutes.

Checking code extracted from Isabelle We use Isabelle/HOL code generation to produce a checker as an OCaml module, which can be linked in with the rest of the CPPSEM tool. Our model is stated in higher-order logic with sets and relations. Restricted to finite sets, the predicates and definitions are almost all directly executable, within the domain of the code generation tool (which implements finite sets by OCaml lists). For a few cases (e.g importantly transitive closure), we had to write a more efficient function and an Isabelle/HOL proof of equivalence. The overall checking time per example is on the order of 10^{-3} seconds, for examples with around 10 actions.

6.1 Finite model generation with Nitpick/Kodkod

Given the X_{opsem} part of a candidate execution, the space of possible X_{witness} parts which will lead to valid executions can be explored by tools for model generation. We reused the operational semantics above to produce a X_{opsem} from a program, and then posed problems to Nitpick, a finite model generator built into Isabelle [BN10]. Nitpick is a frontend to Kodkod, a model generator for first order logic extended with relations and transitive closure based on a state-of-the-art SAT solver. Nitpick translates higher-order logic formulae to first-order formulae within Kodkod syntax. For small programs, Nitpick can easily find some consistent execution, or report that none such exists, in a few seconds. In particular, for the IRIW-SC example mentioned above, Nitpick takes 130 seconds to report that no execution exists, while other examples take around 5 seconds. Of course, Nitpick can also validate an execution X with both parts X_{opsem} and X_{witness} concretely specified, but this is significantly slower than running the Isabelle-extracted validator. The bottleneck here is the translation process, which is quite involved.

7 Related work

The starting points for this paper were the draft standard itself and the work of Boehm and Adve [BA08], who introduced the rationale for the C++0x overall design and gave a model for non-atomic, lock, and SC atomic operations, without going into low-level atomics or fences in any detail. It was expressed in informal mathematics, an intermediate point between the prose of the standard and

the mechanised definitions of our model. The most closely related other work is the extensive line of research on the Java Memory Model (JMM) [Pug00, MPA05, CKS07, SA08, TVD10]. Java imposes very different constraints to C++ as there it is essential to prohibit thin-air reads, to prevent forging of pointers and hence security violations.

Turning to the sequential semantics of C++, Norrish has recently produced an extensive HOL4 model [Nor08], and Zalewski [Zal08] formalised the proposed extension of C++ concepts.

There is also a body of research on tool support for memory models, notably including (among others) the MEMSAT of Torlak et al. [TVD10], which uses Kodkod for formalisations of the JMM, and NEMOSFINDER of Yang et al. [YGLS04], which is based on Prolog encodings of memory models and included an Itanium specification. Building on our previous experience with the MEMEVENTS tool for hardware (x86 and Power) memory models [SSZN⁺09, OSS09, SSO⁺10, AMSS10], we designed CPP-MEM to eliminate the need for hand-coding of the tool to reflect changes in the model, by automatically generating the checker code from the Isabelle/HOL definition. We made it practically usable for exploring our non-idealised (and hence rather complex) C++0x model by a variety of user-interface features, letting us explore the executions of a program in various ways.

8. Conclusion

We have put the semantics of C++ and C concurrency on a mathematically sound footing, following the current final committee draft standard as far as possible, except as we describe in §4. This should support future improvements to the standard and the development of semantics, analysis, and reasoning tools for concurrent systems code.

Having done so, the obvious question is the extent to which the formal model could be incorporated as a *normative* part of a future standard. The memory model is subtle but it uses only simple mathematical machinery, of various binary relations over a fixed set of concrete actions, that can be visualised graphically. There is a notational problem: one would probably have to translate (automatically or by hand) the syntax of first-order logic into natural language, to make it sufficiently widely accessible. But given that, we suspect that the formal model would be clearer than the current ‘standardsese’ for all purposes, not only for semantics and analysis.

Acknowledgements This work would not have been possible without discussions with members of the C++ Concurrency subcommittee and the `cpp-threads` mailing list, including Hans Boehm, Lawrence Crowl, Peter Dimov, Doug Lea, Nick McLaren, Paul McKenney, Clark Nelson, and Anthony Williams. Jasmin Blanchette assisted us with the Nitpick tool. We acknowledge funding from EPSRC grants EP/F036345, EP/H005633, EP/H027351, and EP/F067909.

References

- [AB10] S. V. Adve and H.-J. Boehm. Memory models: A case for rethinking parallel languages and hardware. *C. ACM*, 2010.
- [AMSS10] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Fences in weak memory models. In *Proc. CAV*, 2010.
- [ARM08] ARM. *ARM Architecture Reference Manual (ARMv7-A and ARMv7-R edition)*. April 2008.
- [BA08] H.-J. Boehm and S.V. Adve. Foundations of the C++ concurrency memory model. In *Proc. PLDI*, 2008.
- [Bec10] P. Becker, editor. *Programming Languages — C++: Final Committee Draft*. 2010. ISO/IEC JTC1 SC22 WG21 N3092.
- [BN10] Jasmin Christian Blanchette and Tobias Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In *Proc. ITP*, 2010.
- [BOS] www.cl.cam.ac.uk/users/pes20/cpp.
- [C1X] JTC1/SC22/WG14 — C. <http://www.open-std.org/jtc1/sc22/wg14/>.
- [CKS07] P. Cenciarelli, A. Knapp, and E. Sibilio. The Java memory model: Operationally, denotationally, axiomatically. In *Proc. ESOP*, 2007.
- [GN00] E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *Softw. Pract. Exper.*, 30(11):1203–1233, 2000.
- [Haf09] Florian Haftmann. *Code Generation from Specifications in Higher-Order Logic*. PhD thesis, TU München, 2009.
- [HOL] The HOL 4 system. <http://hol.sourceforge.net/>.
- [Int02] Intel. A formal specification of Intel Itanium processor family memory ordering. <http://www.intel.com/design/itanium/downloads/251429.htm>, October 2002.
- [Isa] Isabelle 2009-2. <http://isabelle.in.tum.de/>.
- [Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, C-28(9):690–691, 1979.
- [MPA05] J. Manson, W. Pugh, and S.V. Adve. The Java memory model. In *Proc. POPL*, 2005.
- [MS10] P. E. McKenney and R. Silvera. Example POWER implementation for C/C++ memory model. <http://www.rdrop.com/users/paulmck/scalability/paper/N2745r.2010.02.19a.html>, 2010.
- [MW] P. E. McKenney and J. Walpole. What is RCU, fundamentally? Linux Weekly News, <http://lwn.net/Articles/262464/>.
- [NMRW02] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Proc. CC*, 2002.
- [Nor08] M. Norrish. A formal semantics for C++. Technical report, NICTA, 2008.
- [OSS09] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In *Proc. TPHOLs*, 2009.
- [Owe10] S. Owens. Reasoning about the implementation of concurrency abstractions on x86-TSO. In *Proc. ECOOP*, 2010.
- [Pow09] Power ISA Version 2.06. IBM, 2009.
- [Pug00] W. Pugh. The Java memory model is fatally flawed. *Concurrency - Practice and Experience*, 12(6), 2000.
- [SA08] J. Ševčík and D. Aspinall. On validity of program transformations in the Java memory model. In *ECOOP*, 2008.
- [Spa] The SPARC architecture manual, v. 9. <http://developers.sun.com/solaris/articles/sparcv9.pdf>.
- [SSO⁺10] P. Sewell, S. Sarkar, S. Owens, F. Zappa Nardelli, and M. O. Myreen. x86-TSO: A rigorous and usable programmer’s model for x86 multiprocessors. *C. ACM*, 53(7):89–97, 2010.
- [SSZN⁺09] S. Sarkar, P. Sewell, F. Zappa Nardelli, S. Owens, T. Ridge, T. Braibant, M. Myreen, and J. Alglave. The semantics of x86-CC multiprocessor machine code. In *Proc. POPL*, 2009.
- [Ter08] A. Terekhov. Brief tentative example x86 implementation for C/C++ memory model. `cpp-threads` mailing list, <http://www.decadent.org.uk/pipermail/cpp-threads/2008-December/001933.html>, Dec. 2008.
- [TJ07] E. Torlak and D. Jackson. Kodkod: a relational model finder. In *Proc. TACAS*, 2007.
- [TVD10] E. Torlak, M. Vaziri, and J. Dolby. MemSAT: checking axiomatic specifications of memory models. In *PLDI*, 2010.
- [YGLS04] Y. Yang, G. Gopalakrishnan, G. Lindstrom, and K. Slind. Nemos: A framework for axiomatic and executable specifications of memory consistency models. In *IPDPS*, 2004.
- [Zal08] M. Zalewski. *Generic Programming with Concepts*. PhD thesis, Chalmers University, November 2008.

Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects

Maged M. Michael

Abstract—Lock-free objects offer significant performance and reliability advantages over conventional lock-based objects. However, the lack of an efficient portable lock-free method for the reclamation of the memory occupied by dynamic nodes removed from such objects is a major obstacle to their wide use in practice. This paper presents *hazard pointers*, a memory management methodology that allows memory reclamation for arbitrary reuse. It is very efficient, as demonstrated by our experimental results. It is suitable for user-level applications—as well as system programs—with dependence on special kernel or scheduler support. It is wait-free. It requires only single-word reads and writes for memory access in its core operations. It allows reclaimed memory to be returned to the operating system. In addition, it offers a lock-free solution for the ABA problem using only practical single-word instructions. Our experimental results on a multiprocessor system show that the new methodology offers equal and, more often, significantly better performance than other memory management methods, in addition to its qualitative advantages regarding memory reclamation and independence of special hardware support. We also show that lock-free implementations of important object types, using hazard pointers, offer comparable performance to that of efficient lock-based implementations under no contention and no multiprogramming, and outperform them by significant margins under moderate multiprogramming and/or contention, in addition to guaranteeing continuous progress and availability, even in the presence of thread failures and arbitrary delays.

Index Terms—Lock-free, synchronization, concurrent programming, memory management, multiprogramming, dynamic data structures.

1 INTRODUCTION

A shared object is *lock-free* (also called nonblocking) if it guarantees that whenever a thread executes some finite number of steps toward an operation on the object, *some* thread (possibly a different one) must have made progress toward completing an operation on the object, during the execution of these steps. Thus, unlike conventional lock-based objects, lock-free objects are immune to deadlock when faced with thread failures, and offer robust performance, even when faced with arbitrary thread delays.

Many algorithms for lock-free dynamic objects have been developed, e.g., [11], [9], [23], [21], [4], [5], [16], [7], [25], [18]. However, a major concern regarding these objects is the reclamation of the memory occupied by removed nodes. In the case of a lock-based object, when a thread removes a node from the object, it is easy to guarantee that no other thread will subsequently access the memory of that node, before it is reused or reallocated. Consequently, it is usually safe for the removing thread to reclaim the memory occupied by the removed node (e.g., using `free`) for arbitrary future reuse by the same or other threads (e.g., using `malloc`).

This is not the case for a typical lock-free dynamic object, when running in programming environments without support for automatic garbage collection. In order to guarantee lock-free progress, each thread must have unrestricted opportunity to operate on the object, at any time. When a thread removes a node, it is possible that some other contending thread—in the course of its lock-free operation—has earlier read a reference to that node, and is

about to access its contents. If the removing thread were to reclaim the removed node for arbitrary reuse, the contending thread might corrupt the object or some other object that happens to occupy the space of the freed node, return the wrong result, or suffer an access error by dereferencing an invalid pointer value. Furthermore, if reclaimed memory is returned to the operating system (e.g., using `munmap`), access to such memory locations can result in fatal access violation errors. Simply put, the memory reclamation problem is how to allow the memory of removed nodes to be freed (i.e., reused arbitrarily or returned to the OS), while guaranteeing that no thread accesses free memory, and how to do so in a lock-free manner.

Prior methods for allowing node reuse in dynamic lock-free objects fall into three main categories. 1) The IBM tag (update counter) method [11], which hinders memory reclamation for arbitrary reuse and requires double-width instructions that are not available on 64-bit processors. 2) Lock-free reference counting methods [29], [3], which are inefficient and use unavailable strong multiaddress atomic primitives in order to allow memory reclamation. 3) Methods that depend on aggregate reference counters or per-thread timestamps [13], [4], [5]. Without special scheduler support, these methods are blocking. That is, the failure or delay of even one thread can prevent an aggregate reference counter from reaching zero or a timestamp from advancing and, hence, preventing the reuse of unbounded memory.

This paper presents *hazard pointers*, a methodology for memory reclamation for lock-free dynamic objects. It is efficient; it takes constant expected amortized time per retired node (i.e., a removed node that is no longer needed by the removing thread). It offers an upper bound on the total number of retired nodes that are not yet eligible for reuse, regardless of thread failures or delays. That is, the failure or delay of any number of threads can prevent only a

• The author is with the IBM T.J. Watson Research Center, PO Box 218, Yorktown Heights, NY 10598. E-mail: magedm@us.ibm.com.

Manuscript received 15 Apr. 2003; revised 22 Oct. 2003; accepted 2 Jan. 2004. For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0058-0403.

bounded number of retired nodes from being reused. The methodology does not require the use of double-width or strong multiaddress atomic primitives. It uses only single-word reads and writes for memory access in its core operations. It is wait-free [8], i.e., progress is guaranteed for active threads individually, not just collectively; thus, it is also applicable to wait-free algorithms without weakening their progress guarantee. It allows reclaimed memory to be returned to the operating system. It does not require any special support from the kernel or the scheduler.

The core idea is to associate a number (typically one or two) of single-writer multireader shared pointers, called *hazard pointers*, with each thread that intends to access lock-free dynamic objects. A hazard pointer either has a null value or points to a node that may be accessed later by that thread without further validation that the reference to the node is still valid. Each hazard pointer can be written only by its owner thread, but can be read by other threads.

The methodology requires lock-free algorithms to guarantee that no thread can access a dynamic node at a time when it is possibly removed from the object, unless at least one of the thread's associated hazard pointers has been pointing to that node continuously, from a time when the node was guaranteed to be reachable from the object's roots. The methodology prevents the freeing of any retired node continuously pointed to by one or more hazard pointers of one or more threads from a point prior to its removal.

Whenever a thread retires a node, it keeps the node in a private list. After accumulating some number R of retired nodes, the thread scans the hazard pointers of other threads for matches for the addresses of the accumulated nodes. If a retired node is not matched by any of the hazard pointers, then it is safe for this node to be reclaimed. Otherwise, the thread keeps the node until its next scan of the hazard pointers.

By organizing a private list of snapshots of nonnull hazard pointers in a hash table that can be searched in constant expected time, and if the value of R is set such that $R = H + \Omega(H)$, where H is the total number of hazard pointers, then the methodology is guaranteed in every scan of the hazard pointers to identify $\Theta(R)$ nodes as eligible for arbitrary reuse, in $O(R)$ expected time. Thus, the expected amortized time complexity of processing each retired node until it is eligible for reuse is constant.

Note that a small number of hazard pointers per thread can be used to support an arbitrary number of objects as long as that number is sufficient for supporting each object individually. For example, in a program where each thread may operate arbitrarily on hundreds of shared objects that each requires up to two hazard pointers per thread (e.g., hash tables [25], FIFO queues [21], LIFO stacks [11], linked lists [16], work queues [7], and priority queues [9]), only a total of two hazard pointers are needed per thread.

Experimental results on an IBM RS/6000 multiprocessor system show that the new methodology, applied to lock-free implementations of important object types, offers equal and, more often, significantly better performance than other memory management methods, in addition to its qualitative advantages regarding memory reclamation and independence of special hardware support. We also show that lock-free implementations of important object types, using hazard pointers, offer comparable performance to that of efficient lock-based implementations under no contention and no multiprogramming, and outperform them by significant

margins under moderate multiprogramming and/or contention, in addition to guaranteeing continuous progress and availability even in the presence of thread failures and arbitrary delays.

The rest of this paper is organized as follows: In Section 2, we discuss the computational model for our methodology and memory management issues for lock-free objects. In Section 3, we present the hazard pointer methodology. In Section 4, we discuss applying hazard pointers to lock-free algorithms. In Section 5, we present our experimental performance results. In Section 6, we discuss related work and summarize our results.

2 PRELIMINARIES

2.1 The Model

The basic computational model for our methodology is the asynchronous shared memory model. Formal descriptions of this model appeared in the literature, e.g., [8]. Informally, in this model, a set of threads communicate through primitive memory access operations on a set of shared memory locations. Threads run at arbitrary speeds and are subject to arbitrary delays. A thread makes no assumptions about the speed or status of any other thread. That is, it makes no assumptions about whether another thread is active, delayed, or crashed, and the time or duration of its suspension, resumption, or failure. If a thread crashes, it halts execution instantaneously.

A shared object occupies a set of shared memory locations. An object is an instance of an implementation of an abstract object type, that defines the semantics of allowable operations on the object.

2.2 Atomic Primitives

In addition to atomic reads and writes, primitive operations on shared memory locations may include stronger atomic primitives such as compare-and-swap (CAS) and the pair load-linked/store-conditional (LL/SC). CAS takes three arguments: the address of a memory location, an expected value, and a new value. If and only if the memory location holds the expected value, the new value is written to it, atomically. A Boolean return value indicates whether the write occurred. That is, $CAS(addr, exp, new)$ performs the following atomically:

```
{if (*addr ≠ exp) return false; *addr ← new; return true; }.
```

LL takes one argument: the address of a memory location, and returns its contents. SC takes two arguments: the address of a memory location and a new value. Only if no other thread has written the memory location since the current thread last read it using LL, the new value is written to the memory location, atomically. A Boolean return value indicates whether the write occurred. An associated instruction, Validate (VL), takes one argument: the address of a memory location, and returns a Boolean value that indicates whether any other thread has written the memory location since the current thread last read it using LL.

For practical architectural reasons, none of the architectures that support LL/SC (Alpha, MIPS, PowerPC) support VL or the ideal semantics of LL/SC as defined above. None allow nesting or interleaving of LL/SC pairs, and most prohibit any memory access between LL and SC. Also, all such architectures, occasionally—but not infinitely often—allow SC to fail spuriously; i.e., return false even when the

```

// Hazard pointer record
structure HPRecType
{ HP[K]:*NodeType; Next:*HPRecType; }
// The header of the HPRec list
HeadHPRec : *HPRecType;
// Per-thread private variables
rlist : listType; // initially empty
rcount : integer; // initially 0

```

Fig. 1. Types and structures.

memory location was not written by other threads since it was last read by the current thread using LL. For all the algorithms presented in this paper, $CAS(addr, exp, new)$ can be implemented using restricted LL/SC as follows:

```

{do {if (LL(addr)≠exp) return false; }
until SC(addr,new); return true; }.

```

Most current mainstream processor architectures support either CAS or restricted LL/SC on aligned single words. Support for CAS and LL/SC on aligned double-words is available on most 32-bit architectures (i.e., support for 64-bit instructions), but not on 64-bit architecture (i.e., no support for 128-bit instructions).

2.3 The ABA problem

A different but related problem to memory reclamation is the ABA problem. It affects almost all lock-free algorithms. It was first reported in the documentation of CAS on the IBM System 370 [11]. It occurs when a thread reads a value A from a shared location, and then other threads change the location to a different value, say B, and then back to A again. Later, when the original thread checks the location, e.g., using read or CAS, the comparison succeeds, and the thread erroneously proceeds under the assumption that the location has not changed since the thread read it earlier. As a result, the thread may corrupt the object or return a wrong result.

The ABA problem is a fundamental problem that must be prevented regardless of memory reclamation. Its relation to memory reclamation is that solutions of the latter problem, such as automatic garbage collection (GC) and the new methodology, often prevent the ABA problem as a side-effect with little or no additional overhead.

This is true for most lock-free dynamic objects. But, it should be noted that a common misconception is that GC inherently prevents the ABA problem in all cases. However, consider a program that moves dynamic nodes back and forth between two lists (e.g., LIFO stacks [11]). The ABA problem is possible in such a case, even with perfect GC.

The new methodology is as powerful as GC with respect to ABA prevention in lock-free algorithms. That is, if a lock-free algorithm is ABA-safe under GC, then applying hazard pointers to it makes it ABA-safe without GC. As we discuss in a recent report [19], lock-free algorithms can *always* be made ABA-safe under GC, as well as using hazard pointers in the absence of GC. In the rest of this paper, when discussing the use of hazard pointers for ABA prevention in the absence of support for GC, we assume that lock-free algorithms are already ABA-safe under GC.

```

RetireNode(node:*NodeType) {
    rlist.push(node);
    rcount++;
    if (rcount ≥ R)
        Scan(HeadHPRec);
}

```

Fig. 2. The RetireNode routine.

3 THE METHODOLOGY

The new methodology is primarily based on the observation that, in the vast majority of algorithms for lock-free dynamic objects, a thread holds only a small number of references that may later be used without further validation for accessing the contents of dynamic nodes, or as targets or expected values of ABA-prone atomic comparison operations.

The core idea of the new methodology is associating a number of single-writer multireader shared pointers, called *hazard pointers*, with each thread that may operate on the associated objects. The number of hazard pointers per thread depends on the algorithms for associated objects and may vary among threads depending on the types of objects they intend to access. Typically, this number is one or two. For simplicity of presentation, we assume that each thread has the same number K of hazard pointers.

The methodology communicates with the associated algorithms only through hazard pointers and a procedure *RetireNode* that is called by threads to pass the addresses of retired nodes. The methodology consists of two main parts: the algorithm for processing retired nodes, and the condition that lock-free algorithms must satisfy in order to guarantee the safety of memory reclamation and ABA prevention.

3.1 The Algorithm

Fig. 1 shows the shared and private structures used by the algorithm. The main shared structure is the list of hazard pointer (HP) records. The list is initialized to contain one HP record for each of the N participating threads. The total number of hazard pointers is $H = NK$.¹ Each thread uses two static private variables, *rlist* (retired list) and *rcount* (retired count), to maintain a private list of retired nodes.

Fig. 2 shows the *RetireNode* routine, where the retired node is inserted into the thread's list of retired nodes and the length of the list is updated. Whenever the size of a thread's list of retired nodes reaches a threshold R , the thread scans the list of hazard pointers using the *Scan* routine. R can be chosen arbitrarily. However, in order to achieve a constant expected amortized processing time per retired node, R must satisfy $R = H + \Omega(H)$.

Fig. 3 shows the *Scan* routine. A scan consists of two stages. The first stage involves scanning the HP list for nonnull values. Whenever a nonnull value is encountered, it is inserted in a local list *plist*, which can be implemented as a hash table. The second stage of *Scan* involves checking each node in *rlist* against the pointers in *plist*. If the lookup yields no match, the node is identified to be ready for arbitrary reuse. Otherwise, it is retained in *rlist* until the

¹ As discussed in Section 3.2, the algorithm can be extended such that the values of N and H do not need to be known in advance, and threads can join and leave the system dynamically and allocate and deallocate hazard pointers dynamically.

```

Scan(head:*HPRecType) {
    // Stage 1: Scan HP list and insert non-null values in plist
    plist.init();
    hprec ← head;
    while (hprec ≠ null) {
        for (i ← 0 to K-1) {
            hptr ← hprec.^HP[i];
            if (hptr ≠ null)
                plist.insert(hptr);
        }
        hprec ← hprec.^Next;
    }
}

// Stage 2: Search plist
tmpList ← rlist.popAll();
rcount ← 0;
node ← tmpList.pop();
while (node ≠ null) {
    if (plist.lookup(node)) {
        rlist.push(node);
        rcount++;
    } else {
        PrepareForReuse(node);
    }
    node ← tmpList.pop();
}
plist.free();
}

```

Fig. 3. The Scan routine.

next scan by the current thread. Insertion and lookup in *plist* take constant expected time.

Alternatively, if a lower worst-case—instead of average—time complexity is desired, *plist* can be implemented as a balanced search tree with $O(\log p)$ insertion and lookup time complexities, where p is the number of nonnull hazard pointers encountered in Stage 1 of *Scan*. In such a case, the amortized time complexity per retired node is $O(\log p)$.

In practice, for simplicity and speed, we recommend implementing *plist* as an array and sorting it at the end of Stage 1 of *Scan*, and then using binary search in Stage 2. We use the latter implementation for our experiments in Section 5. We omit the algorithms for hash tables, balanced search trees, sorting, and binary search, as they are well-known sequential algorithms [2].

The task of the memory reclamation method in the context of this paper is to determine *when* a retired node is eligible for reuse safely while allowing memory reclamation. Thus, the definition of the *PrepareForReuse* routine is open for several implementation options and is not an integral part of this methodology. An obvious implementation of that routine is to reclaim the node immediately for arbitrary reuse using the standard library call for memory deallocation, e.g., *free*. Another possibility—in order to reduce the overhead of calling *malloc* and *free* for every node allocation and deallocation—is that each thread can maintain a limited size private list of free nodes. When a thread runs out of private free nodes, it allocates new nodes, and when it accumulates too many private free nodes, it deallocates the excess nodes.

The algorithm is wait-free; it takes $O(R)$ expected time—or $O(R \log p)$ worst-case time if a logarithmic search structure is used—to identify $\Theta(R)$ retired nodes as eligible for arbitrary reuse. It only uses single-word reads and writes. It offers an upper bound NR on the number of retired nodes that are not yet eligible for reuse, even if some or all threads are delayed or have crashed.

3.2 Algorithm Extensions

The following are optional extensions to the core algorithm that enhance the methodology's flexibility.

If the maximum number N of participating threads is not known before hand, we can add new HP records to the HP list using a simple push routine [11]. Note that such a routine is wait-free, as the maximum number of threads is

finite. This can be useful also, if it is desirable for threads to be able to allocate additional hazard pointers dynamically.

In some applications, threads are created and retired dynamically. In such cases, it is desirable to allow HP records to be reused. Adding a Boolean flag to each HP record can serve as an indicator if the HP record is in use or available for reuse. Before retiring, a thread can clear the flag, and when a new thread is created, it can search the HP list for an available HP record and acquire it using test-and-set (TAS). If no HP records are available, a new one can be added as described above.

Since a thread may have leftover retired nodes not yet identified as eligible for reuse, two fields can be added to the HP record structure so that a retiring thread can pass the values of its *rlist* and *rcount* variables to the next thread that inherits the HP record.

Furthermore, it may be desirable to guarantee that every node that is eligible for reuse is eventually freed, barring thread failures. To do so, after executing *Scan*, a thread executes a *HelpScan*, where it checks every HP record. If an HP record is inactive, the thread locks it using TAS and pops nodes from its *rlist*. Whenever, the thread accumulates R nodes, it performs a *Scan*. Therefore, even if a thread retires leaving behind an HP record with a nonempty *rlist* and its HP record happens not to be reused, the nodes in the *rlist* will still be processed by other threads performing *HelpScan*.

Fig. 4 shows a version of the algorithm that incorporates the above mentioned extensions. The algorithm is still wait-free, and only single-word instructions are used.

3.3 The Condition

For a correct algorithm for a dynamic lock-free object to use the new methodology for memory reclamation and ABA prevention, it must satisfy a certain condition. When a thread assigns a reference (i.e., a node's address) to one of its hazard pointers, it basically announces to other threads that it may use that reference in a hazardous manner (e.g., access the contents of the node without further validation of the reference), so that other threads will refrain from reclaiming or reusing the node until the reference is no longer hazardous. This announcement (i.e., setting the hazard pointer) must take place before the node is retired and the hazard pointer must continue to hold that reference until the reference is no longer hazardous.

```

structure HPRecType { HP[K]:*NodeType; Next:*HPRecType;
    Active:Boolean; rlist:listType; rcount:integer; }
// Shared variables
HeadHPRec : *HPRecType; // initially null
H : integer; // initially 0
// Per-thread private variable
myhprec : *HPRecType; // initially null

AllocateHPRec() {
    // First try to reuse a retired HP record
    for (hprec ← HeadHPRec; hprec ≠ null; hprec ← hprec^.Next) {
        if (hprec^.Active) continue;
        // TAS(addr) ≡ ¬CAS(addr,false,true)
        if TAS(&hprec^.Active) continue;
        // Succeeded in locking an inactive HP record
        myhprec ← hprec;
        return;
    }
    // No HP records available for reuse
    // Increment H, then allocate a new HP and push it
    do { // wait-free - max. num. of threads is finite
        oldcount ← H;
    } until CAS(&H,oldcount,oldcount+K);
    // Allocate and push a new HP record
    hprec ← NewHPRec();
    Initialize the fields of the new HP record.
    do { // wait-free - max. num. of threads is finite
        oldhead ← HeadHPRec;
        hprec^.Next ← oldhead;
    } until CAS(&HeadHPRec,oldhead,hprec);
    myhprec ← hprec;
}

RetireHPRec() {
    for (i ← 0 to K-1) myhprec^.HP[i] ← null;
    myhprec^.Active ← false;
}

RetireNode(node:*NodeType) {
    myhprec^.rlist.push(node);
    myhprec^.rcount++;
    head ← HeadHPRec;
    if (myhprec^.rcount ≥ R(H)) { // R(H)=H + Ω(H)
        Scan(head);
        HelpScan();
    }
}

// Scan() is the same as in Figure 3 except that rlist and rcount
// are fields of *myhprec instead of being private variables.

HelpScan() {
    for (hprec ← HeadHPRec; hprec ≠ null; hprec ← hprec^.Next) {
        if (hprec^.Active) continue;
        if TAS(&hprec^.Active) continue;
        while (hprec^.rcount > 0) {
            node ← hprec^.rlist.pop();
            hprec^.rcount--;
            myhprec^.rlist.push(node);
            myhprec^.rcount++;
            head ← HeadHPRec;
            if (myhprec^.rcount ≥ R(H))
                Scan(head);
        }
        hprec^.Active ← false;
    }
}

```

Fig. 4. Algorithm extensions.

For a formal description of the condition, we first define some terms:

Node: We use the term *node* to describe a range of memory locations that at some time may be viewed as a logical entity either through its actual use in an object that uses hazard pointers, or from the point of view of a participating thread. Thus, it is possible for multiple nodes to overlap physically, but still be viewed as distinct logical entities.

At any time t , each *node n* is in one of the following states:

1. *Allocated*: n is allocated by a participating thread, but not yet inserted in an associated object.
2. *Reachable*: n is reachable by following valid pointers starting from the roots of an associated object.
3. *Removed*: n is no longer reachable, but may still be in use by the removing thread.
4. *Retired*: n is already removed and consumed by the removing thread, but not yet free.
5. *Free*: n 's memory is available for allocation.
6. *Unavailable*: all or part of n 's memory is used by an unrelated object.
7. *Undefined*: n 's range of memory locations is not currently viewed as a node.

Own: A thread j *owns* a node n at time t , iff at t , n is *allocated*, *removed*, or *retired* by j . Each node can have at most

one *owner*. The *owner* of an *allocated* node is the thread that allocated it (e.g., by calling `malloc`). The *owner* of a *removed* node is the thread that executed the step that removed it from the object (i.e., changed its state from *reachable* to *removed*). The *owner* of a *retired* node is the same one that removed it.

Safe: A node n is *safe* for a thread j at time t , iff at time t , either n is *reachable*, or j *owns* n .

Possibly unsafe: A node is *possibly unsafe* at time t from the point of view of thread j , if it is impossible solely by examining j 's private variables and the semantics of the algorithm to determine definitely in the affirmative that at time t the node is *safe* for j .

Access hazard: A step s in thread j 's algorithm is an *access hazard* iff it may result in access to a node that is *possibly unsafe* for j at the time of its execution.

ABA hazard: A step s in thread j 's algorithm is an *ABA hazard* iff it includes an ABA-prone comparison that involves a dynamic node that is *possibly unsafe* for j at the time of the execution of s , such that either 1) the node's address—or an arithmetic variation of it—is an expected value of the ABA-prone comparison, or 2) a memory location contained in the dynamic node is the target of the ABA-prone comparison.

Access-hazardous reference: A thread j holds an *access-hazardous reference* to a node n at time t , iff at time t one or more of j 's private variables holds n 's address or an

arithmetic variation of it, and j is guaranteed—unless it crashes—to reach an *access hazard* s that uses n 's address hazardously, i.e., accesses n when n is possibly unsafe for j .

ABA-hazardous reference: A thread j holds an *ABA-hazardous reference* to a node n at time t , iff at time t , one or more of j 's private variables holds n 's address or a mathematical variation of it, and j is guaranteed—unless it crashes—to reach an *ABA hazard* s that uses n 's address hazardously.

Hazardous reference: A reference is *hazardous* if it is *access-hazardous* and/or *ABA-hazardous*.

Informally, a hazardous reference is an address that without further validation of safety will be used later in a hazardous manner, i.e., to access possibly unsafe memory and/or as a target address or an expected value of an ABA-prone comparison.

A thread that holds a reference to a node uses hazard pointers to announce to other threads that it may use the reference later without further validation in a hazardous step. However, this announcement is useless if it happens after the reference is already *hazardous*, or in other words, after the node is *possibly unsafe*, since another thread might have already removed the node, and then scanned the HP list and found no match for that node. Therefore, the condition that an associated algorithm must satisfy is that whenever a thread is holding a hazardous reference to a node, it must be the case that at least one of the thread's hazard pointers has been continuously holding that reference from a time when the node was definitely *safe* for the thread. Note that this condition implies that no thread can create a new hazardous reference to a node while it is retired.

Formally, the condition is as follows, where HP_j is the set of thread j 's hazard pointers:

$$\begin{aligned} \forall \text{ times } t, \text{ threads } j, \text{ and nodes } n, \\ (\text{at } t, j \text{ holds a hazardous reference to } n) \Rightarrow \\ (\exists hp \in HP_j, t' \leq t :: \\ (\text{at } t', n \text{ is safe for } j) \wedge \\ (\forall \text{ times during } [t', t], hp = \&n)). \end{aligned}$$

3.4 Correctness

The following lemmas and theorem are contingent on satisfying the condition in Section 3.3.

Lemma 1. $\forall \text{ times } t, \text{ threads } j, \text{ and nodes } n, (\forall hp \in HP_j, t' \leq t, (\forall \text{ times during } [t', t], n \text{ is not safe for } j) \wedge (\exists t'' \in [t', t] :: \text{at } t'', hp \neq \&n)) \Rightarrow (\text{at } t, j \text{ does not hold a hazardous reference to } n)$.

Informally, if a scan of the hazard pointers of a thread j finds no match for a retired node n , then it must be the case that j holds no hazardous reference to n at the end of the scan.

Proof sketch: For a proof by contradiction, assume that the lemma is false, i.e., the antecedent of the implication is true and the consequent is false. Then, at t , j holds a hazardous reference to n . Then, by the condition in Section 3.3, there must be some time t_0 when n was safe for j , but t_0 must be before t' because we already assume that n is not safe for j during $[t_0, t]$. Also by the condition in Section 3.3, there must be at least one of j 's hazard pointers that is pointing to n continuously during $[t_0, t]$. But, this contradicts the initial assumption that for each of j 's hazard pointers, there is some time during $[t', t]$

(and, hence, during $[t_0, t]$) when the hazard pointer does not point to n . Therefore, the initial assumption must be false and the lemma is true. \square

Lemma 2. $\forall \text{ times } t, \text{ threads } j, \text{ and nodes } n, (\text{at } t, n \text{ is identified in stage 2 of Scan as eligible for reuse}) \Rightarrow (\forall hp \in HP_j, \exists t' \in [t_0, t] :: \text{at } t', hp \neq \&n)$, where t_0 is the start time of the current execution of Scan.

Informally, a retired node is identified as eligible for reuse in stage 2 of *Scan* only after a scan of the hazard pointers of participating threads finds no match.

Proof sketch: For a proof by contradiction, assume that the lemma is false. Then, at least one of j 's hazard pointers was continuously pointing to n since t_0 . Then, by the flow of control (of stage 1), at the end of stage 1, *plist* must contain a pointer to n . Then, by the flow of control (of stage 2), n is not identified as eligible for reuse. A contradiction to the initial assumption. \square

Theorem 1. $\forall \text{ times } t, \text{ threads } j, \text{ and nodes } n, (\text{at } t, n \text{ is identified in stage 2 of Scan as eligible for reuse}) \Rightarrow (\text{at } t, j \text{ does not hold a hazardous reference to } n)$.

Informally, if *Scan* identifies a node as eligible for reuse, then it must be the case that no thread holds a hazardous reference to it.

Proof sketch: If j is the thread executing *Scan*, the theorem is trivially true. Consider the case where j is a different thread. Assume that at t , n is identified in stage 2 of *Scan* as eligible for reuse. Then, by the definition of *safe*, n is not safe for j since the beginning of *Scan*, and by Lemma 2, for each hazard pointer, there was a time during the current execution of *Scan* when the hazard pointer did not point to n . Then, by Lemma 1, at t , j does not hold a hazardous reference to n . \square

By the definition of *access-hazardous reference* and Theorem 1, it follows that the hazard pointer methodology (i.e., algorithm and condition) guarantees that while a node is free or unavailable, no thread accesses its contents, i.e., the hazard pointer methodology guarantees safe memory reclamation.

By the definition of *ABA-hazardous reference* and Theorem 1, it follows that the hazard pointer methodology guarantees that, while a node is free or unavailable, no thread can hold a reference to it without further validation with the intention to use that reference as a target or an expected value of an ABA-prone comparison. This is the same guarantee offered by GC with respect to the ABA problem.

4 APPLYING HAZARD POINTERS

This section discusses the methodology for adapting existing lock-free algorithms to the condition in Section 3.3. The following is an outline:

1. Examine the target algorithm as follows:
 - a. Identify *hazards* and the *hazardous references* they use.
 - b. For each distinct *hazardous reference*, determine the point where it is created and the last *hazard* that uses it. The period between these two

```

structure NodeType { Data:DataType; Next:*NodeType; }
// Shared variables
Head,Tail:*NodeType;
// Initially both Head and Tail point to a dummy node

Enqueue(data:DataType) {
1: node ← NewNode();
2: node^.Data ← data;
3: node^.Next ← null;
while true {
4:   t ← Tail;
5:   next ← t^.Next;
6:   if (Tail ≠ t) continue;
7:   if (next ≠ null) { CAS(&Tail,t,next); continue; }
8:   if CAS(&t^.Next,null,node) break;
}
9: CAS(&Tail,t,node);
}

```

```

Dequeue() : DataType {
while true {
11:   h ← Head;
12:   t ← Tail;
13:   next ← h^.Next;
14:   if (Head ≠ h) continue;
15:   if (next = null) return EMPTY;
16:   if (h = t) { CAS(&Tail,t,next); continue; }
17:   data ← next^.Data;
18:   if CAS(&Head,h,next) break;
}
19: return data;
}

```

Fig. 5. A memory-management-oblivious lock-free queue algorithm.

points is when a hazard pointer needs to be dedicated to that reference.

- c. Compare the periods determined in the previous step for all *hazardous references*, and determine the maximum number of distinct references that can be hazardous—for the same thread—at the same time. This is the maximum number of hazard pointers needed per thread.
2. For each hazardous reference, insert the following steps in the target algorithm after the creation of the reference and before any of the hazards that use it:
 - a. Write the address of the node that is the target of the reference to an available hazard pointer.
 - b. Validate that the node is *safe*. If the validation succeeds, follow the normal flow of control of the target algorithm. Otherwise, skip over the hazards and follow the path of the target algorithm when conflict is detected, i.e., try again, backoff, exit loop, etc. This step is needed, as the node might have been already removed before the previous step was executed.

We applied hazard pointers to many algorithms, e.g., [11], [9], [23], [28], [21], [26], [4], [16], [7], [6], [18], for the purposes of allowing memory reclamation and ABA prevention. We use several algorithms for important object types to demonstrate the application of the steps described above.

Note that, while applying these steps is easy for algorithm designers, these steps are not readily applicable automatically (e.g., by a compiler). For example, it is not clear if a compiler can determine if a node is no longer *reachable*. Identifying ABA hazards is even more challenging for a compiler, as the ABA problem is a subtle problem that involves the implicit intentions of the algorithm designer.

4.1 FIFO Queues

Fig. 5 shows a version of Michael and Scott’s [21] lock-free FIFO queue algorithm, stripped of memory management code. The algorithm demonstrates typical use of hazard pointers. We use it as the main case study.

Briefly, the algorithm represents the queue as a singly linked list with a dummy node at its head. If an enqueue operation finds the *Tail* pointer pointing to the last node, it

links the new node at the end of the list and then updates *Tail*. Otherwise, it updates *Tail* first, and then tries to enqueue the new node. A dequeue operation swings the *Head* pointer after reading the data from the second node in the list, while ensuring that *Tail* will not lag behind *Head*. As it is the case with any lock-free object, if a thread is delayed at any point while operating on the object and leaves it in an unstable state, any other thread can take the object to a stable state and then proceed with its own operation.

First, we examine Fig. 5 to identify *hazards* and *hazardous references*, starting with the enqueue routine:

1. The node accesses in lines 2 and 3 are not hazardous because, at that time, the node **node* is guaranteed to be *allocated*—and, hence, *owned*—by the current thread.
2. The node access in line 5 is an *access hazard* because the node **t* may have been removed and reclaimed by another thread after the current thread executed line 4.
3. The function of what we call a *validation condition* in line 6 is to guarantee that the thread proceeds to line 7, only if at the time of reading *t^.Next* in line 5, *Tail* was equal to *t*. Without this guarantee, the queue may be corrupted. It is ABA-prone and, hence, it is an *ABA hazard*.
4. The CAS in line 7 is an *ABA hazard*.
5. The CAS in line 8 is both an *access hazard* and an *ABA hazard*.
6. The CAS in line 9 is an *ABA hazard*.

Therefore, the reference to *t* is hazardous between lines 4 and 9. Only one hazard pointer is sufficient since there is only one hazardous reference at any time in this routine.

Fig. 6 shows the enqueue routine augmented with a hazard pointer and code guaranteeing safe memory reclamation and ABA prevention. The technique in lines 4a and 4b is the fundamental mechanism for applying hazard pointers to target algorithm, as shown in the rest of this section.

After creating a reference (line 4) that is identified as hazardous in the memory-management-oblivious algorithm, the thread takes the following steps: 1) It assigns the address of the referenced node to a hazard pointer

//**hp0** and **hp1** are private ptrs to 2 of the thread's hazard ptrs

```

Enqueue(data:DataType) {
  1: node ← NewNode();
  2: node^.Data ← data;
  3: node^.Next ← null;
  while true {
    4:   t ← Tail;
    4a:   *hp0 ← t;
    4b:   if (Tail ≠ t) continue;
    5:   next ← t^.Next;
    6:   if (Tail ≠ t) continue;
    7:   if (next ≠ null) { CAS(&Tail,t,next); continue; }
    8:   if CAS(&t^.Next,null,node) break;
  }
  9: CAS(&Tail,t,node);
}

```

Fig. 6. Enqueue routine with hazard pointers.

(line 4a). 2) Then, it validates that that node is safe (line 4b). If not, it skips over the hazards and tries again.

The second step is needed, since it is possible that after line 4 and before line 4a, some other thread had removed the node $*t$ and checked $*hp0$ and concluded that the current thread does not hold hazardous references to $*t$. Line 4b serves to guarantee that at that point, which is before any hazards, $*hp0$ already points to $*t$, and that $*t$ is safe.

It is possible that the node $*t$ was removed and then reinserted by other threads between the current thread's execution of lines 4 and 4b. However, this is acceptable, as it does not violate the condition in Section 3.3. After executing line 4 and before line 4b, the reference t is not hazardous as the thread is not guaranteed to reach a hazard (lines 5, 6, 7, 8, or 9). It starts to be hazardous only upon the success of the validation condition in line 4b. But, at that point, $*hp0$ already covers t and continues to do so until t ceases to be hazardous (after line 9).

Next, we examine the dequeue routine in Fig. 5:

1. The node access in lines 13 is an *access hazard* using the reference h .
2. The validation condition in line 14 is an *ABA hazard* using h .
3. The CAS in line 16 is an *ABA hazard* using t . However, since t is guaranteed to be equal to h at that point, then it is covered if h is covered.
4. The node access in line 17 is an *access hazard* using $next$.
5. The CAS in line 18 is an *ABA hazard* using h .

Therefore, h is hazardous between lines 11 and 18, and $next$ is hazardous between lines 13 and 17. Since the two periods overlap, two hazard pointers are needed.

Fig. 7 shows the dequeue routine augmented with hazard pointers. For the reference h , lines 11a and 11b employ the same technique as lines 4a and 4b in Fig. 6. For the reference $next$, no extra validation is needed. The validation condition in line 14 (from the original algorithm) guarantees that $*hp1$ equals $next$ from a point when $*next$ was safe. The semantics of the algorithm guarantee that the node $*next$ cannot be *removed* at line 14 unless its predecessor $*h$ has been removed first and then reinserted after line 13 and before line 14. But, this is impossible since $*hp0$ covers h from before line 11b until after line 18.

```

Dequeue() : DataType {
  while true {
    11:   h ← Head;
    11a:   *hp0 ← h;
    11b:   if (Head ≠ h) continue;
    12:   t ← Tail;
    13:   next ← h^.Next;
    13a:   *hp1 ← next;
    14:   if (Head ≠ h) continue;
    15:   if (next = null) return EMPTY;
    16:   if (h = t) { CAS(&Tail,t,next); continue; }
    17:   data ← next^.Data;
    18:   if CAS(&Head,h,next) break;
  }
  19: RetireNode(h); return data;
}

```

Fig. 7. Dequeue routine with hazard pointers.

Note that hazard pointers allow some optimizations to the original algorithm [21] that are not safe when only the IBM tag method [11] is used for ABA prevention. Line 6 can be removed from the enqueue routine and line 17 can be moved out of the main loop in the dequeue routine.

If only safe memory reclamation is desired—and the ABA problem is not a concern, e.g., assuming support for ideal LL/SC/VL—then only one hazard pointer is sufficient for protecting both $*h$ and $*next$ in the *Dequeue* routine. We leave this as an exercise for the reader.

4.2 LIFO Stacks

Fig. 8 shows a lock-free stack based on the IBM freelist algorithm [11] augmented with hazard pointers. In the push routine, the node accesses in lines 2 and 4 are not *hazardous* (as $*node$ is *owned* by the thread at the time), and the CAS in line 5 is not ABA-prone (as the change of Top between lines 3 and 5 can never lead to corrupting the stack or any other object). Thus, no hazard pointers are needed for the push routine.

In the pop routine, the node access in line 10 is hazardous and the CAS in line 11 is ABA-prone. All hazards use the reference t . The technique employed in transforming the pop routine is the same as that used in the enqueue routine of Fig. 6.

4.3 List-Based Sets and Hash Tables

Fig. 9 shows an improved version of the lock-free list-based set implementation in [16], using hazard pointers. The algorithm in [16] improves on that of Harris [5] by allowing efficient memory management. It can be used as a building block for implementing lock-free chaining hash tables.

Node insertion is straightforward. Deletion of a node involves first marking the low bit of its *Next* pointer and then removing it from the list, to prevent other threads from linking newly inserted nodes to removed nodes [23], [5]. Whenever a traversing thread encounters a node marked for deletion, it removes the node before proceeding, to avoid creating references to nodes after their removal.

The traversal of the list requires the protection of at most two nodes at any time: a current node $*cur$, if any, and its predecessor, if any. Two hazard pointers, $*hp0$ and $*hp1$, are used for protecting these two nodes, respectively. The traversing thread starts the main loop (lines 12–25) with $*prev$ already protected. In the first iteration, $*prev$ is the root and, hence, it is safe. The hazard pointer $*hp0$ is set by

```

structure NodeType { Data:DataType; Next:*NodeType; }
// Shared variables
Top:*NodeType; // Initially null
// hp is a private ptr to one of the thread's hazard ptrs.

Push(data:DataType) {
1: node ← NewNode();
2: node^.Data ← data;
while true {
3: t ← Top;
4: node^.Next ← t;
5: if CAS(&Top,t,node) return;
}
}

Pop() : DataType {
while true {
6: t ← Top;
7: if (t = null) return EMPTY;
8: *hp ← t;
9: if (Top ≠ t) continue;
10: next ← t^.Next;
11: if CAS(&Top,t,next) break;
}
12: data ← t^.Data;
13: RetireNode(t); return data;
}

```

Fig. 8. Lock-free stack using hazard pointers.

assigning it the hazardous reference *cur* (line 13) and then validating that the node **cur* is in the list at a point subsequent to the setting of the hazard pointer. The validation is done by validating that the pointer **prev* still has the value *cur*. This validation suffices as the semantics of the algorithm guarantee that the value of **prev* must change, if either the node that includes **prev* or its successor, the node **cur*, is removed. The protection of **prev* itself is guaranteed, either by being the root (in the first iteration), or by guaranteeing (as described below) that the node that contains it is covered by **hp1*.

For protecting the pointer **prev* in subsequent iterations, as *prev* takes an arithmetic variation of the value of *cur* (line 23), the algorithm exchanges the private labels of **hp0* and **hp1* (line 24). There are no windows of vulnerability since the steps after line 21 to the end of the loop do not

contain any hazards. Also, since **hp0* already protects the reference *cur* at line 11a, then there is no need for further validation that the node formerly pointed to by *cur* and now contains **prev* is in the list.

Therefore, all the hazards (lines 4, 6, 7, 15, 17, 20, and 21) are covered by hazard pointers according to the condition in Section 3.3.

This algorithm demonstrates an interesting case where hazard pointers are used to protect nodes that are not adjacent to the roots of the object, unlike the cases of the queue and stack algorithms.

4.4 Single-Writer Multireader Dynamic Structures

Tang et al. [27] used lock-free single-writer multireader doubly-linked lists for implementing point-to-point send and receive queues, to improve the performance of threaded MPI

```

structure NodeType { Key:KeyType; Next:*NodeType; }
// Per-thread private variables
prev: **NodeType; cur,next: *NodeType;
// hp0 and hp1 are private ptrs to 2 of the thread's hazard ptrs.
// Integer arithmetic in lines 6, 17, and 19.

Insert(head:**NodeType,node:*NodeType):Boolean {
1: key ← node^.Key;
while true {
2: if Find(head,key) return false;
3: node^.Next ← cur;
4: if CAS(prev,cur,node) return true;
}
}

Delete(head:**NodeType,key:KeyType):Boolean {
while true {
5: if ¬Find(head,key) return false;
6: if ¬CAS(&cur^.Next,next,next+1) continue;
7: if CAS(prev,cur,next) RetireNode(cur); else Find(head,key);
8: return true;
}
}

Search(head:**NodeType,key:KeyType):Boolean {
9: return Find(head,key);
}
}

Find(head:**NodeType;key:KeyType) : Boolean {
try_again:
10: prev ← head;
11: cur ← *prev;
12: while (cur ≠ null) {
13: *hp0 ← cur;
14: if (*prev ≠ cur) goto try_again;
15: next ← cur^.Next;
16: if (next & 1) { // bitwise AND
17: if ¬CAS(prev,cur,next-1) goto try_again;
18: RetireNode(cur);
19: cur ← next-1;
} else {
20: ckey ← cur^.Key;
21: if (*prev ≠ cur) goto try_again;
22: if (ckey ≥ key) return (ckey = key);
23: prev ← &cur^.Next;
24: tmp ← hp0; hp0 ← hp1; hp1 ← tmp; // all private
cur ← next;
}
}
26: return false;
}

```

Fig. 9. Lock-free list-based set with hazard pointers.

```

structure NodeType { Key:KeyType; Data:DataType;
  Prev:**NodeType; Next:*NodeType; };
//hp0 and hp1 are private ptrs to 2 of the thread's hazard ptrs.

SingleWriterInsertAfter(prev:**NodeType,node:*NodeType) {
1:  next ← *prev;
2:  node^.Prev ← prev;
3:  node^.Next ← next;
4:  if (next ≠ null) next^.Prev ← &node^.Next;
5:  *prev ← node; // Inserted
}

SingleWriterDelete(node:*NodeType) {
6:  prev ← node^.Prev;
7:  next ← node^.Next;
8:  if (next ≠ null) next^.Prev ← prev;
9:  *prev ← next; // Deleted
10: node^.Next ← null; // To alert readers not to proceed
11: RetireNode(node);
}

```

```

ReaderSearch(head:**NodeType,key:KeyType) : DataType {
try_again:
12:  prev ← head;
13:  cur ← *prev;
14:  while (cur ≠ null) {
15:    *hp0 ← cur;
16:    if (*prev ≠ cur) goto try_again;
17:    next ← cur^.Next;
18:    ckey ← cur^.Key;
19:    if (cur^.Key = key) {
20:      data ← cur^.Data;
21:      if (*prev ≠ cur) goto try_again;
22:      return data;
    }
23:    if (*prev ≠ cur) goto try_again;
24:    prev ← &cur^.Next;
25:    tmp ← hp0; hp0 ← hp1; hp1 ← tmp;
26:    cur ← next;
  }
27:  return NOTFOUND;
}

```

Fig. 10. Lock-free single-writer multiple-reader doubly-linked list with hazard pointers.

on multiprogrammed shared memory systems. The main challenge for that implementation was memory management. That is, how does the owner (i.e., the single writer) guarantee that no readers still hold references to a removed node before reusing or freeing it? In order to avoid inefficient per-node reference counting, they use instead aggregate (per-list) reference counting. However, the delay or failure of a reader thread can prevent the owner indefinitely from reusing an unbounded number of removed nodes. As discussed in Section 6.1, this type of aggregate methods is not lock-free as it is sensitive to thread delays and failures. Furthermore, even without any thread delays, in that implementation, if readers keep accessing the list, the aggregate reference counter may never reach zero, and the owner remains indefinitely unable to reuse removed nodes. Also, the manipulation of the reference counters requires atomic operations such as CAS or atomic add.

We use hazard pointers to provide an efficient single-writer multireader doubly-linked list implementation that allows lock-free memory reclamation. Fig. 10 shows the code for the readers' search routine, and the owner's insertion and deletion routines. Unlike using locks or reference counting, reader threads do not write to any shared variables other than their own hazard pointers, which are rarely accessed by the writer thread, thus decreasing cache coherence traffic. Also, instead of a quadratic number of per-list locks or reference counters, only two hazard pointers are needed per reader thread for supporting an arbitrary number of lists.

The algorithm uses only single-word reads and writes for memory access. Thus, it demonstrates the importance of the feasibility of the hazard pointer methodology on systems with hardware support for memory access limited to these instructions.

The algorithm is mostly straightforward. However, it is worth noting that changing `node^.Next` in line 10 in the `SingleWriterDelete` routine is necessary if `*node` is not the last node in the list. Otherwise, it is possible that the validation condition in line 23 of a concurrent execution of the `ReaderSearch` routine by a reader thread may succeed even

after the owner has retired the node containing `*prev`. If so, and if the owner also removes the following node, the reader might continue to set a hazard pointer for the following node, but too late after it has already been removed and reused.

5 EXPERIMENTAL PERFORMANCE RESULTS

This section presents experimental performance results comparing the performance of the new methodology to that of other lock-free memory management methods. We implemented lock-free algorithms for FIFO queues [21], LIFO stacks [11], and chaining hash tables [16], using hazard pointers, ABA-prevention tags [11], and lock-free reference counting [29]. Details of the latter memory management methods are discussed in Section 6.1.

Also, we included in the experiments efficient commonly-used lock-based implementations of these object types. For FIFO queues and LIFO stacks, we used the ubiquitous test-and-test-and-set [24] lock with bounded exponential backoff. For the hash tables, we used implementations with 100 separate locks protecting 100 disjoint groups of buckets, thus allowing complete concurrency between operations on different bucket groups. For these locks, we used Mellor-Crummey and Scott's [15] simple fair reader-writer lock to allow intrabucket group concurrency among read-only operations.

Experiments were performed on an IBM RS/6000 multiprocessor with four 375 MHz POWER3-II processors. We ran the experiments when no other users used the system. Data structures of all implementations were aligned to cache line boundaries and padded where appropriate to eliminate false sharing. In all experiments, all needed memory fit in physical memory. Fence instructions were inserted in the code of all implementations wherever memory ordering is required. Locks and single-word and double-word CAS were implemented using the single-word and double-word LL/SC instructions supported on the POWER3 architecture in 32-bit mode. All implementations were compiled at the highest optimization level.

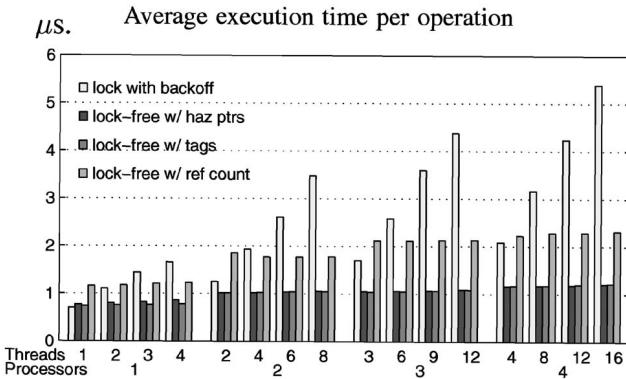


Fig. 11. Performance of FIFO queues.

We ran each experiment five times and reported the average of the median three. Variance was negligible in all experiments. Reported times exclude initialization. For each implementation, we varied the number of processors used from one to four, and the number of threads per processor from one to four. At initialization, each thread was bound to a specific processor. All needed nodes were allocated at initialization. During time measurement, removed nodes were made ready for reuse, but were not deallocated. The pseudorandom sequences for generating keys and operations for different threads were nonoverlapping in each experiment, but were repeatable in every experiment for fairness in comparing different implementations.

For the hazard pointer implementations, we set the number of threads N conservatively to 64, although smaller numbers of threads were used. For implementations with reference counting, we carefully took into account the semantics of the target algorithms to minimize the number of updates to reference counters.

Figs. 11 and 12 show the average execution time per operation on shared FIFO queue and LIFO stack implementations, respectively. In each experiment, each thread executed 1,000,000 operations, i.e., enqueue and dequeue operations for queues and push and pop operations for stacks. The abbreviated label “**haz ptrs**” refers to hazard pointers, “**tags**” refers to ABA-prevention tags, and “**ref count**” refers to lock-free reference counting.

Figs. 13 and 14 show the average CPU time (product of execution time and the number of processors used) per operation on shared chaining hash tables with 100 buckets and load factors of 1 and 5, respectively. The load factor of a

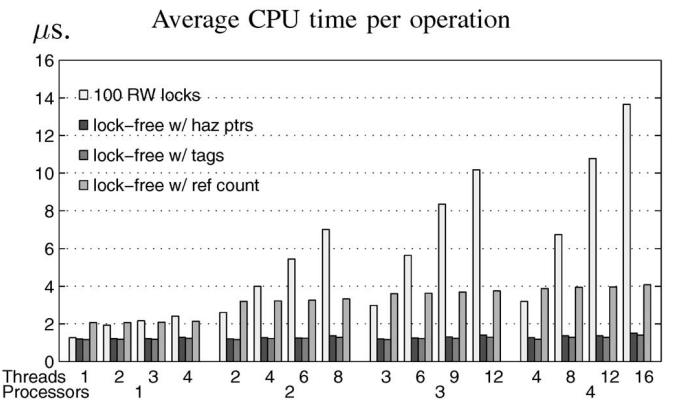


Fig. 13. Performance of hash tables with load factor 1.

hash table is the average number of items per bucket. In each experiment, each thread executed 2,000,000 operations (Insert, Delete, and Search).

A common observation in all the graphs is that lock-free implementations with hazard pointers perform as well as and often significantly better than the other implementations in virtually all cases. Being lock-free, the performance of lock-free objects is unaffected by preemption, while locks perform poorly under preemption. For example, the lock-free hash table with hazard pointers achieves throughput that is 251 percent, 496 percent, 792 percent, and 905 percent that of the lock-based implementation, with 4, 8, 12, and 16 threads, respectively, running on four processors (Fig. 13).

In all experiments, the performance of hazard pointers is comparable to that of ABA-prevention tags. However, unlike hazard pointers, tags require double-width instructions and hinder rather than assist memory reclamation.

Lock-free objects with hazard pointers handle contention (Figs. 11 and 12) significantly better than locks even in the absence of preemption. For example, under contention by four processors, and even without preemption, they achieve throughputs that are 178 percent that of locks when operating on queues. Note that experiments using locks without backoff (not shown) resulted in more than doubling the execution time for locks under contention by four processors. Using backoff in the lock-free implementations with hazard pointers resulted in moderate improvements in performance under contention by four processors (25 percent on queues and 44 percent on stacks). However, we conservatively report the results without backoff.

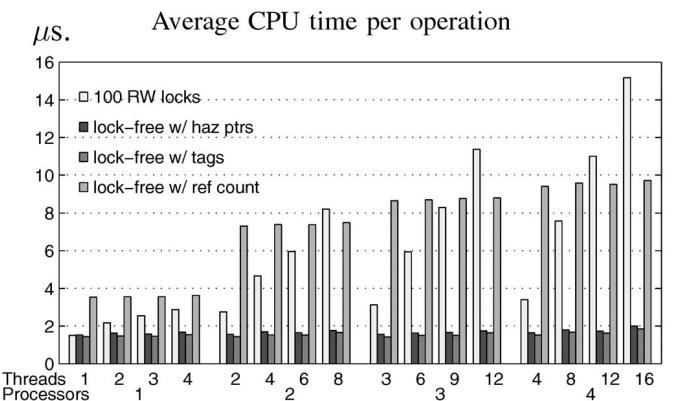


Fig. 14. Performance of hash tables with load factor 5.

Fig. 12. Performance of LIFO stacks.

Lock-free implementations with hazard pointers also outperform lock-based implementations as well as those with reference counting in the case of no or negligible contention, but under sharing, as is the case with hash tables (Figs. 13 and 14). This is because they do not write to any shared locations other than hazard pointers, during read-only Search operation as well as traversal, thus minimizing cache coherence traffic.² On the other hand, lock acquisition and release, even for read-only transactions, result in writes to lock variables. As for reference counting, the situation is even worse, the reference counter of each traversed node needs to be incremented and then decremented, even during read-only transactions. The effect is most evident in the case of hash tables with a load factor of 5 (Fig. 14). The throughput of the implementation using hazard pointers is 573 percent that of the one using reference counting, on four processors.

For hash tables, we ran experiments (not shown) using a single lock for the whole hash table. As expected, the performance of these implementations was extremely poor (more than 10 fold increase in execution time on four processors over the highly concurrent implementations with 100 disjoint locks). We also ran experiments using 100 test-and-test-and-set mutual exclusion locks instead of 100 reader-writer locks. The performance of the former (not shown) is slightly worse than the latter (shown), as they do not allow intrabucket concurrency of read-only operations. Hence, we believe that we have chosen very efficient lock-based implementations as the baseline for evaluating lock-free implementations and their use of hazard pointers.

Also, we conservatively opted to focus on low levels of contention and multiprogramming, that tend to limit the performance advantages of lock-free synchronization. A common criticism of unconventional synchronization techniques is that they achieve their advantages only under uncommonly high levels of contention, and that they often perform very poorly in comparison to simple mutual exclusion in the common case of low contention. Our experimental results show that lock-free objects, with hazard pointers, offer comparable performance to that of the simplest and most efficient lock-based implementations, under no contention and no preemption, in addition to their substantial performance advantages under higher levels of contention and preemption.

The superior performance of lock-free implementations using hazard pointers to that of lock-based implementations is attributed to several factors. Unlike locks,

1. they operate directly on the shared objects without the need for managing additional lock variables,
2. read-only operations do not result in any writes to shared variables (other than the mostly private hazard pointers),
3. there is no useless spinning, as each attempt has a chance of success when it starts, which makes them more tolerant to contention, and
4. progress is guaranteed under preemption.

Note that the effects of the first two factors are applicable even under no contention and no preemption.

² Typically, a write by a processor to a location that is cached in its cache with a read-only permission results in invalidating all cached copies of the location in other processors' caches, and results in additional traffic if the other processors later need to access the same cache line. However, a read to a cached copy with read-only permission does not result in any coherence traffic.

It is worth noting that, while scalable queue-based locks [14], [15] outperform the simple locks we used in this study under high levels of contention, they underperform them in the common case of no or low contention. Furthermore, these locks are extremely sensitive to even low levels of preemption, as the preemption of any thread waiting in the lock queue—not just the lock holder—can result in blocking.

Overcoming the effects of preemption on locks may be partially achieved, but only by using more complicated and expensive techniques such as handshaking, timeout, and communication with the kernel [1], [12], [22], [30] that further degrade the performance of the common case. On the other hand, as shown in the graphs, lock-free implementations with hazard pointers are inherently immune to thread delays, at virtually no loss of performance in the case of no contention and no preemption, and outperform lock-based implementations under preemption and contention.

6 DISCUSSION

6.1 Related Work

6.1.1 IBM ABA-Prevention Tags

The earliest and simplest lock-free method for node reuse is the tag (update counter) method introduced with the documentation of CAS on the IBM System 370 [11]. It requires associating a tag with each location that is the target of ABA-prone comparison operations. By incrementing the tag when the value of the associated location is written, comparison operations (e.g., CAS) can determine if the location was written since it was last accessed by the same thread, thus preventing the ABA problem. The method requires that the tag contains enough bits to make full wraparound impossible during the execution of any single lock-free attempt. This method is very efficient and allows the immediate reuse of retired nodes.

On the downside, when applied to arbitrary pointers as it is the case with dynamic sized objects, it requires double-width instructions to allow the atomic manipulation of the pointer along with its associated tag. These instructions are not supported on 64-bit architectures. Also, in most cases, the semantics of the tag field must be preserved indefinitely. Thus, if the tag is part of a dynamic node structure, these nodes can never be reclaimed. Their memory cannot be divided or coalesced, as this may lead to changing the semantics of the tag fields. That is, once a range of memory locations are used for a certain node type, they cannot be reused for other node types that do not preserve the semantics of the tag fields.

6.1.2 Lock-Free Reference Counting

Valois [29] presented a lock-free reference counting method that requires the inclusion of a reference counter in each dynamic node, reflecting the maximum number of references to that node in the object and the local variables of threads operating on the object. Every time a new reference to a dynamic node is created/destroyed, the reference counter is incremented/decremented, using fetch-and-add and CAS. Only after its reference counter goes to zero, can a node be reused. However, due to the use of single-address CAS to manipulate pointers and independently located reference counters nonatomically, the resulting timing windows dictate the permanent retention of nodes of their

type and reference counter field semantics, thus hindering memory reclamation.

Detlefs et al. [3] presented a lock-free reference counting method that uses the mostly unsupported DCAS primitive (i.e., CAS on two independent locations) to operate on both pointers and reference counters atomically to guarantee that a reference counter is never less than the actual number of references. When the reference counter of a node reaches zero, it becomes safe to reclaim for reuse. However, free memory cannot be returned to the operating system.

The most important disadvantage of per-node reference counting is the prohibitive performance cost due to the—otherwise, unnecessary—updates of the reference counters of referenced nodes, even for read-only access, thus causing significant increase in cache coherence traffic.

6.1.3 Scheduler-Dependent and Blocking Methods

Methods in this category are either sensitive to thread failures and delays, i.e., the delay of a single thread can—and most likely will—prevent the reuse of unbounded memory indefinitely; or dependent on special kernel or scheduler support for recovery from such delays and failures.

McKenney and Slingwine [13] presented read-copy update, a framework where a retired node can be reclaimed only after ascertaining that each of the other threads has reached a *quiescence point* after the node was removed. The definition of quiescence points, varies depending on the programming environment. Typically, implementations of read-copy update use timestamps or collective reference counters. Not all environments are suitable for the concept of quiescence points, and without special scheduler support the method is blocking, as the delay of even one thread prevents the reuse of unbounded memory.

Greenwald [4] presented brief outlines of *type stable memory* implementations that depend on special support from the kernel for accessing the private variables of threads and detecting thread delays and failures. The core idea is that threads set timestamps whenever they reach *safe points* such as the kernel’s top level loop, where processes are guaranteed not to be holding stale references. The earliest timestamp represents a *high water mark* where all nodes retired before that time can be reclaimed safely.

Harris [5] presented a brief outline of a deferred freeing method that requires each thread to record a timestamp of the latest time it held no references to dynamic nodes, and it maintains two to-be-freed lists of retired nodes: an old list and a new list. Retired nodes are placed on the new list and when the time of the latest insertion in the old list precedes the earliest per-thread timestamp, the nodes of the old list are freed and the old and new lists exchange labels. The method is blocking, as the failure of a thread to update its timestamp causes the indefinite prevention of an unbounded number of retired nodes from being reused. This can happen even without thread delays. If a thread simply does not operate on the target object, then the thread’s timestamp will remain unupdated.

One crucial difference between hazard pointers and these methods [13], [4], [5] is that the former does not use reference counters or timestamps. The use of aggregate reference counters for unbounded numbers of nodes and/or the reliance on per-thread timestamps makes a memory management method inherently vulnerable to the failure or delay of even one thread.

6.1.4 Recent Work

A preliminary version of this work [17] was published in January 2002. Independently, Herlihy et al. [10] developed a memory reclamation methodology. The basic idea of their methodology is the same as that of ours. The difference is in the *Liberate* routine that corresponds to our *Scan* routine. *Liberate* is more complex than *Scan* and uses double-word CAS. Our methodology retains important advantages over theirs, regarding performance as demonstrated experimentally and regarding independence of special hardware support. Our methodology—even with the algorithm extensions in Section 3.2—uses only single-word instructions, while theirs requires double-word CAS—which is not supported on 64-bit processor architecture—in its core operations. Also, our methodology uses only reads and writes in its core operations, while theirs uses CAS in its core operations. This prevents their methodology from supporting algorithms that require only reads and writes (e.g., Fig. 10)—which are otherwise feasible—on systems without hardware support for CAS or LL/SC.

6.2 Conclusions

The problem of memory reclamation for dynamic lock-free objects has long discouraged the wide use of lock-free objects, despite their inherent performance and reliability advantages over conventional lock-based synchronization.

In this paper, we presented the *hazard pointer methodology*, a practical and efficient solution for memory reclamation for dynamic lock-free objects. It allows unrestricted memory reclamation, it takes only constant expected amortized time per retired node, it does not require special hardware support, it uses only single-word instructions, it does not require special kernel support, it guarantees an upper bound on the number of retired nodes not yet ready for reuse at any time, it is wait-free, it offers a lock-free solution for the ABA problem, and it does not require any extra shared space per pointer, per node, or per object.

Our experimental results demonstrate the overall excellent performance of hazard pointers. They show that the hazard pointer methodology offers equal and more often significantly better performance than other memory management methods, in addition to its qualitative advantages regarding memory reclamation and independence of special hardware support.

Our results also show that lock-free implementations of important object types, using hazard pointers, offer comparable performance to that of efficient lock-based implementations under no contention and no multiprogramming, and outperform them by significant margins under moderate multiprogramming and/or contention, in addition to guaranteeing continuous progress and availability even in the presence of thread failures and arbitrary delays.

A growing number of efficient algorithms for lock-free dynamic objects are available. The hazard pointer methodology enhances their practicality by enabling memory reclamation, and at the same time allowing them to achieve excellent robust performance. In combination with a recent completely lock-free algorithm for dynamic memory allocation [20], the hazard pointer methodology enables these objects at last to be completely dynamic and at the same time completely lock-free, regardless of support for automatic garbage collection.

ACKNOWLEDGMENTS

The author thanks the editor and the reviewers for valuable comments on this paper, and Marc Auslander, Maurice Herlihy, Victor Luchangco, Paul McKenney, Mark Moir, Bryan Rosenberg, Michael Scott, Ori Shalev, Nir Shavit, Yefim Shuf, and Robert Wisniewski for useful discussions and comments on various stages of this work, and Victor Luchangco for suggesting pointer renaming as an alternative to pointer ordering.

REFERENCES

- [1] T.E. Anderson, B.N. Bershad, E.D. Lazowska, and H.M. Levy, "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism," *ACM Trans. Computer Systems*, vol. 10, no. 1, pp. 53-79, Feb. 1992.
- [2] T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms*. MIT Press, 1990.
- [3] D.L. Detlefs, P.A. Martin, M. Moir, and G.L. Steele Jr., "Lock-Free Reference Counting," *Proc. 20th Ann. ACM Symp. Principles of Distributed Computing*, pp. 190-199, Aug. 2001.
- [4] M.B. Greenwald, "Non-Blocking Synchronization and System Design," PhD thesis, Stanford Univ., Aug. 1999.
- [5] T.L. Harris, "A Pragmatic Implementation of Non-Blocking Linked Lists," *Proc. 15th Int'l Symp. Distributed Computing*, pp.*nbsp;300-314, Oct. 2001.
- [6] T.L. Harris, K. Fraser, and I.A. Pratt, "A Practical Multi-Word Compare-and-Swap Operation," *Proc. 16th Int'l Symp. Distributed Computing*, pp. 265-279, Oct. 2002.
- [7] D. Hendler and N. Shavit, "Work Dealing," *Proc. 14th Ann. ACM Symp. Parallel Algorithms and Architectures*, pp. 164-172, Aug. 2002.
- [8] M.P. Herlihy, "Wait-Free Synchronization," *ACM Trans. Programming Languages and Systems*, vol. 13, no. 1, pp. 124-149, Jan. 1991.
- [9] M.P. Herlihy, "A Methodology for Implementing Highly Concurrent Objects," *ACM Trans. Programming Languages and Systems*, vol. 15, no. 5, pp. 745-770, Nov. 1993.
- [10] M.P. Herlihy, V. Luchangco, and M. Moir, "The Repeat Offender Problem: A Mechanism for Supporting Dynamic-Sized Lock-Free Data Structures," *Proc. 16th Int'l Symp. Distributed Computing*, pp. 339-353, Oct. 2002.
- [11] IBM, IBM System/370 Extended Architecture, Principles of Operation, publication no. SA22-7085, 1983.
- [12] L.I. Kontothanassis, R.W. Wisniewski, and M.L. Scott, "Scheduler-Conscious Synchronization," *ACM Trans. Computer Systems*, vol. 15, no. 1, pp. 3-40, Feb. 1997.
- [13] P.E. McKenney and J.D. Slingwine, "Read-Copy Update: Using Execution History to Solve Concurrency Problems," *Proc. 10th IASTED Int'l Conf. Parallel and Distributed Computing and Systems*, Oct. 1998.
- [14] J.M. Mellor-Crummey and M.L. Scott, "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors," *ACM Trans. Computer Systems*, vol. 9, no. 1, pp. 21-65, Feb. 1991.
- [15] J.M. Mellor-Crummey and M.L. Scott, "Scalable Reader-Writer Synchronization for Shared-Memory Multiprocessors," *Proc. Third ACM Symp. Principles and Practice of Parallel Programming*, pp. 106-113, Apr. 1991.
- [16] M.M. Michael, "High Performance Dynamic Lock-Free Hash Tables and List-Based Sets," *Proc. 14th Ann. ACM Symp. Parallel Algorithms and Architectures*, pp. 73-82, Aug. 2002.
- [17] M.M. Michael, "Safe Memory Reclamation for Dynamic Lock-Free Objects Using Atomic Reads and Writes," *Proc. 21st Ann. ACM Symp. Principles of Distributed Computing*, pp. 21-30, July 2002. earlier version in Research Report RC 22317, IBM T.J. Watson Research Center, Jan. 2002.
- [18] M.M. Michael, "CAS-Based Lock-Free Algorithm for Shared Deques," *Proc. Ninth Euro-Par Conf. Parallel Processing*, pp. 651-660, Aug. 2003.
- [19] M.M. Michael, "ABA Prevention Using Single-Word Instructions," Technical Report RC 23089, IBM T.J. Watson Research Center, Jan. 2004.
- [20] M.M. Michael, "Scalable Lock-Free Dynamic Memory Allocation," *Proc. 2004 ACM SIGPLAN Conf. Programming Language Design and Implementation*, June 2004.
- [21] M.M. Michael and M.L. Scott, "Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms," *Proc. 15th Ann. ACM Symp. Principles of Distributed Computing*, pp. 267-275, May 1996.
- [22] M.M. Michael and M.L. Scott, "Nonblocking Algorithms and Preemption-Safe Locking on Multiprogrammed Shared Memory Multiprocessors," *J. Parallel and Distributed Computing*, vol. 51, no. 1, pp. 1-26, May 1998.
- [23] S. Prakash, Y.-H. Lee, and T. Johnson, "A Nonblocking Algorithm for Shared Queues Using Compare-and-Swap," *IEEE Trans. Computers*, vol. 43, no. 5, pp. 548-559, May 1994.
- [24] L. Rudolph and Z. Segall, "Dynamic Decentralized Cache Schemes for MIMD Parallel Processors," *Proc. 11th Int'l Symp. Computer Architecture*, pp. 340-347, June 1984.
- [25] O. Shalev and N. Shavit, "Split-Ordered Lists: Lock-Free Extensible Hash Tables," *Proc. 22nd Ann. ACM Symp. Principles of Distributed Computing*, pp. 102-111, July 2003.
- [26] N. Shavit and D. Touitou, "Software Transactional Memory," *Distributed Computing*, vol. 10, no. 2, pp. 99-116, 1997.
- [27] H. Tang, K. Shen, and T. Yang, "Program Transformation and Runtime Support for Threaded MPI Execution on Shared Memory Machines," *ACM Trans. Programming Languages and Systems*, vol. 22, no. 4, pp. 673-700, July 2000.
- [28] J. Turek, D. Shasha, and S. Prakash, "Locking Without Blocking: Making Lock Based Concurrent Data Structure Algorithms Nonblocking," *Proc. 11th ACM Symp. Principles of Database Systems*, pp. 212-222, June 1992.
- [29] J.D. Valois, "Lock-Free Linked Lists Using Compare-and-Swap," *Proc. 14th Ann. ACM Symp. Principles of Distributed Computing*, pp. 214-222, Aug. 1995.
- [30] J. Zahorjan, E.D. Lazowska, and D.L. Eager, "The Effect of Scheduling Discipline on Spin Overhead in Shared Memory Parallel Systems," *IEEE Trans. Parallel and Distributed Systems*, vol. 2, no. 2, pp. 180-198, Apr. 1991.



Maged M. Michael received the PhD degree in computer science from the University of Rochester in 1997. Since then, he has been a research staff member at the IBM Thomas J. Watson Research Center at Yorktown Heights, New York. Dr. Michael's research interests include concurrent programming, memory management, distributed and fault-tolerant computing, parallel computer architecture, cache coherence protocol design, and multiprocessor simulation methodology.

▷ For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/publications/dlib.

Safe Memory Reclamation for Dynamic Lock-Free Objects Using Atomic Reads and Writes

Maged M. Michael

IBM Thomas J. Watson Research Center
P.O. Box 218 Yorktown Heights NY 10598 USA

magedm@us.ibm.com

ABSTRACT

A major obstacle to the wide use of lock-free data structures, despite their many performance and reliability advantages, is the absence of a practical lock-free method for reclaiming the memory of dynamic nodes removed from dynamic lock-free objects for arbitrary reuse.

The only prior lock-free memory reclamation method depends on the DCAS atomic primitive, which is not supported on any current processor architecture. Other memory management methods are blocking, require special operating system support, or do not allow arbitrary memory reuse.

This paper presents the first lock-free memory management method for dynamic lock-free objects that allows arbitrary memory reuse, and does not require special operating system or hardware support. It guarantees an upper bound on the number of removed nodes not yet freed at any time, regardless of thread failures and delays. Furthermore, it is wait-free, it is only logarithmically contention-sensitive, and it uses only atomic reads and writes for its operations. In addition, it can be used to prevent the ABA problem for pointers to dynamic nodes in most algorithms, without requiring extra space per pointer or per node.

1. INTRODUCTION

A shared object is lock-free (also called non-blocking) if it guarantees that in a system with multiple threads attempting to perform operations on the object, some thread will complete an operation successfully in a finite number of system steps even with the possibility of arbitrary thread delays, provided that not all threads are delayed indefinitely [10]. By definition, lock-free objects are immune to deadlock even with thread failures and provide robust performance even when faced with inopportune thread delays. Dynamic lock-free objects have the added advantage of arbitrary size. Many such objects have been developed [1, 5, 7, 8, 10, 12, 15, 16, 17, 20, 22, 24].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PODC 2002 July 21-24, 2002, Monterey California, USA.
Copyright 2002 ACM 1-58113-485-1/02/0007 ...\$5.00.

However, a major problem associated with dynamic objects is the safe reclamation of memory occupied by removed nodes. In a lock-based dynamic object, when a thread removes a node from the object, it can be guaranteed that no other thread will subsequently access the contents of the removed node while still assuming its retention of type and content semantics. Consequently, it is safe for the removing thread to free the memory occupied by the removed node into the general memory pool for arbitrary future reuse.

This is not the case for a lock-free object. When a thread removes a node, it is possible that one or more contending threads, in the course of their lock-free operations, have earlier read a pointer to the subsequently removed node, and are about to access its contents. A contending thread might corrupt the shared object or another object, if the thread performing the removal were to free the removed node for arbitrary reuse. Furthermore, on some systems, even read access to freed memory may result in fatal access errors.

Lock-free objects generally use universal atomic primitives such as CAS and LL/SC [9]. CAS (compare-and-swap) takes three arguments: the address of a memory location, an expected value and a new value. If the memory location holds the expected value, it is assigned the new value atomically. A Boolean return value indicates whether the replacement occurred. SC (store-conditional) takes two arguments: the address of a memory location and a new value. If no other thread has written the memory location since the current thread last read it using LL (load-linked), it is assigned the new value atomically. A Boolean return value indicates whether the replacement occurred. All architectures that support LL/SC restrict memory accesses between LL and SC. Associated with most uses of CAS (and restricted LL/SC) is the ABA problem [12]. If a thread reads a value A from a shared location, computes a new value, and then attempts a CAS operation, the CAS may succeed when it should not, if between the read and the CAS other threads change the value of the shared location from A to B and back to A again.

The simplest and most efficient solution to the ABA problem is to include a tag with the target memory location such that both are manipulated atomically and the tag is incremented with updates of the target location [12]. CAS succeeds only if the tag has not changed since the thread last read the location. However, applying this solution or more elaborate tag techniques [18] to pointers contained in dynamic nodes, without means of detecting when threads no longer need the tag values, dictates that the tag fields retain their values indefinitely, thus preventing the arbitrary reuse

of deleted dynamic nodes. Once allocated and inserted in a dynamic lock-free object, a dynamic node may be reused but only if it retains its size and the semantics of its tag fields.

Prior memory management methods for lock-free dynamic objects suffer from one or more serious drawbacks: not allowing arbitrary memory reuse, blocking, requiring special operating system support, or using DCAS.

In this paper we present a new method for safe memory reclamation (SMR) that is wait-free¹, is operating system independent, is only logarithmically contention-sensitive, requires no extra space per node but only a small constant space per participating thread, and requires only atomic reads and writes. It can be used to prevent the ABA problem for dynamic pointers without the need for extra space per pointer or per node. SMR applies to the vast majority of known lock-free dynamic algorithms, and in each of the few cases of incompatible algorithms [5, 8], other similar or better algorithms for the same objects were found to be compatible with it [15, 16].

The core idea of SMR is to associate a small constant number K (typically no more than three) of shared pointers, called *hazard pointers*, with each participating thread. Hazard pointers, either have null values or point to nodes that may potentially be accessed by the thread without further verification of the validity of the local references used in their access. Each hazard pointer can be written only by its associated thread, but can be read by all threads.

SMR requires target lock-free algorithms to guarantee that no thread can access a dynamic node at a time when it is possibly deleted, unless its associated hazard pointers have been pointing to the node continuously, from a time when it was not deleted.

For example, Figure 1 shows an SMR-compatible version of a lock-free stack based on the IBM freelist [12] that guarantees that no dynamic node is accessed while free and prevents the ABA problem. The pointer hp is a static private pointer to the hazard pointer associated with the executing thread, and the procedure `DeleteNode` is part of the SMR algorithm (Section 3). The `Push` routine need not change, as no dynamic node that is possibly free is accessed, and the CAS is not ABA-prone as observed by Treiber [22]. In the `Pop` routine the pointer t is used to access a dynamic node t^{\wedge} and holds the expected value of an ABA-prone CAS. By setting the hazard pointer to t (line 2) and then checking that t^{\wedge} is not deleted (line 3), we guarantee that the hazard pointer is continuously pointing to t^{\wedge} from a point when it was not deleted (line 3) until the end of hazards, i.e., accessing t^{\wedge} (line 4) and using t to hold the expected value of an ABA-prone CAS (line 5).

SMR prevents the freeing of a node continuously pointed to by one or more hazard pointers of one or more threads from a point prior to its deletion. When a thread deletes a node by calling `DeleteNode`, it stores it in a private list. After accumulating a certain number R of deleted nodes, the thread scans the hazard pointers for matches for the addresses of the accumulated nodes. If a deleted node is not matched by any of the hazard pointers, then the thread frees that node making its memory available for arbitrary reuse. Otherwise, the thread keeps the node until its next scan of

¹An operation is wait-free if it is guaranteed to complete successfully in a finite number of its own steps regardless of other threads' actions [9].

```
// j is the thread id for SMR purposes
// HP is the shared array of hazard pointers
// static private pointer hp = &HP[j]
// initially Top = null

// calling Push
// if node ← AllocateNode() ≠ null
//   { node^.Data ← data; Push(node);}
Push(node:*NodeType) {
    while true {
        t ← Top;
        node^.Next ← t;
        if CAS(&Top,t,node) return;
    }
}

// calling Pop
// if node ← Pop() ≠ null
//   { data ← node^.Data; DeleteNode(node);}
Pop() : *NodeType {
    while true {
        1   if t ← Top = null break;
        2   *hp ← t;
        3   if t ≠ Top continue;
        4   next ← t^.Next;
        5   if CAS(&Top,t,next) break;
    }
    *hp ← null; return t;
}
```

Figure 1: SMR-compatible lock-free stack based on the IBM freelist algorithm [12] (SMR-related code is in a different font).

the hazard pointers which it performs after the number of accumulated deleted nodes reaches R again.

By setting R to a number such that $R - N = \Omega(N)$ (e.g., $R = 2N$), where $N = KP$ and P is the number of participating threads, and sorting a private list of snapshots of non-null hazard pointers, SMR is guaranteed in every scan of the N hazard pointers to free $\Theta(R)$ nodes in $O(R \log p)$ time, when p threads have non-null hazard pointers. Thus, the amortized time complexity of processing each deleted node until it is freed is only logarithmically contention-sensitive, i.e., constant in the absence of contention and $O(\log p)$ when p threads are operating on the object during the scan of their associated hazard pointers. SMR also guarantees that no more than PR deleted nodes are not yet freed at any time regardless of thread failures and delays.

In Section 2 we review prior approaches to memory management for dynamic lock-free objects. In Section 3 we present the SMR method. In Section 4 we discuss its correctness and performance. In Section 5 we apply SMR to known dynamic lock-free algorithms and we conclude with Section 6.

2. RELATED WORK

Lock-free OS-Independent Methods

The earliest and simplest lock-free memory management method is the use of a freelist implemented as a stack of nodes available for restricted reuse [12, 22]. When a thread

deletes a node, it pushes it in a singly linked list stack, where the same or a different thread can later reuse the node by popping it from the stack. The operations are similar to those in Figure 1 excluding data and hazard pointer manipulation and node allocation and deallocation. Nodes must retain their size and the semantics of some fields indefinitely (i.e., until application termination). Once allocated and inserted in a lock-free object, a node cannot be freed for arbitrary reuse. Freelist require applying ABA prevention mechanisms such as tags to the anchor of the freelist as well as pointers contained in nodes for almost all lock-free objects except stacks.

Valois [25] presented a lock-free garbage collection method that requires the inclusion of a reference counter in each dynamic node, reflecting the maximum number of references to that node in the object and the registers (and local variables) of threads operating on the object. Every time a new reference to a dynamic node is created/destroyed, the reference counter is incremented/decremented, using fetch-and-add and CAS instructions. Only after its reference counter goes to zero, can a node be pushed in a freelist, where it can be subsequently allocated. However, due to the use of single-address CAS to manipulate pointers and independently located reference counters non-atomically, the resulting timing windows dictate the permanent retention of nodes of their type and field semantics, thus prohibiting memory reclamation as is the case with freelist.

Detlefs *et. al.* [6] presented lock-free reference counting (LFRC), a lock-free garbage collection method that uses DCAS² to operate on pointers and reference counters atomically to guarantee that a reference counter is never less than the actual number of references. When the reference counter of a node reaches zero, it becomes safe to reclaim for arbitrary reuse. Both reference counting methods avoid the ABA problems without using a tag per pointer (they use a counter per node instead). LFRC also allows memory reclamation, which is a significant advantage. However, the dependence on DCAS makes the method impractical. For both methods the performance cost of reference counting is prohibitive.

The reference counting methods transform statements involving pointers to dynamic nodes (even reads and register to register assignments) into unbounded loops containing CAS and DCAS operations. This increases the total work time complexity of the target algorithms by a *multiplicative* factor of $O(p)$ where p is contention.

DCAS was supported on some generations of the Motorola 68000 processor family (as CAS2) in the 1980s. These implementations were extremely inefficient, often requiring locking system buses while one CAS2 is in progress. Since then no processor architecture supports DCAS.

As discussed later in Section 5, while examining known lock-free dynamic algorithms, we found only two [5, 8] to be incompatible with SMR and freelist, but compatible with the reference counting methods. However, in each case, we found other similar or better algorithms that are compatible with SMR and freelist for the same objects [15, 16].

²DCAS (double-compare-and-swap) takes six arguments: the addresses of two independent memory locations, two expected values and two new values. If both memory locations hold the corresponding expected values, they are assigned the corresponding new values atomically. A Boolean return value indicates whether the replacements occurred.

Blocking and OS-Dependent Methods

Herlihy and Moss [11] presented a lock-free garbage collection method that requires a clean sweep from each participating thread, as a precondition for memory reclamation. The failure of a thread can indefinitely prevent the freeing of unbounded memory (i.e., bounded only by the size of memory). The method also suffers from substantial copying overhead.

McKenney and Slingwine [14] presented read-copy update, a framework where only when each thread is certain to have reached a *quiescence point* after a node is deleted, can the memory of that node be freed. The definition of quiescence points, if any, varies depending on the environment, but mostly depends on timestamps or collective reference counters. Not all environments are suitable for the concept of quiescence points, and the method is blocking as the delay of even one thread prevents freeing an unbounded number of nodes.

Greenwald [7] presented sketches of Type Stable Memory (TSM) implementations in the OS kernel and in user-level. TSM requires deleted nodes to retain their type while references to them are potentially active. The kernel-based TSM implementation relies on the kernel’s knowledge of the status of processes and its access to their private memory and stack space. The drawbacks of kernel-dependence are that it limits the portability of dependent algorithms to systems lacking the special kernel support, and precludes their use by most user-level threads.

The user level TSM implementation requires threads to increment and decrement a per-type reference counter upon the activation and end of scope, respectively, of local pointers to the node type. A variant of the method uses two reference counters per type: an old counter and a new counter. A thread always increments the new counter and decrements the counter it incremented earlier. The counters switch roles whenever the old counter reaches zero. At such time, deleted nodes are released to the general pool. The method is blocking as the failure of a thread to decrement a per-type reference counter, due to the thread’s failure or delay, may cause the indefinite prevention of an unbounded number of deleted nodes from being freed.

Harris [8] presented a brief outline of a deferred freeing method that requires each thread to record a timestamp of the latest time it held no references to dynamic nodes, and it maintains two to-be-freed lists of deleted nodes: an old list and a new list. Deleted nodes are placed on the new list and when the time of the latest insertion in the old list precedes the earliest per-thread timestamp, the nodes of the old list are freed and the old and new lists exchange labels. The method is blocking, as the failure of a thread to update its timestamp causes the indefinite prevention of an unbounded number of deleted nodes from being freed.

One crucial difference between SMR and these methods [7, 8, 14] is that SMR does not use reference counters or timestamps at all. The use of collective reference counters for unbounded numbers of nodes and/or the reliance on per thread timestamps make a memory management method vulnerable to the failure or delay of even one thread.

Tracing garbage collectors do not provide OS-independent non-blocking solutions for the memory reclamation problem. They mostly require mutual exclusion with mutator threads, use stop-the-world techniques, or require special OS support to access private stack space and registers [11]. Furthermore,

the failure or indefinite delay of the garbage collector can prevent the reclamation of unbounded number of deleted nodes indefinitely [6].

3. THE METHOD

The core idea of the SMR method is associating K shared pointers, called hazard pointers, with each of the P threads operating on the target object. The value of K depends on the target algorithm and is typically no more than three. Hazard pointers are implemented as a shared array HP of size N , where $N = KP$.

The SMR algorithm communicates with the target algorithms only through a `DeleteNode` procedure that is part of the SMR algorithm, and the hazard pointer array HP . Each hazard pointer can be written only by its associated thread in the target algorithm. The SMR algorithm itself does not perform any shared writes.

3.1 The Algorithm

Figure 2 shows the structures and operations of the SMR algorithm. We assume a sequentially consistent memory model. Otherwise, memory barriers may be needed in between instructions with critical relative order.

When a thread j deletes a node n^{\wedge} by calling `DeleteNode(n)`, it inserts n^{\wedge} into a static private list $dlist$ of deleted nodes, and increments a static private counter $dcount$ that holds the number of deleted nodes accumulated by j that are not freed yet. When $dcount$ reaches the value R , j starts a scan by calling `Scan()`. R is chosen such that $R - N = \Omega(N)$ (e.g., $R = 2N$), in order to achieve amortized execution time that is logarithmic in contention per freed node.

A scan includes four stages. The first stage involves scanning the array HP for non-null values. Whenever a non-null value is encountered, it is inserted in a local pointer list $plist$. The counter p holds the size of $plist$, which is proportionally bounded by contention. Only stage 1 accesses shared variables. The following stages operate only on private variables.

The second stage of a scan involves sorting $plist$ to allow binary search in the third stage.

The third stage of a scan involves checking each node in $dlist$ against the pointers in $plist$. If the binary search yields no match, the node is freed. Otherwise, it is inserted into a local list new_dlist .

The forth stage involves copying the local list new_dlist of the nodes that could not be freed during the current scan to the private static list $dlist$, where they remain until the next scan, which occurs after $R - new_dcount$ more nodes are deleted by j .

We assume the use of a comparison-based sorting algorithm that takes $\Theta(p \log p)$ time, such as heap sort, to sort $plist$ in the second stage. Binary search in the third stage takes $O(\log p)$ time. We omit the code for these algorithms, as they are widely known sequential algorithms [4].

Optimizations

The static space requirements per thread can be reduced to a constant, if the node structures used by the target algorithm contain a word size field that is guaranteed not to be accessed by the target algorithm after a node is deleted. The vast majority of algorithms use nodes that contain such fields (e.g., data fields). Also, nodes often contain extra space for alignment or for cache line padding to avoid false

```
// Constants
// P : number of participating threads
// K : number of hazard pointers per thread
// N : total number of hazard pointers = KP
// R : batch size, R-N = Ω(N)

// shared variables
HP[N] = null : *NodeType;

// static private variables per thread
dcount = 0 : 0..R;
dlist[R] : *NodeType;

DeleteNode(node:*NodeType) {
    dlist[dcount++] ← node;
    if dcount = R Scan();
}

Scan() {
    i : 0..R;
    p = 0, new_dcount = 0 : 0..N;
    hptr, plist[N], new_dlist[N] : *NodeType;

    // Stage 1
    for i ← 0 to N-1
        if hptr ← HP[i] ≠ null
            plist[p++] ← hptr;
    // Stage 2
    sort(p, plist);
    // Stage 3
    for i ← 0 to R-1
        if binary-search(dlist[i], p, plist)
            new_dlist[new_dcount++] ← dlist[i];
        else
            FreeNode(dlist[i]);
    // Stage 4
    for i ← 0 to new_dcount-1 { dlist[i] ← new_dlist[i]; }
    dcount ← new_dcount;
}


```

Figure 2: The SMR algorithm.

sharing. If so, the chosen field is also defined as a pointer $smrp$ to `NodeType` and is used to link deleted nodes into a linked list instead of using an array of size R per thread to accumulate deleted nodes.

Removing duplicates from $plist$ after sorting it can reduce search time when duplicates are frequent, which is very common for constant time algorithms such as those for stacks and queues. Figure 3 shows a version of the SMR algorithm incorporating these optimizations.

In order to reduce the overhead of calling the standard allocation and deallocation procedures (e.g., `malloc` and `free`) for every node allocation and deallocation, SMR can allow each thread to maintain a limited size private list of free nodes. When a thread runs out of private free nodes it allocates new nodes when needed, and when a thread accumulates too many private free nodes it deallocates the excess nodes.

In order to avoid the adverse performance effect of false sharing on multiprocessor systems, elements of the HP array can be padded such that no two hazard pointers belonging to different threads share the same cache line.

```

// Constants and shared vars are unchanged

// static private variables per thread
dcount = 0: 0..R;
dlist = null : *NodeType;

DeleteNode(node:*NodeType) {
    node^.smrp ← dlist; dlist ← node; dcount++;
    if dcount = R Scan();
}

Scan() {
    i,p=0,new_dcount = 0 : 0..N;
    hptr,plist[N],new_dlist = null,node : *NodeType;

    // Stage 1
    for i ← 0 to N-1
        if hptr ← HP[i] ≠ null
            plist[p++] ← hptr;
    // Stage 2
    sort(p,plist);
    remove_duplicates(&p,plist);
    // Stage 3
    while dlist ≠ null {
        node ← dlist; dlist ← node^.smrp;
        if binary_search(node,p,plist)
            { node^.smrp ← new_dlist;
              new_dlist ← node; new_dcount++;}
        else
            FreeNode(node);
    }
    // Stage 4
    dlist ← new_dlist;
    dcount ← new_dcount;
}

```

Figure 3: The SMR algorithm with optimizations.

In many applications, threads are created and destroyed frequently. In such cases, a static-size (of size P) lock-free freelist based on the IBM freelist [12] can be used for maintaining available thread ids. A thread acquires an SMR thread id by popping it from the freelist, and retires its id by pushing it in the freelist. The freelist elements can include fields for use by retiring threads to pass their list of deleted but not yet freed nodes to their next namesake.

3.2 Compatibility Conditions

For a correct target algorithm to benefit from SMR's guarantees of safe memory reclamation, it must satisfy a set of conditions. First, we define some notation.

Delete(t, j, n): Thread j deletes node n^{\wedge} at time t : at t , j calls DeleteNode(n).

Allocate(t, j, n): Thread j allocates node n^{\wedge} at time t : at t , j 's call to AllocateNode() returns n .

IsDeleted(t, n): Node n^{\wedge} is deleted at time t : $\exists t_1 < t, j_1 :: Delete(t_1, j_1, n) \wedge \exists t_2 \in [t_1, t], j_2 :: Allocate(t_2, j_2, n)$.

PossiblyDeleted(t, j, n): A node n^{\wedge} is possibly deleted from the point of view of thread j at time t : at t , it is impossible solely by examining j 's registers (private variables included) and the semantics of the algorithm to establish that $\neg IsDeleted(t, n)$.

CreateRef(\hat{t}, j, n, \hat{r}): Thread j creates the reference

$\langle \hat{t}, j, n, \hat{r} \rangle$ to node n^{\wedge} in register \hat{r} at time \hat{t} : \exists shared pointer $p ::$ at \hat{t} , j reads the value n from p into \hat{r} .

AssignRef($\hat{t}, j, n, \hat{r}, t, r$): Thread j assigns the reference $\langle \hat{t}, j, n, \hat{r} \rangle$ to register r at time t : $(t = \hat{t} \wedge r \equiv \hat{r}) \vee (\exists$ register r_1 , time $t_1 \in [\hat{t}, t] :: AssignRef(\hat{t}, j, n, \hat{r}, t_1, r_1) \wedge$

at t , j assigns the value of r_1 to $r \wedge \exists t_2 \in (t_1, t] ::$ at t_2 , r_1 is the target of an assignment).

HoldRef($\hat{t}, j, n, \hat{r}, t, r$): Register r of thread j holds reference $\langle \hat{t}, j, n, \hat{r} \rangle$ at time t : $\exists t_1 \leq t :: AssignRef(\hat{t}, j, n, \hat{r}, t_1, r) \wedge \exists t_2 \in (t_1, t] ::$ at t_2 , r is the target of an assignment.

HoldHazRef($\hat{t}, j, n, \hat{r}, t, r$): Register r of thread j holds the hazardous reference $\langle \hat{t}, j, n, \hat{r} \rangle$ at time t :

$HoldRef(\hat{t}, j, n, \hat{r}, t, r) \wedge \exists$ a code path c , statement $s ::$ at t , it is possible for j to follow $c \wedge s$ is the last statement of $c \wedge \exists$ register $r_2 :: s$ uses r_2 to access n^{\wedge} when it is $PossiblyDeleted \wedge ((\exists t_2 \geq t :: j$ follows $c \wedge j$ executes s at $t_2) \implies ((r \equiv r_2 \wedge \exists$ statement in c that uses r as the target of an assignment) $\vee (\exists r_1, t_1 \in [t, t_2] ::$ at t_1 j assigns the value of r to $r_1 \wedge HoldHazRef(\hat{t}, j, n, \hat{r}, t_1, r_1)))$.

Informally, A hazardous reference is an address that without further validation can be used to access a node after it has been deleted.

The Conditions

For a lock-free dynamic algorithm to be SMR-compatible, it must satisfy the following main condition:

$$\forall t, n, j, r, \hat{t}, \hat{r}, \\ PossiblyDeleted(t, j, n) \wedge HoldHazRef(\hat{t}, j, n, \hat{r}, t, r) \implies \\ \text{at } t, \exists 0 \leq i < K :: HP[Kj + i] = n \quad (C1)$$

Informally, whenever a thread accesses a dynamic node, it must guarantee that if the node was possibly deleted (by another thread) after the creation of the reference to it, then continuously from a time equal to or earlier than the time of its possible deletion, one or more of the thread's hazard pointers is pointing to the node.

When $K > 1$, it is acceptable for a thread to overwrite the address of a node n^{\wedge} in one of its associated hazard pointers, when n^{\wedge} is possibly deleted and one of its registers holds a hazardous reference to n^{\wedge} , as long as another associated hazard pointer was already assigned the value n .

However, since the SMR algorithm scans the array HP non-atomically (one hazard pointer at a time) in a certain order, passing the responsibility for hazardous references from one hazard pointer to another must strictly follow the same order as scanning HP. This leads to the second condition:

$$\forall j, n, t_1 < t_2, 0 \leq i_1 < K, 0 \leq i_2 < K, i_1 \neq i_2, \\ ((\exists \hat{t}, \hat{r}, r :: \forall t \in [t_1, t_2], \text{at } t, \\ PossiblyDeleted(t, j, n) \wedge \\ HoldHazRef(\hat{t}, j, n, \hat{r}, t, r)) \wedge \\ (\text{at } t_1, HP[Kj + i_1] = n \wedge HP[Kj + i_2] \neq n) \wedge \\ (\text{at } t_2, HP[Kj + i_1] \neq n \wedge HP[Kj + i_2] = n)) \\ \implies i_1 < i_2 \quad (C2)$$

The last condition is optional. It is needed only to guarantee that the number of non-null hazard pointers (p at the beginning of stage 3 of Scan) is proportionally bounded by contention:

$$\forall t, j, 0 \leq i < K, \\ \text{at } t, \text{thread } j \text{ is not operating on the object} \implies \\ \text{at } t, HP[Kj + i] = null \quad (C3)$$

4. CORRECTNESS AND PERFORMANCE

For brevity, we provide only informal proof sketches. Formal proofs are to be included in the extended version of this paper.

Safety

LEMMA 1. $\forall j, t, n, \exists k < K, t_0 \leq \dots \leq t_{k-1} \leq t :: \forall 0 \leq i < k, \text{at } t_i, HP[Kj+i] \neq n \wedge IsDeleted(t, n) \wedge \exists r, \hat{r}, \hat{t} :: HoldHazRef(\hat{t}, j, n, \hat{r}, t, r) \implies \exists k \leq m < K :: \text{at } t, HP[Kj+m] = n.$

If a scan in ascending order of low indexed hazard pointers of a thread finds no match for a deleted node for which the thread holds a hazardous reference, then there must be at least one higher indexed hazard pointer of that thread that points to that node. This follows from (C1) and (C2) and the fact that $IsDeleted(t, n) \implies \forall j, PossiblyDeleted(t, j, n)$.

LEMMA 2. $\forall j, t, n, \exists t_0 \leq \dots \leq t_{K-1} \leq t :: \forall 0 \leq i < K, \text{at } t_i, HP[Kj+i] \neq n \wedge IsDeleted(t, n) \implies \exists r, \hat{r}, \hat{t} :: HoldHazRef(\hat{t}, j, n, \hat{r}, t, r).$

If a scan in ascending index order of the hazard pointers of a thread finds no match for a deleted node, then it must be the case that the thread holds no hazardous reference for the node. This follows from Lemma 1.

LEMMA 3. $\forall t, n, \text{node } n^{\wedge} \text{ is freed at } t \text{ in stage 3 of Scan} \implies \forall j, \exists t_0 \leq \dots \leq t_{K-1} \leq t :: \forall 0 \leq i < K, \text{at } t_i, HP[Kj+i] \neq n.$

A node is freed in stage 3 of Scan only if a scan of the hazard pointers in ascending index order finds no match. This follows from the fact that stage 1 scans HP in ascending index order and the fact that the list *plist* does not lose distinct pointer values throughout stages 1, 2, and 3. That is, if a pointer value is not in *plist* in stage 3 then it couldn't have been there any time during stage 1.

THEOREM 1. $\forall t, n, \text{node } n^{\wedge} \text{ is freed at } t \text{ in stage 3 of Scan} \implies \exists j, r, \hat{r}, \hat{t} :: HoldHazRef(\hat{t}, j, n, \hat{r}, t, r).$

If SMR frees a node then it must be the case that no thread holds a hazardous reference to it. This follows transitively from Lemmas 2 and 3 and the fact that only deleted nodes are processed in Scan.

From the definition of HoldHazRef, we get the following corollary.

COROLLARY. *SMR guarantees that no thread accesses the contents of a node while the node is free.*

Time Complexity

LEMMA 4. *At the beginning of Scan, the list *dlist* contains exactly R distinct deleted nodes.*

LEMMA 5. *Throughout Scan, $p \leq N$.*

LEMMA 6. *At most N of the searches of the list *plist* in stage 3 of Scan find matching pointers.*

LEMMA 7. *Every call to Scan results in freeing at least $R - N$ (i.e., $\Theta(R)$) nodes.*

LEMMA 8. *The execution time of DeleteNode is constant.*

LEMMA 9. *The execution time of Scan (excluding calls to FreeNode) is $O(R \log p)$.*

Stage 1 takes $\Theta(N)$ time. Stage 2 takes $\Theta(p \log p)$ time. Stage 3 takes $O(R \log p)$ time (excluding calls to FreeNode). Stage 4 takes $O(p)$ time (constant time if *dlist* is implemented as a linked list). $p = O(N)$. $N = O(R)$.

THEOREM 2. *The amortized time complexity of processing a deleted node until freeing it is $O(\log p)$.*

Wait-Freedom

THEOREM 3. *The SMR algorithm is wait-free.*

A thread executing the SMR algorithm (at most R calls to DeleteNode and one call to Scan) is guaranteed to complete successfully (i.e., free $\Theta(R)$ nodes) in a finite number of its own steps (i.e., $O(R \log p)$).

Bound on Number of Deleted Nodes not Freed

LEMMA 10. $\forall t, j, |\{n : \exists t_1 < t :: Delete(t_1, j, n) \wedge \exists t_2 \in [t_1, t] :: \text{at } t_2, j \text{ frees } n^{\wedge}\}| \text{ is finite (at most } R\text{)}.$

Each thread holds a finite number of deleted but not yet freed nodes.

THEOREM 4. *At any time, the total number of deleted nodes not freed is finite (at most PR).*

There is a tradeoff between the bound on deleted nodes not freed and the amortized time of processing each freed node. In an earlier version of the Scan algorithm that used single-word CAS, the upper bound on the number of deleted nodes not freed was only $O(P)$ (constant R), but the amortized time complexity per freed node was $\Theta(P)$.

Space Requirements

The space requirements of SMR are N (i.e., $\Theta(P)$) shared words for hazard pointers and $\Theta(R)$ static private words per thread (only a constant when using a linked list to store deleted nodes instead of an array as in Figure 3). The temporary space needed per thread when performing a scan is $\Theta(P)$.

5. APPLYING SMR

In this section, we apply the SMR method to the best known dynamic lock-free algorithms. We examined dozens of dynamic lock-free algorithms, many of which are not mentioned here for brevity. Only two correct algorithms [5, 8] were found to be inherently incompatible with SMR as well as freelists. In each case a similar or better SMR-compatible algorithm is found for the same object.

ABA Prevention

In all of the cases of applying SMR to dynamic algorithms discussed in this section, it not only solved the memory reclamation problem, but was also used to prevent the ABA problem for all or some pointers without using any extra space per pointer or per node. This was done by expanding the definition of hazardous references (defined in Section 3) to include references to dynamic nodes that may be used as expected values of ABA-prone instructions. However, we

do not claim SMR to be a general solution for the ABA problem for pointers to dynamic nodes, as we can construct hypothetical cases where a reference to a dynamic node can be the expected value of an ABA-prone instruction even when that node is never removed and reallocated. Also, in some SMR-compatible algorithms (e.g., [15]), a pointer may point to nodes that are already deleted and still be the target of ABA-prone instructions. In such cases, it may be preferable to use ABA-prevention tags than to reconstruct the algorithm to prevent the deletion of nodes pointed to by pointers that are possible targets of ABA-prone instructions. Of course, in such cases, the SMR algorithm will still be able to detect when the dynamic node containing such a pointer can be freed safely.

We present and discuss SMR-compatible versions of the best known dynamic lock-free algorithms for FIFO queues, stacks, double-ended queues, list-based sets, priority queue skew heaps, and universal methodologies. We also discuss the cases of algorithms that are inherently incompatible with SMR.

FIFO Queues

We were able to create SMR-compatible versions of all three dynamic lock-free FIFO queue algorithms [20, 24, 17] that are correct, completely defined, fully functional, CAS-based, and have constant time operations in the absence of contention.

Figure 4 shows an SMR-compatible version of Michael and Scott’s [17] algorithm. In the enqueue routine, we notice that register t holds hazardous references. It is used to access dynamic nodes (lines 4 and 7). It holds the expected value of ABA-prone CAS operations (lines 6 and 8). It holds the expected value of an ABA-prone validation condition (line 5). Therefore we dedicate a hazard pointer ($*hp0$) for pointing to t^\wedge whenever t holds a hazardous reference.

The pattern of lines 1–3 is ubiquitous in SMR-compatible algorithms. We observe that after executing line 1 and before executing line 3, it is possible that t^\wedge is removed and then reinserted in the object. But this window poses no problem. During that period the reference held in t is not hazardous, as line 3 verifies the reachability of t^\wedge , and hence precludes the possibility of it being deleted at that point. The reference only becomes hazardous immediately after passing the condition in line 3. If t^\wedge is removed before line 2 and not reinserted before line 3, the condition in line 3 forces the thread to retry. If t^\wedge is reinserted before line 3 and the condition in line 3 allows the thread to proceed then at that point it is guaranteed that $*hp0$ points to t^\wedge from a point when it was not deleted and continues to do so until the reference is no longer hazardous, thus complying with condition (C1).

We employed the technique of lines 1–3 in creating SMR-compatible transformations of most known algorithms. We present additional and alternative techniques when discussing list-based sets and universal methodologies.

The dequeue routine employs a similar technique for dealing with register h . For the register $next$, no equality check ($if next \neq t^\wedge.Next continue;$) is needed as the validation condition in line 15 (from the original algorithm) guarantees that $*hp1$ points to $next^\wedge$ from a point when it was not deleted ($next^\wedge$ cannot be removed unless h^\wedge is removed first) and continues to do so until the reference in $next$ is no longer

```
// j is thread id for SMR purposes
// hp0 = &HP[2*j]
// hp1 = &HP[2*j+1]

// initially both Head and Tail point to a dummy node

Enq(data:DataType) : boolean {
    if node ← AllocateNode() = null return false;
    node^.Data ← data; node^.Next ← null;
    while true {
        1   t ← Tail;
        2   *hp0 ← t;
        3   if t ≠ Tail continue;
        4   next ← t^.Next;
        5   if t ≠ Tail continue;
        6   if next ≠ null { CAS(&Tail,t,next); continue; }
        7   if CAS(&t^.Next,null,node) break;
    }
    8   CAS(&Tail,t,node);
    *hp0 ← null; return true;
}

Deq() : DataType {
    while true {
        9   h ← Head;
        10  *hp0 ← h;
        11  if h ≠ Head continue;
        12  t ← Tail;
        13  next ← h^.Next;
        14  *hp1 ← next;
        15  if h ≠ Head continue;
        16  if next = null { *hp0 ← null; return EMPTY; }
        17  if h = t { CAS(&Tail,t,next); continue; }
        18  data ← next^.Data;
        19  if CAS(&Head,h,next) break;
    }
    *hp0 ← null; *hp1 ← null;
    DeleteNode(h); return data;
}
```

Figure 4: SMR-compatible version of Michael and Scott’s [17] lock-free queue algorithm.

hazardous (i.e. after line 18). Only when register t is equal to h , is it used in CAS. So it is always covered by $*hp0$. No other registers in the algorithm can hold hazardous reference, and so the algorithm satisfies condition (C1).

It is easy to show that the algorithm satisfies condition (C2) since no hazard pointer inherits its value from another hazard pointer. As for condition (C3), the algorithm guarantees that whenever a thread exits Enq or Deq each of its hazard pointers is equal to null.

Stacks

In Figure 1, we presented an SMR-compatible lock-free stack based on the IBM freelist [12]. The techniques employed in transforming it are covered by those employed in lines 1–3 of Figure 4.

Double-Ended Queues (Deques)

Agesen *et. al.* [1] presented a DCAS-based dynamic lock-free deque algorithm. We were successful in making that

algorithm SMR-compatible. Detlefs *et al.* [5] presented a simpler algorithm that also uses DCAS. However, the algorithms assumes automatic garbage collection, since after popping a node, it is not always possible for a thread, solely based on its registers, to determine if it had removed nodes from the deque’s doubly-linked list or not. The algorithm is inherently incompatible with SMR and freelists. It is impossible to determine when to call `DeleteNode`, which is an integral part of explicit memory management methods.

Recently, we developed the first CAS-based lock-free deque algorithm [15]. The algorithm is compatible with SMR and freelists (as well as all other memory management methods). As part of presenting that algorithms, we demonstrate its compatibility with SMR. The techniques employed in generating its SMR-compatible version are covered by those employed in Figure 4.

List-Based Sets

Valois [25] presented CAS-based lock-free algorithms for link-based sets that use shared cursors, such that whenever a thread operates on the object it first moves the cursor (using CAS) to the desired location and then attempts its intended operation, such as inserting, deleting, and searching for a key. Actually, the algorithms are not even livelock-free, as two threads may indefinitely alternate moving the cursor to their respective desired locations, while neither succeeds in performing its intended operation on the object.

Harris [8] presented a lock-free list-based set algorithm using CAS. The algorithm cannot use freelists for memory management. Harris implemented the algorithm using Valois’ [25] memory management method, and in order to avoid the high overhead of reference counting he also introduced his own memory management method that is actually blocking, as discussed in Section 2. Harris’ set algorithm is not compatible with SMR as a thread may traverse a sequence of nodes after they have already been removed from the object, and hence possibly deleted. Thus, it is impossible for the algorithm to comply with SMR’s condition (C1).

Recently, we developed a simple CAS-based lock-free list-based set algorithm (as part of a lock-free hash table algorithm [16]) that is compatible with all memory management methods. It yields better performance than Harris’ algorithm, as it is compatible with simpler and better performing memory management methods. As part of presenting that algorithm we demonstrate its compatibility with SMR.

The algorithm involves the traversal of the linked list using three registers, we use three hazard pointers per thread to track the registers. As the registers move down the list, the third register assumes the value of the second, then the second assumes the value of the first, and then the first moves to the next node. The hazard pointers mirror the same movement. The first hazard pointer uses a technique similar to that used in lines 1–3 of Figure 4. The third and second hazard pointers inherit their values from the second and the first hazard pointers, respectively. So, in order to comply with SMR’s condition (C2), the indices in the array HP of the first, second and third hazard pointers associated with each thread must be in ascending order, respectively.

Universal Methodologies

Herlihy [10] presented a universal methodology for transforming sequential dynamic object into lock-free ones. The methodology uses a single anchor for the target object. The

value of the anchor must change with every modification of the object. A thread operating on the object starts by reading the anchor then as it traverses the linked object it makes a private copy of each node it needs to access and uses the copy instead. At the end of the operation the thread uses CAS (or SC) to change the anchor to a new value. If successful the private nodes (if any) become part of the object, and the thread puts the removed nodes in a private pool of nodes. If the CAS is not successful, the thread takes the allocated private nodes back into its private pool and starts over by reading the anchor again. Herlihy provides a recoverable set algorithm for maintaining the pool of private nodes. In order to establish the validity of copied nodes, Herlihy uses two check bits written and read in reverse order.

We demonstrate an SMR-compatible version of Herlihy’s methodology, using a priority queue skew heap as an example in Figure 5. We also provide memory management routines compatible with SMR and similar in functionality to Herlihy’s implementation of private node pools. The sequential operations `skew_deq` and `skew_meld` need no fundamental modification. Only two parts involve hazard pointers: the outer routines `End` and `Deq`, and the `copy` function. Only two hazard pointers are needed. The first continuously points to the anchor node. The setting of the first hazard pointer uses the same technique as that of lines 1–3 of Figure 4.

In the `copy` routine, a thread sets its second hazard pointers to the address of the source node (of the `copy` operation) then it verifies that the anchor has not been removed. Since no node that was reachable from an anchor node is removed without the anchor node being removed. Since the anchor node is already covered by the first hazard pointer and the source node is already pointed to by the second hazard pointer, the thread can proceed to perform the `copy` safely. When the `copy` is done the second hazard pointer can be safely nullified and reused in subsequent calls to the `copy` function. SMR eliminates the need for the two check bits for verifying the consistency of a copied node.

Turek, Shasha, and Prakash [23] and Barnes [3] presented universal methodologies for transforming lock-based implementation (including those using multiple locks) into lock-free ones. The methodologies translate each atomic statements (one instruction involving shared locations and any number of instructions involving only registers and private variables) and lock operation of the original algorithm into a continuation. When a thread operates on the object it tries to establish its sequence of continuations as the current operation. If it finds another operation in progress, it tries to help it complete by reading the current step and attempting to execute it. These methodologies apply to both static and dynamic objects. To be SMR-compatible, the programmer must indicate in each continuation if a certain argument is a dynamic node. In such a case, a technique similar to that in the `copy` routine of Figure 5 can be used but instead of verifying the value of the anchor Q, the address of the continuation is verified.

Other universal methodologies provide multi-word or multi-address CAS or LL/SC implementations from single address CAS (or LL/SC) [2, 7, 13, 19, 21]. When applicable to dynamic objects, these techniques are orthogonal to SMR-compatibility, as we have seen that CAS and DCAS can be used to develop algorithms that are compatible with SMR as well as those that are not.

```

// static private variables for methodology
r : result_type; old_q : *skew_type;
// static private variables for memory management
alloc_ptr,avail_ptr = 0..MAX_SIZE;
alloc[MAX_SIZE],removed[MAX_SIZE] : *skew_type;

Enq(v:valueType) {
    node ← allocate_node(); node^.value ← v;
    node^.Child[0] ← null; node^.Child[1] ← null;
    while true {
        removed_ptr ← 0; alloc_ptr ← 0;
        old_q ← Q;
        *hp0 ← old_q;
        if old_q ≠ Q continue;
        if !skew_meld(node,old_q,&r.version) continue;
        if CAS(&Q,old_q,r.version) break;
    }
    *hp0 ← null; reset_memory_management();
}

Deq() {
    while true {
        removed_ptr ← 0; alloc_ptr ← 0;
        old_q ← Q;
        *hp0 ← old_q;
        if old_q ≠ Q continue;
        if !skew_deq(old_q) continue;
        if r.value = SKEW_EMPTY break;
        if CAS(&Q,old_q,r.version) break;
    }
    *hp0 ← null; reset_memory_management();
}

copy(q,p:*NodeType) : boolean {
    *hp1 ← q;
    if old_q ≠ Q return false;
    memcpy(p,q,sizeof(NodeType));
    *hp1 ← null;
    if removed_ptr = MAX_SIZE
        {error("exceeded size"); exit;}
    removed[removed_ptr++] ← q;
    return true;
}

skew_deq(q:*skew_type) : boolean {
    *left,*new_left,*right,buffer : skew_type;
    if q = null{r.value ← SKEW_EMPTY; return true;}
    if !copy(q,&buffer) return false;
    r.value ← buffer.value;
    left ← buffer.Child[0]; right ← buffer.Child[1];
    if left = null{r.version ← right; return true;}
    new_left ← allocate_node();
    if !copy(left,new_left) return false;
    return skew_meld(new_left,right,&r.version);
}

skew_meld(q,qq,*res:*skew_type) : boolean {
    if q = null{ *res ← qq; return true;}
    if qq = null{ *res ← q; return true;}
    p ← allocate_node();
    if !copy(qq,p) return false;
    if q^.Value < p^.Value
        { p1 ← q; p2 ← p;} else { p1 ← p; p2 ← q;}
    toggle ← p1^.toggle;
    p1^.toggle ← !toggle;
    *res ← p1;
    return skew_meld(p2,p1^.Child[toggle],&p1^.Child[toggle]);
}

allocate_node() : *skew_type {
    if alloc_ptr < avail_ptr return alloc[alloc_ptr++];
    if alloc_ptr = MAX_SIZE
        {error("exceeded size"); exit;}
    if alloc[avail_ptr+] ← AllocateNode() = null
        {error("out of memory"); exit;}
    return alloc[alloc_ptr++];
}

reset_memory_management() {
    // free unused allocated nodes
    for i ← alloc_ptr to avail_ptr-1 FreeNode(alloc[i]);
    avail_ptr ← 0;
    // delete removed nodes
    for i ← 0 to removed_ptr-1 DeleteNode(removed[i]);
}

```

Figure 5: SMR-compatible version of Herlihy’s [10] implementation of a lock-free skew heap.

6. CONCLUSIONS

The problem of arbitrary memory reuse for dynamic lock-free object has long obstructed the wide adoption of lock-free synchronization in multiprocessor applications, despite their clear inherent advantages of resilience when faced with thread failures and robust performance when faced with thread delays in comparison to lock-based synchronization.

Prior memory management methods for dynamic lock-free objects fall into four categories. First, those that restrict the reuse of deleted nodes to the same type and require the retention of the semantics and values of some fields indefinitely [12, 25]. Second, those that use the DCAS atomic operation which is not supported on any current system, and use reference counting that results in significant performance overhead [6]. Third, those that require special op-

erating system support for inspecting thread registers and private stack space [7]. Fourth, blocking methods that depend on actions by each thread, such as decrementing an aggregate counter or setting a timestamp, for allowing the reuse of potentially unbounded numbers of deleted nodes, thus allowing the failure or delay of one thread to cause an unbounded number of nodes to be not freed indefinitely [7, 8, 14].

In this paper we presented SMR, a practical and efficient solution for safe memory reclamation for dynamic lock-free objects. It allows arbitrary memory reuse, does not require special hardware as it uses only atomic reads and writes, does not require special operating system support, guarantees an upper bound on the number of deleted nodes not yet freed at all times, and is wait-free. Furthermore, it is only

logarithmically contention-sensitive. It guarantees constant amortized time for processing each deleted node until it is freed in the absence of contention, and only $O(\log p)$ time in the case of contention by p threads. Recent performance results [16] indicate that, in practice, the time overhead of using SMR is negligible. Also, it does not require any extra space per pointer or per node.

We demonstrated the ease of complying with the SMR compatibility conditions as we examined most known dynamic lock-free algorithms and methodologies and presented examples for the best known algorithms in each category. We found only two correct algorithms [5, 8] to be incompatible with SMR and freelists, and in each case we found similar or better algorithms for the same object that are compatible with all memory management methods [15, 16]. The performance and qualitative advantages of SMR and freelists make compatibility with them a crucial factor in evaluating currently known and future lock-free algorithms.

Even in cases where memory reclamation is not crucial, SMR still offers a lock-free solution for the ABA problem for pointers to dynamic nodes without the need for any extra space per pointer or per node, applicable to most dynamic lock-free algorithms. It is the only lock-free method that allows these algorithm to avoid the use of extra-width CAS and high-overhead reference counting.

Finally, this paper demonstrates that DCAS is not needed for lock-free memory reclamation.

7. REFERENCES

- [1] Ole Agesen, David L. Detlefs, Christine H. Flood, Alexander T. Garthwaite, Paul Martin, Nir N. Shavit, and Guy L. Steele Jr. DCAS-Based Concurrent Deques. In *Proceedings of the 12th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 137–146, July 2000.
- [2] James H. Anderson and Mark Moir. Universal Constructions for Large Objects. *IEEE Transactions on Parallel and Distributed Systems* 10(12): 1317–1332, December 1999.
- [3] Greg Barnes. A Method for Implementing Lock-Free Shared-Data Structures. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 261–270, June–July 1993.
- [4] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.
- [5] David L. Detlefs, Christine H. Flood, Alexander T. Garthwaite, Paul Martin, Nir N. Shavit, and Guy L. Steele Jr. Even Better DCAS-Based Concurrent Deques. In *Proceedings of the 14th International Symposium on Distributed Computing*, pages 59–73, October 2000.
- [6] David L. Detlefs, Paul A. Martin, Mark Moir, and Guy L. Steele Jr. Lock-Free Reference Counting. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing*, pages 190–199, August 2001.
- [7] Michael B. Greenwald. *Non-Blocking Synchronization and System Design*. Ph.D. thesis, Stanford University Technical Report STAN-CS-TR-99-1624, August 1999.
- [8] Timothy L. Harris. A Pragmatic Implementation of Non-Blocking Linked Lists. In *Proceedings of the 15th International Symposium on Distributed Computing*, pages 300–314, October 2001.
- [9] Maurice P. Herlihy. Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems* 13(1): 124–149, January 1991.
- [10] Maurice P. Herlihy. A Methodology for Implementing Highly Concurrent Objects. *ACM Transactions on Programming Languages and Systems* 15(5): 745–770, November 1993.
- [11] Maurice P. Herlihy and J. Eliot B. Moss. Lock-Free garbage Collection for Multiprocessors. In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 229–236, July 1991.
- [12] IBM System/370 Extended Architecture, Principles of Operation. IBM Publication No. SA22-7085, 1983.
- [13] Amos Israeli and Lihu Rappoport. Disjoint-Access-Parallel Implementations of Strong Shared Memory Primitives. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, pages 151–160, August 1994.
- [14] Paul E. McKenney and John D. Slingwine. Read-Copy Update: Using Execution History to Solve Concurrency Problems. In *Proceedings of the 11th International Conference on Parallel and Distributed Computing Systems*, October 1998.
- [15] Maged M. Michael. Dynamic Lock-Free Deque Algorithm Using Single-Address Double-Word CAS. Research Report RC 22401, IBM T. J. Watson Research Center, April 2002.
- [16] Maged M. Michael. High Performance Dynamic Lock-Free Hash Tables and List-Based Sets. In *Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures*, August 2002.
- [17] Maged M. Michael and Michael L. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, pages 267–275, May 1996.
- [18] Mark Moir. Practical Implementations of Non-Blocking Synchronization Primitives. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, pages 219–228, August 1997.
- [19] Mark Moir. Laziness pays! Using lazy Synchronization Mechanisms to Improve Non-Blocking Constructions. *Distributed Computing* 14(4): 193–204, 2001.
- [20] Sundeep Prakash, Yann-Hang Lee, and Theodore Johnson. A Nonblocking Algorithm for Shared Queues Using Compare-and-Swap. *IEEE Transactions on Computers* 43(5): 548–559, May 1994.
- [21] Nir Shavit and Dan Touitou. Software Transactional Memory. *Distributed Computing* 10(2): 99–116, 1997.
- [22] R. Kent Treiber. Systems Programming: Coping with Parallelism. Research Report RJ 5118, IBM Almaden Research Center, April 1986.
- [23] John Turek, Dennis Shasha, and Sundeep Prakash. Locking without Blocking: Making Lock Based Concurrent Data Structure Algorithms Nonblocking. In *Proceedings of the 11th ACM Symposium on Principles of Database Systems*, pages 212–222, June 1992.
- [24] John D. Valois. Implementing Lock-Free Queues. In *Proceedings of the 7th International Conference on Parallel and Distributed Computing Systems*, pages 64–69, October 1994.
- [25] John D. Valois. Lock-Free Linked Lists Using Compare-and-Swap. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 214–222, August 1995.

Scalable Lock-Free Dynamic Memory Allocation

Maged M. Michael

IBM Thomas J. Watson Research Center
P.O. Box 218, Yorktown Heights, NY 10598, USA
magedm@us.ibm.com

ABSTRACT

Dynamic memory allocators (malloc/free) rely on mutual exclusion locks for protecting the consistency of their shared data structures under multithreading. The use of locking has many disadvantages with respect to performance, availability, robustness, and programming flexibility. A lock-free memory allocator guarantees progress regardless of whether some threads are delayed or even killed and regardless of scheduling policies. This paper presents a completely lock-free memory allocator. It uses only widely-available operating system support and hardware atomic instructions. It offers guaranteed availability even under arbitrary thread termination and crash-failure, and it is immune to deadlock regardless of scheduling policies, and hence it can be used even in interrupt handlers and real-time applications without requiring special scheduler support. Also, by leveraging some high-level structures from Hoard, our allocator is highly scalable, limits space blowup to a constant factor, and is capable of avoiding false sharing. In addition, our allocator allows finer concurrency and much lower latency than Hoard. We use PowerPC shared memory multiprocessor systems to compare the performance of our allocator with the default AIX 5.1 libc malloc, and two widely-used multithread allocators, Hoard and Ptmalloc. Our allocator outperforms the other allocators in virtually all cases and often by substantial margins, under various levels of parallelism and allocation patterns. Furthermore, our allocator also offers the lowest contention-free latency among the allocators by significant margins.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features—*dynamic storage management*; D.4.1 [**Operating Systems**]: Process Management—*concurrency, deadlocks, synchronization, threads*.

General Terms: Algorithms, Performance, Reliability.

Keywords: malloc, lock-free, async-signal-safe, availability.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLDI'04, June 9–11, 2004, Washington, DC, USA.
Copyright 2004 ACM 1-58113-807-5/04/0006 ...\$5.00.

1. INTRODUCTION

Dynamic memory allocation functions, such as malloc and free, are heavily used by a wide range of important multithreaded applications, from commercial database and web servers to data mining and scientific applications. In order to be safe under multithreading (MT-safe), current allocators employ mutual exclusion locking in a variety of ways, ranging from the use of a single lock wrapped around single-thread malloc and free, to the distributed use of locks in order to allow more concurrency and higher scalability. The use of locking causes many problems and limitations with respect to performance, availability, robustness, and programming flexibility.

A desirable but challenging alternative approach for achieving MT-safety is lock-free synchronization. A shared object is lock-free (nondblocking) if it guarantees that whenever a thread executes some finite number of steps, at least one operation on the object by some thread must have made progress during the execution of these steps. Lock-free synchronization implies several inherent advantages:

Immunity to deadlock: By definition, a lock-free object must be immune to deadlock and livelock. Therefore, it is much simpler to design deadlock-free systems when all or some of their components are lock-free.

Async-signal-safety: Due to the use of locking in current implementations of malloc and free, they are not considered *async-signal-safe* [9], i.e., signal handlers are prohibited from using them. The reason for this prohibition is that if a thread receives a signal while holding a user-level lock in the allocator, and if the signal handler calls the allocator, and in the process it must acquire the same lock held by the interrupted thread, then the allocator becomes deadlocked due to circular dependence. The signal handler waits for the interrupted thread to release the lock, while the thread cannot resume until the signal handler completes. Masking interrupts or using kernel-assisted locks in malloc and free is too costly for such heavily-used functions. In contrast, a completely lock-free allocator is capable of being *async-signal-safe* without incurring any performance cost.

Tolerance to priority inversion: Similarly, in real-time applications, user-level locking is susceptible to deadlock due to priority inversion. That is, a high priority thread can be waiting for a user-level lock to be released by a lower priority thread that will not be scheduled until the high priority thread completes its task. Lock-free synchronization guarantees progress regardless of scheduling policies.

Kill-tolerant availability: A lock-free object must be immune to deadlock even if any number of threads are killed while operating on it. Accordingly, a lock-free object must offer guaranteed availability regardless of arbitrary thread

```
CAS(addr,expval,newval) atomically do
  if (*addr == expval) {
    *addr = newval;
    return true;
  } else
    return false;
```

Figure 1: Compare-and-Swap.

termination or crash-failure. This is particularly useful for servers that require a high level of availability, but can tolerate the infrequent loss of tasks or servlets that may be killed by the server administrator in order to relieve temporary resource shortages.

Preemption-tolerance: When a thread is preempted while holding a mutual exclusion lock, other threads waiting for the same lock either spin uselessly, possibly for the rest of their time slices, or have to pay the performance cost of yielding their processors in the hope of giving the lock holder an opportunity to complete its critical section. Lock-free synchronization offers preemption-tolerant performance, regardless of arbitrary thread scheduling.

It is clear that it is desirable for memory allocators to be completely lock-free. The question is how, and more importantly, how to be completely lock-free and (1) offer good performance competitive with the best lock-based allocators (i.e., low latency, scalability, avoiding false sharing, constant space blowup factor, and robustness under contention and preemption), (2) using only widely-available hardware and OS support, and (3) without making trivializing assumptions that make lock-free progress easy, but result in unacceptable memory consumption or impose unreasonable restrictions.

For example, it is trivial to design a wait-free allocator with pure per-thread private heaps. That is, each thread allocates from its own heap and also frees blocks to its own heap. However, this is hardly an acceptable general-purpose solution, as it can lead to unbounded memory consumption (e.g., under a producer-consumer pattern [3]), even when the program’s memory needs are in fact very small. Other unacceptable characteristics include the need for initializing large parts of the address space, putting an artificial limit on the total size or number of allocatable dynamic blocks, or restricting beforehand regions of the address to specific threads or specific block sizes. An acceptable solution must be general-purpose and space efficient, and should not impose artificial limitations on the use of the address space.

In this paper we present a completely lock-free allocator that offers excellent performance, uses only widely-available hardware and OS support, and is general-purpose.

For constructing our lock-free allocator and with only the simple atomic instructions supported on current mainstream processor architectures as our memory access tools, we break down malloc and free into fine atomic steps, and organize the allocator’s data structures such that if any thread is delayed arbitrarily (or even killed) at any point, then any other thread using the allocator will be able to determine enough of the state of the allocator to proceed with its own operation without waiting for the delayed thread to resume.

By leveraging some high-level structures from Hoard [3], a scalable lock-based allocator, we achieve concurrency between operations on multiple processors, avoid inducing false sharing, and limit space blowup to a constant factor. In addition, our allocator uses a simpler and finer grained or-

```
AtomicInc(addr)
  do {
    oldval = *addr;
    newval = oldval+1;
  } until CAS(addr,oldval,newval);
```

Figure 2: Atomic increment using CAS.

ganization that allows more concurrency and lower latency than Hoard.

We use POWER3 and POWER4 shared memory multiprocessors to compare the performance of our allocator with the default AIX 5.1 libc malloc, and two widely-used lock-based allocators with mechanisms for scalable performance, Hoard [3] and Ptmalloc [6]. The experimental performance results show that not only is our allocator competitive with some of the best lock-based allocators, but also that it outperforms them, and often by substantial margins, in virtually all cases including under various levels of parallelism and various sharing patterns, and also offers the lowest contention-free latency.

The rest of the paper is organized as follows. In Section 2, we discuss atomic instructions and related work. Section 3 describes the new allocator in detail. Section 4 presents our experimental performance results. We conclude the paper with Section 5.

2. BACKGROUND

2.1 Atomic Instructions

Current mainstream processor architectures support either Compare-and-Swap (CAS) or the pair Load-Linked and Store-Conditional (LL/SC). Other weaker instructions, such as Fetch-and-Add and Swap, may be supported, but in any case they are easily implemented using CAS or LL/SC.

CAS was introduced on the IBM System 370 [8]. It is supported on Intel (IA-32 and IA-64) and Sun SPARC architectures. In its simplest form, it takes three arguments: the address of a memory location, an expected value, and a new value. If the memory location is found to hold the expected value, the new value is written to it, atomically. A Boolean return value indicates whether the write occurred. If it returns true, it is said to succeed. Otherwise, it is said to fail. Figure 1 shows the semantics of CAS.

LL and SC are supported on the PowerPC, MIPS, and Alpha architectures. LL takes one argument: the address of a memory location, and returns its contents. SC takes two arguments: the address of a memory location and a new value. Only if no other thread has written the memory location since the current thread last read it using LL, the new value is written to the memory location, atomically. A Boolean return value indicates whether the write occurred. Similar to CAS, SC is said to succeed or fail if it returns true or false, respectively. For architectural reasons, current architectures that support LL/SC do not allow the nesting or interleaving of LL/SC pairs, and infrequently allow SC to fail spuriously, even if the target location was never written since the last LL. These spurious failures happen, for example, if the thread was preempted or a different location in the same cache line was written by another processor.

For generality, we present the algorithms in this paper using CAS. If LL/SC are supported rather than CAS, then `CAS(addr,expval,newval)` can be simulated in a lock-free

```
manner as follows: {do {if (LL(addr)!=expval) return  
false;} until SC(addr,newval); return true;}.
```

Support for CAS and restricted LL/SC on aligned 64-bit blocks is available on both 32-bit and 64-bit architectures, e.g., CMPXCHG8 on IA-32. However, support for CAS or LL/SC on wider block sizes is generally not available even on 64-bit architectures. Therefore, we focus our presentation of the algorithms on 64-bit mode, as it is the more challenging case while 32-bit mode is simpler.

For a very simple example of lock-free synchronization, Figure 2 shows the classic lock-free implementation of Atomic-Increment using CAS [8]. Note that if a thread is delayed at any point while executing this routine, other active threads will be able to proceed with their operations without waiting for the delayed thread, and every time a thread executes a full iteration of the loop, some operation must have made progress. If the CAS succeeds, then the increment of the current thread has taken effect. If the CAS fails, then the value of the counter must have changed during the loop. The only way the counter changes is if a CAS succeeds. Then, some other thread’s CAS must have succeeded during the loop and hence that other thread’s increment must have taken effect.

2.2 Related Work

The concept of lock-free synchronization goes back more than two decades. It is attributed to early work by Lamport [12] and to the motivating basis for introducing the CAS instruction in the IBM System 370 documentation [8]. The impossibility and universality results of Herlihy [7] had significant influence on the theory and practice of lock-free synchronization, by showing that atomic instructions such as CAS and LL/SC are more powerful than others such as Test-and-Set, Swap, and Fetch-and-Add, in their ability to provide lock-free implementations of arbitrary object types. In other publications [17, 19], we review practical lock-free algorithms for dynamic data structures in light of recent advances in lock-free memory management.

Wilson et. al. [23] present a survey of sequential memory allocation. Berger [2, 3] presents an overview of concurrent allocators, e.g., [4, 6, 10, 11, 13]. In our experiments, we compare our allocator with two widely-used malloc replacement packages for multiprocessor systems, Ptmalloc and Hoard. We also leverage some scalability-enabling high-level structures from Hoard.

Ptmalloc [6], developed by Wolfram Gloger and based on Doug Lea’s dlmalloc sequential allocator [14], is part of GNU glibc. It uses multiple arenas in order to reduce the adverse effect of contention. The granularity of locking is the arena. If a thread executing malloc finds an arena locked, it tries the next one. If all arenas are found to be locked, the thread creates a new arena to satisfy its malloc and adds the new arena to the main list of arenas. To improve locality and reduce false sharing, each thread keeps thread-specific information about the arena it used in its last malloc. When a thread frees a chunk (block), it returns the chunk to the arena from which the chunk was originally allocated, and the thread must acquire that arena’s lock.

Hoard [2, 3], developed by Emery Berger, uses multiple processor heaps in addition to a global heap. Each heap contains zero or more superblocks. Each superblock contains one or more blocks of the same size. Statistics are maintained individually for each superblock as well as collectively for the superblocks of each heap. When a processor heap is found to have too much available space, one of its su-

perblocks is moved to the global heap. When a thread finds that its processor heap does not have available blocks of the desired size, it checks if any superblocks of the desired size are available in the global heap. Threads use their thread ids to decide which processor heap to use for malloc. For free, a thread must return the block to its original superblock and update the fullness statistics for the superblock as well as the heap that owns it. Typically, malloc and free require one and two lock acquisitions, respectively.

Dice and Garthwaite [5] propose a partly lock-free allocator. The allocator requires special operating system support, which makes it not readily portable across operating systems and programming environments. In the environment for their allocator, the kernel monitors thread migration and preemption and posts upcalls to user-mode. When a thread is scheduled to run, the kernel posts the CPU id of the processor that the thread is to run on during its upcoming time slice. The kernel also saves the user-mode instruction pointer in a thread-specific location and replaces it with the address of a special notification routine that will be the first thing the thread executes when it resumes. The notification routine checks if the thread was in a critical section when it was preempted. If so, the notification routine passes control to the beginning of the critical section instead of the original instruction pointer, so that the thread can retry its critical section. The allocator can apply this mechanism only to CPU-specific data. So, it is only used for the CPU’s local heap. For all other operations, such as freeing a block that belongs to a remote CPU heap or any access to the global heap, mutual exclusion locks are used. The allocator is not completely lock-free, and hence—without additional special support from the kernel—it is susceptible to deadlock under arbitrary thread termination or priority inversion.

3. LOCK-FREE ALLOCATOR

This section describes our lock-free allocator in detail. Without loss of generality we focus on the case of a 64-bit address space. The 32-bit case is simpler, as 64-bit CAS is supported on 32-bit architectures.

3.1 Overview

First, we start with the general structure of the allocator. Large blocks are allocated directly from the OS and freed directly to the OS. For smaller block sizes, the heap is composed of large superblocks (e.g., 16 KB). Each superblock is divided into multiple equal-sized blocks. Superblocks are distributed among size classes based on their block sizes. Each size class contains multiple processor heaps proportional to the number of processors in the system. A processor heap contains at most one active superblock. An active superblock contains one or more blocks available for reservation that are guaranteed to be available to threads that reach them through the header of the processor heap. Each superblock is associated with a descriptor. Each allocated block contains a prefix (8 bytes) that points to the descriptor of its superblock. On the first call to malloc, the static structures for the size classes and processor heaps (about 16 KB for a 16 processor machine) are allocated and initialized in a lock-free manner.

Malloc starts by identifying the appropriate processor heap, based on the requested block size and the identity of the calling thread. Typically, the heap already has an active superblock with blocks available for reservation. The thread atomically reads a pointer to the descriptor of the active superblock and reserves a block. Next, the thread atomically

```

// Superblock descriptor structure
typedef anchor : // fits in one atomic block
    unsigned avail:10, count:10, state:2, tag:42;
// state codes ACTIVE=0 FULL=1 PARTIAL=2 EMPTY=3

typedef descriptor :
    anchor Anchor;
    descriptor* Next;
    void* sb; // pointer to superblock
    procheap* heap; // pointer to owner procheap
    unsigned sz; // block size
    unsigned maxcount; // superblock size/sz

```

```

// Processor heap structure
typedef active : unsigned ptr:58, credits:6;
typedef procheap :
    active Active; // initially NULL
    descriptor* Partial; // initially NULL
    sizeclass* sc; // pointer to parent sizeclass

// Size class structure
typedef sizeclass :
    descList Partial; // initially empty
    unsigned sz; // block size
    unsigned sbsize; // superblock size

```

Figure 3: Structures.

pops a block from that superblock and updates its descriptor. A typical free pushes the freed block into the list of available blocks of its original superblock by atomically updating its descriptor. We discuss the less frequent more complicated cases below when describing the algorithms in detail.

3.2 Structures and Algorithms

For the most part, we provide detailed (C-like) code for the algorithms, as we believe that it is essential for understanding lock-free algorithms, unlike lock-based algorithms where sequential components protected by locks can be described clearly using high-level pseudocode.

3.2.1 Structures

Figure 3 shows the details of the above mentioned structures. The `Anchor` field in the superblock descriptor structure contains subfields that can be updated together atomically using CAS or LL/SC. The subfield `avail` holds the index of the first available block in the superblock, `count` holds the number of unreserved blocks in the superblock, `state` holds the state of the superblock, and `tag` is used to prevent the ABA problem as discussed below.

The `Active` field in the processor heap structure is primarily a pointer to the descriptor of the active superblock owned by the processor heap. If the value of `Active` is not `NULL`, it is guaranteed that the active superblock has at least one block available for reservation. Since the addresses of superblock descriptors can be guaranteed to be aligned to some power of 2 (e.g., 64), as an optimization, we can carve a `credits` subfield to hold the number of blocks available for reservation in the active superblock less one. That is, if the value of `credits` is n , then the active superblock contains $n+1$ blocks available for reservation through the `Active` field. Note that the number of blocks in a superblock is not limited to the maximum reservations that can be held in the `credits` subfield. In a typical malloc operation (i.e., when `Active ≠ NULL` and `credits > 0`), the thread reads `Active` and then atomically decrements `credits` while validating that the active superblock is still valid.

3.2.2 Superblock States

A superblock can be in one of four states: `ACTIVE`, `FULL`, `PARTIAL`, or `EMPTY`. A superblock is `ACTIVE` if it is the active superblock in a heap, or if a thread intends to try to install it as such. A superblock is `FULL` if all its blocks are either allocated or reserved. A superblock is `PARTIAL` if it is not `ACTIVE` and contains unreserved available blocks. A superblock is `EMPTY` if all its blocks are free and it is not `ACTIVE`. An `EMPTY` superblock is safe to be returned to the OS if desired.

3.2.3 Malloc

Figure 4 shows the `malloc` algorithm. The outline of the algorithm is as follows. If the block size is large, then the block is allocated directly from the OS and its prefix is set to indicate the block’s size. Otherwise, the appropriate heap is identified using the requested block size and the id of the requesting thread. Then, the thread tries the following in order until it allocates a block: (1) Allocate a block from the heap’s active superblock. (2) If no active superblock is found, try to allocate a block from a `PARTIAL` superblock. (3) If none are found, allocate a new superblock and try to install it as the active superblock.

Malloc from Active Superblock

The vast majority of malloc requests are satisfied from the heap’s active superblock as shown in the `MallocFromActive` routine in Figure 4. The routine consists of two main steps. The first step (lines 1–6) involves reading a pointer to the active superblock and then atomically decrementing the number of available credits—thereby reserving a block—while validating that the active superblock is still valid. Upon the success of CAS in line 6, the thread is guaranteed that a block in the active superblock is reserved for it.

The second step of `MallocFromActive` (lines 7–18) is primarily a lock-free pop from a LIFO list [8]. The thread reads the index of the first block in the list from `Anchor.avail` in line 8, then it reads the index of the next block in line 10,¹ and finally in line 18 it tries to swing the head pointer (i.e., `Anchor.avail`) atomically to the next block, while validating that at that time what it “thinks” to be the first two indexes in the list (i.e., `oldanchor.avail` and `next`) are indeed the first two indexes in the list, and hence in effect popping the first available block from the list.

Validating that the CAS in line 18 succeeds only if `Anchor.avail` is equal to `oldanchor.avail` follows directly from the semantics of CAS. However, validating that at that time `*addr=next` is more subtle and without the `tag` subfield is susceptible to the ABA problem [8, 19]. Consider the case where in line 8 thread X reads the value A from `Anchor.avail` and in line 10 reads the value B from `*addr`. After line 10, X is delayed and some other thread or threads pop (i.e., allocate) block A then block B and then push (i.e., free) some block C and then block A back in the list. Later, X resumes and executes the CAS in line 18. Without the `tag` subfield (for simplicity ignore the `count` subfield), the CAS would find `Anchor` equal to `oldanchor` and succeeds where

¹This is correct even if there is no next block, because in such a case no subsequent malloc will target this superblock before one of its blocks is freed.

```

void* malloc(sz) {
    // Use sz and thread id to find heap.
1   heap = find_heap(sz);
2   if (!heap) // Large block
3       Allocate block from OS and return its address.
    while(1) {
4       addr = MallocFromActive(heap);
5       if (addr) return addr;
6       addr = MallocFromPartial(heap);
7       if (addr) return addr;
8       addr = MallocFromNewSB(heap);
9       if (addr) return addr;
} }

void* MallocFromActive(heap) {
    do { // First step: reserve block
1   newactive = oldactive = heap->Active;
2   if (!oldactive) return NULL;
3   if (oldactive.credits == 0)
4       newactive = NULL;
        else
5       newactive.credits--;
6   } until CAS(&heap->Active,oldactive,newactive);
// Second step: pop block
7 desc = mask_credits(oldactive);
    do {
        // state may be ACTIVE, PARTIAL or FULL
8   newanchor = oldanchor = desc->Anchor;
9   addr = desc->sb+oldanchor.avail*desc->sz;
10  next = *(unsigned*)addr;
11  newanchor.avail = next;
12  newanchor.tag++;
13  if (oldactive.credits == 0) {
        // state must be ACTIVE
14      if (oldanchor.count == 0)
15          newanchor.state = FULL;
        else {
16          morecredits =
            min(oldanchor.count,MAXCREDITS);
17          newanchor.count -= morecredits;
        }
    }
18 } until CAS(&desc->Anchor,oldanchor,newanchor);
19 if (oldactive.credits==0 && oldanchor.count>0)
20     UpdateActive(heap,desc,morecredits);
21 *addr = desc; return addr+EIGHTBYTES;
}

UpdateActive(heap,desc,morecredits) {
1   newactive = desc;
2   newactive.credits = morecredits-1;
3   if CAS(&heap->Active,NULL,newactive) return;
    // Someone installed another active sb
    // Return credits to sb and make it partial
    do {
4       newanchor = oldanchor = desc->Anchor;
5       newanchor.count += morecredits;
6       newanchor.state = PARTIAL;
7   } until CAS(&desc->Anchor,oldanchor,newanchor);
8   HeapPutPartial(desc);
}

void* MallocFromPartial(heap) {
retry:
1   desc = HeapGetPartial(heap);
2   if (!desc) return NULL;
3   desc->heap = heap;
    do { // reserve blocks
4       newanchor = oldanchor = desc->Anchor;
5       if (oldanchor.state == EMPTY) {
            DescRetire(desc); goto retry;
        }
        // oldanchor state must be PARTIAL
        // oldanchor count must be > 0
7   morecredits =
        min(oldanchor.count-1,MAXCREDITS);
8   newanchor.count -= morecredits+1;
9   newanchor.state =
        (morecredits > 0) ? ACTIVE : FULL;
10 } until CAS(&desc->Anchor,oldanchor,newanchor);
    do { // pop reserved block
11       newanchor = oldanchor = desc->Anchor;
12       addr = desc->sb+oldanchor.avail*desc->sz;
13       newanchor.avail = *(unsigned*)addr;
14       newanchor.tag++;
15   } until CAS(&desc->Anchor,oldanchor,newanchor);
16   if (morecredits > 0)
17       UpdateActive(heap,desc,morecredits);
18   *addr = desc; return addr+EIGHTBYTES;
}

descriptor* HeapGetPartial(heap) {
    do {
1   desc = heap->Partial;
2   if (desc == NULL)
3       return ListGetPartial(heap->sc);
4   } until CAS(&heap->Partial,desc,NULL);
5   return desc;
}

void* MallocFromNewSB(heap) {
1   desc = DescAlloc();
2   desc->sb = AllocNewSB(heap->sc->sbsize);
3   Organize blocks in a linked list starting with index 0.
4   desc->heap = heap;
5   desc->Anchor.avail = 1;
6   desc->sz = heap->sc->sz;
7   desc->maxcount = heap->sc->sbsize/desc->sz;
8   newactive = desc;
9   newactive.credits =
        min(desc->maxcount-1,MAXCREDITS)-1;
10  desc->Anchor.count =
        (desc->maxcount-1)-(newactive.credits+1);
11  desc->Anchor.state = ACTIVE;
12  memory fence.
13  if CAS((&heap->Active,NULL,newactive) {
14      addr = desc->sb;
15      *addr = desc; return addr+EIGHTBYTES;
    } else {
16        Free the superblock desc->sb.
17        DescRetire(desc); return NULL;
    }
}

```

Figure 4: Malloc.

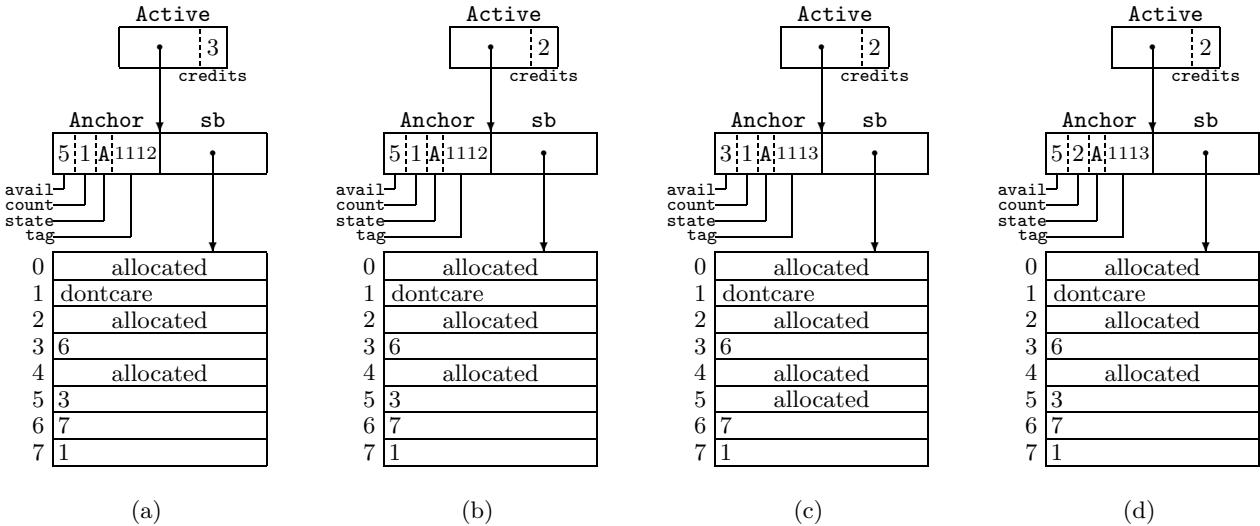


Figure 5: An example of a typical malloc and free from an active superblock. In configuration (a), the active superblock contains 5 available blocks organized in a linked list [5,3,6,7,1], four of which are available for reservation as indicated by `Active.credits=3`. In the first step of malloc, a block is reserved by decrementing `Active.credits`, resulting in configuration (b). In the second step of malloc, block 5 is popped, resulting in configuration (c). Free pushes the freed block (block 5) resulting in configuration (d).

it should not, as the new head of the free list would become block *B* which is actually not free. Without the tag subfield, *X* is unable to detect that the value of `Anchor.avail` changed from *A* to *B* and finally back to *A* again (hence the name ABA). To prevent this problem for the `Anchor` field, we use the classic IBM tag mechanism [8]. We increment the tag subfield (line 12) on every pop and validate it atomically with the other subfields of `Anchor`. Therefore, in the above mentioned scenario, when the tag is used, the CAS fails—as it should—and *X* starts over from line 8. The tag must have enough bits to make full wraparound practically impossible in a short time. For an absolute solution for the ABA problem, an efficient implementation of ideal LL/SC—which inherently prevents the ABA problem—using pointer-sized CAS can be used [18, 19].

In lines 13–17, the thread checks if it has taken the last credit in `Active.credit`. If so, it checks if the superblock has more available blocks, either because `maxcount` is larger than `MAXCREDITS` or because blocks were freed. If more blocks are available, the thread reserves as many as possible (lines 16 and 17). Otherwise, it declares the superblock `FULL` (line 15). The reason for doing that is that `FULL` superblocks are not pointed to by any allocator structures, so the first thread to free a block back to a `FULL` superblock needs to know that, in order to take responsibility for linking it back to the allocator structures.

If the thread has taken credits, it tries to update `Active` by executing `UpdateActive`. There is no risk of more than one thread trying to take credits from the same superblock at the same time. Only the thread that sets `Active` to `NULL` in line 6 can do that. Other concurrent threads find `Active` either with `credits>0` or not pointing to `desc` at all.

Finally the thread stores `desc` (i.e., the address of the descriptor of the superblock from which the block was allocated) into the prefix of the newly allocated block (line 21), so that when the block is subsequently freed, `free` can determine from which superblock it was originally allocated. Each block includes an 8 byte prefix (overhead).

Note that, after a thread finds `Active.credits>0` and after the success of the CAS in line 6 and before the thread proceeds to a successful CAS in line 18, it is possible that the “active” superblock might have become `FULL` if all available blocks were reserved, `PARTIAL`, or even the `ACTIVE` superblock of a different processor heap (but must be the same size class). However, it cannot be `EMPTY`. These possibilities do not matter to the original thread. After the success of the CAS in line 6, the thread is guaranteed a block from this specific superblock, and all it need do is pop a block from the superblock and leave the superblock’s `Anchor.state` unchanged. Figure 5 shows a typical malloc and free from an active superblock.

Updating Active Credits

Typically, when the routine `UpdateActive` in Figure 4 is called, it ends with the success of the CAS operation in line 3 that reinstalls `desc->sb` as the active superblock for `heap` with one or more credits. However, it is possible that after the current thread had set `heap->Active` to `NULL` (line 6 of `MallocFromActive`), some other thread installed a new superblock. If so, the current thread must return the credits, indicate that the superblock is `PARTIAL`, and make the superblock available for future use in line 8 by calling `HeapPutPartial` (described below).

Malloc from Partial Superblock

The thread calls `MallocFromPartial` in Figure 4 if it finds `Active=NULL`. The thread tries to get a `PARTIAL` superblock by calling `HeapGetPartial`. If it succeeds, it tries to reserve as many blocks—including one for itself—from the superblock’s descriptor. Upon the success of CAS in line 10, the thread is guaranteed to have reserved one or more blocks. It then proceeds in lines 11–15 to pop its reserved block, and if it has reserved more, it deposits the additional credits in `Active` by calling `UpdateActive`.

In `HeapGetPartial`, the thread first tries to pop a superblock from the `Partial` slot associated with the thread’s

processor heap. If `Partial=NULL`, then the thread checks the Partial list associated with the size class as described in Section 3.2.6.

Malloc from New Superblock

If the thread does not find any `PARTIAL` superblocks, it calls `MallocFromNewSB` in Figure 4. The thread allocates a descriptor by calling `DescAlloc` (line 1), allocates a new superblock, and sets its fields appropriately (lines 2–11). Finally, it tries to install it as the active superblock in `Active` using CAS in line 13. If the CAS fails, the thread deallocates the superblock and retires the descriptor (or alternatively, the thread can take the block, return the credits to the superblock, and install the superblock as `PARTIAL`). The failure of CAS in line 13 implies that `heap->Active` is no longer `NULL`, and therefore a new active superblock must have been installed by another thread. In order to avoid having too many `PARTIAL` superblocks and hence cause unnecessary external fragmentation, we prefer to deallocate the superblock rather than take a block from it and keep it as `PARTIAL`.

On systems with memory consistency models [1] weaker than sequential consistency, where the processors might execute and observe memory accesses out of order, `fence` instructions are needed to enforce the ordering of memory accesses. The memory fence instruction in line 12 serves to ensure that the new values of the descriptor fields are observed by other processors before the CAS in line 13 can be observed. Otherwise, if the CAS succeeds, then threads running on other processors may read stale values from the descriptor.²

3.2.4 Free

Figure 6 shows the `free` algorithm. Large blocks are returned directly to the OS. The free algorithm for small blocks is simple. It primarily involves pushing the freed block into its superblock’s available list and adjusting the superblock’s state appropriately.

The instruction fence in line 14 is needed to ensure that the read in line 13 is executed before the success of the CAS in line 18. The memory fence in line 17 is needed to ensure that the write in line 8 is observed by other processors no later than the CAS in line 18 is observed.

If a thread is the first to return a block to a `FULL` superblock, then it takes responsibility for making it `PARTIAL` by calling `HeapPutParial`, where it atomically swaps the superblock with the prior value in the Partial slot of the heap that last owned the superblock. If the previous value of `heap->Partial` is not `NULL`, i.e., it held a partial superblock, then the thread puts that superblock in the partial list of the size class as described in Section 3.2.6.

If a thread frees the last allocated block in a superblock, then it takes responsibility for indicating that the superblock is `EMPTY` and frees it. The thread then tries to retire the associated descriptor. If the descriptor is in the Partial slot of a processor heap, a simple CAS will suffice to remove it. Otherwise, the descriptor may be in the Partial list of the size class (possibly in the middle). We discuss this case in Section 3.2.6.

²Due to the variety in memory consistency models and fence instructions among architectures, it is customary for concurrent algorithms presented in the literature to ignore them. In this paper, we opt to include fence instructions in the code, but for clarity we assume a typical PowerPC-like architecture. However, different architectures—including future ones—may use different consistency models.

```

free(ptr) {
1  if (!ptr) return;
2  ((void**)ptr)--; // get prefix
3  desc = *(descriptor**)ptr;
4  if (large_block_bit_set(desc))
    // Large block - desc holds sz+1
5  { Return block to OS. return; }
6  sb = desc->sb;
  do {
7    newanchor = oldanchor = desc->Anchor;
8    *(unsigned*)ptr = oldanchor.avail;
9    newanchor.avail = (ptr-sb)/desc->sz;
10   if (oldanchor.state == FULL)
11     newanchor.state = PARTIAL;
12   if (oldanchor.count==desc->maxcount-1) {
13     heap = desc->heap;
14     instruction fence.
15     newanchor.state = EMPTY;
16   } else
17     newanchor.count++;
18   memory fence.
19 } until CAS(&desc->Anchor,oldanchor,newanchor);
20 if (newanchor.state == EMPTY) {
21   Free the superblock sb.
22   RemoveEmptyDesc(heap,desc);
23 } elseif (oldanchor.state == FULL)
24   HeapPutPartial(desc);
}

HeapPutPartial(desc) {
1  do { prev = desc->heap->Partial;
2    } until CAS(&desc->heap->Partial,prev,desc);
3  if (prev) ListPutPartial(prev);
}

RemoveEmptyDesc(heap,desc) {
1  if CAS(&heap->Partial,desc,NULL)
2    DescRetire(desc);
3  else ListRemoveEmptyDesc(heap->sc);
}

```

Figure 6: Free.

3.2.5 Descriptor List

Figure 7 shows the `DescAlloc` and `DescRetire` routines. In `DescAlloc`, the thread first tries to pop a descriptor from the list of available descriptors (lines 3–4). If not found, the thread allocates a superblock of descriptors, takes one descriptor, and tries to install the rest in the global available descriptor list. In order to avoid unnecessarily allocating too many descriptors, if the thread finds that some other thread has already made some descriptors available (i.e., the CAS in line 8 fails), then it returns the superblock to the OS and starts over in line 1, with the hope of finding an available descriptor. `DescRetire` is a straightforward lock-free push that follows the classic freelist push algorithm [8].

As mentioned above in the case of the pop operation in the `MallocFromActive` routine, care must be taken that CAS does not succeed where it should not due to the ABA problem. We indicate this in line 4, by using the term `SafeCAS` (i.e., ABA-safe). We use the hazard pointer methodology [17, 19]—which uses only pointer-sized instructions—in order to prevent the ABA problem for this structure.

```

descriptor* DescAvail; // initially NULL

descriptor* DescAlloc() {
    while (1) {
1      desc = DescAvail;
2      if (desc) {
3          next = desc->Next;
4          if SafeCAS(&DescAvail,desc,next) break;
5      } else {
6          desc = AllocNewSB(DESCSBSIZE);
7          Organize descriptors in a linked list.
8          memory fence.
9          if CAS(&DescAvail,NULL,desc->Next)) break;
        Free the superblock desc.
    }
}
10 return desc;
}

DescRetire(desc) {
    do {
1      oldhead = DescAvail;
2      desc->Next = oldhead;
3      memory fence.
4    } until CAS(&DescAvail,oldhead,desc);
}

```

Figure 7: Descriptor allocation.

In the current implementation, superblock descriptors are not reused as regular blocks and cannot be returned to the OS. This is acceptable as descriptors constitute on average less than 1% of allocated memory. However, if desired, space for descriptors can be reused arbitrarily or returned to the OS, by organizing descriptors in a similar manner to regular blocks and maintaining special descriptors for superblocks of descriptor, with virtually no effect on average performance whether contention-free or under high contention. This can be applied on as many levels as desired, such that at most 1% of 1%—and so on—of allocated space is restricted from being reused arbitrarily or returned to the OS.

Similarly, in order to reduce the frequency of calls to `mmap` and `munmap`, we allocate superblocks (e.g., 16 KB) in batches of (e.g., 1 MB) *hyperblocks* (superblocks of superblocks) and maintain descriptors for such hyperblocks, allowing them eventually to be returned to the OS. We organize the descriptor `Anchor` field in a slightly different manner, such that superblocks are not written until they are actually used, thus saving disk swap space for unused superblocks.

3.2.6 Lists of Partial Superblocks

For managing the list of partial superblocks associated with each size class, we need to provide three functions: `ListGetPartial`, `ListPutPartial`, and `ListRemoveEmptyDesc`. The goal of the latter is to ensure that empty descriptors are eventually made available for reuse, and not necessarily to remove a specific empty descriptor immediately.

In one possible implementation, the list is managed in a LIFO manner, with the possibility of removing descriptors from the middle of the list. The simpler version in [19] of the lock-free linked list algorithm in [16] can be used to manage such a list. `ListPutPartial` inserts `desc` at the head of the list. `ListGetPartial` pops a descriptor from the head of the

list. `ListRemoveEmptyDesc` traverses the list until it removes some empty descriptor or reaches the end of the list.

Another implementation, which we prefer, manages the list in a FIFO manner and thus reduces the chances of contention and false sharing. `ListPutPartial` enqueues `desc` at the tail of the list. `ListGetPartial` dequeues a descriptor from the head of the list. `ListRemoveEmptyDesc` keeps dequeuing descriptors from the head of the list until it dequeues a non-empty descriptor or reaches the end of the list. If the function dequeues a non-empty descriptor, then it reenqueues the descriptor at the tail of the list. By removing any one empty descriptor or moving two non-empty descriptor from the head of the list to its end, we are guaranteed that no more than half the descriptors in the list are left empty. We use a version of the lock-free FIFO queue algorithm in [20] with optimized memory management for the purposes of the new allocator.

For preventing the ABA problem for pointer-sized variables in the above mentioned list implementations, we cannot use IBM ABA-prevention tags (such as in `Anchor.tag`), instead we use ideal LL/SC constructions using pointer-sized CAS [18]. Note that these constructions as described in [18] use memory allocation, however a general-purpose malloc is not needed. In our implementation we allocate such blocks in a manner similar but simpler than allocating descriptors.

Note that in our allocator, unlike Hoard [3], we do not maintain fullness classes or keep statistics about the fullness of processor heaps and we are quicker to move partial superblocks to the partial list of the size class. This simplicity allows lower latency and lower fragmentation. But, one concern may be that this makes it more likely for blocks to be freed to a superblock in the size class partial lists. However, this is not a disadvantage at all, unlike Hoard [3] and also [5] where this can cause contention on the *global* heap's lock. In our allocator, freeing a block into such a superblock does not cause any contention with operations on other superblocks, and in general is no more complex or less efficient than freeing a block into a superblock that is in the thread's own processor heap. Another possible concern is that by moving partial superblocks out of the processor heap *too quickly*, contention and false sharing may arise. This is why we use a most-recently-used Partial slot (multiple slots can be used if desired) in the processor heap structure, and use a FIFO structure for the size class partial lists.

4. EXPERIMENTAL RESULTS

In this section, we describe our experimental performance results on two PowerPC multiprocessor systems. The first system has sixteen 375 MHz POWER3-II processors, with 24 GB of memory, 4 MB second level caches. The second system has eight 1.2 GHz POWER4+ processors, with 16 GB of memory, 32 MB third level caches. Both systems run AIX 5.1. We ran experiments on both systems. The results on the POWER3 system (with more processors) provided more insights into the allocators' scalability and ability to avoid false sharing and contention. The results on the POWER4 system provided insights into the contention-free latency of the allocators and contention-free synchronization costs on recent processor architectures.

We compare our allocator with the default AIX libc malloc,³ Hoard [3] version 3.0.2 (December 2003), and Ptmalloc-

³Our observations on the default libc malloc are based on external experimentation only and are not based on any knowledge of its internal design.

	375 MHz POWER3-II			1.2 GHz POWER4+		
	New	Hoard	Ptmalloc	New	Hoard	Ptmalloc
Linux scalability	2.25	1.11	1.83	2.75	1.38	1.92
Threadtest	2.18	1.20	1.94	2.35	1.23	1.97
Larson	2.90	2.22	2.53	2.95	2.37	2.67

Table 1: Contention-free speedup over libc malloc.

loc2 (Nov. 2002) [6]. All allocators and benchmarks were compiled using gcc and g++ with the highest optimization level (-O6) in 64-bit mode. We used pthreads for multithreading. All allocators were dynamically linked as shared libraries. For meaningful comparison, we tried to use optimal versions of Hoard and Ptmalloc as best we could. We modified the PowerPC lightweight locking routines in Hoard by removing a `sync` instruction from the beginning of the lock acquisition path, replacing the `sync` at the end of lock acquisition with `isync`, and adding `eieio` before lock release. These changes reduced the average contention-free latency of a pair of malloc and free using Hoard from 1.76 μ s. to 1.51 μ s. on POWER3, and from 885 ns. to 560 ns. on POWER4. The default distribution of Ptmalloc2 uses pthread mutex for locking. We replaced calls to pthread mutex by calls to a lightweight mutex that we coded using inline assembly. This reduced the average contention-free latency of a pair of malloc and free using Ptmalloc by more than 50%, from 1.93 μ s. to 923 ns. on POWER3 and from 812 ns. to 404 ns. on POWER4. In addition, Ptmalloc showed substantially better scalability using the lightweight locks than it did using pthread mutex locks.

4.1 Benchmarks

Due to the lack of standard benchmarks for multithreaded dynamic memory allocation, we use microbenchmarks that focus on specific performance characteristics. We use six benchmarks: benchmark 1 of *Linux Scalability* [15], *Threadtest*, *Active-false*, and *Passive-false* from Hoard [3], *Larson* [13], and a lock-free producer-consumer benchmark that we describe below.

In *Linux scalability*, each thread performs 10 million malloc/free pairs of 8 byte blocks in a tight loop. In *Threadtest*, each thread performs 100 iterations of allocating 100,000 8-byte blocks and then freeing them in order. These two benchmarks capture allocator latency and scalability under regular private allocation patterns.

In *Active-false*, each thread performs 10,000 malloc/free pairs (of 8 byte blocks) and each time it writes 1,000 times to each byte of the allocated block. *Passive-false* is similar to *Active-false*, except that initially one thread allocates blocks and hands them to the other threads, which free them immediately and then proceed as in *Active-false*. These two benchmarks capture the allocator's ability to avoid causing false sharing [22] whether actively or passively.

In *Larson*, initially one thread allocates and frees random sized blocks (16 to 80 bytes) in random order, then an equal number of blocks (1024) is handed over to each of the remaining threads. In the parallel phase which lasts 30 seconds, each thread randomly selects a block and frees it, then allocates a new random-sized block in its place. The benchmark measures how many free/malloc pairs are performed during the parallel phase. *Larson* captures the robustness of malloc's latency and scalability under irregular allocation patterns with respect to block-size and order of deallocation over a long period of time.

In the lock-free *Producer-consumer* benchmark, we measure the number of tasks performed by t threads in 30 seconds. Initially, a database of 1 million items is initialized randomly. One thread is the producer and the others, if any, are consumers. For each task, the producer selects a random-sized (10 to 20) random set of array indexes, allocates a block of matching size (40 to 80 bytes) to record the array indexes, then allocates a fixed size task structure (32 bytes) and a fixed size queue node (16 bytes), and enqueues the task in a lock-free FIFO queue [19, 20]. A consumer thread repeatedly dequeues a task, creates histograms from the database for the indexes in the task, and then spends time proportional to a parameter *work* performing local work similar to the work in Hoard's *Threadtest* benchmark. When the number of tasks in the queue exceeds 1000, the producer helps the consumers by dequeuing a task from the queue and processing it. Each task involves 3 malloc operations on the part of the producer, and one malloc and 4 free operations on the part of the consumer. The consumer spends substantially more time on each task than the producer. *Producer-consumer* captures malloc's robustness under the producer-consumer sharing pattern, where threads free blocks allocated by other threads.

4.2 Results

4.2.1 Latency

Table 1 presents contention-free⁴ speedups over libc malloc for the new allocator, Hoard, and Ptmalloc, for the benchmarks that are affected by malloc latency: *Linux scalability*, *Threadtest*, and *Larson*. Malloc's latency had little or no effect on the performance of *Active-false*, *Passive-false*, and *Producer-consumer*.

The new allocator achieves significantly lower contention-free latency than the other allocators under both regular and irregular allocation patterns. The reason is that it has a faster execution path in the common case. Also, unlike lock-based allocators, it operates only on the actual allocator variables without the need to operate on additional lock related variables and to synchronize these accesses with the accesses to the allocator variables through fence instructions.

The new allocator requires only one memory fence instruction (line 17 of `free`) in the common case for each pair of malloc and free, while every lock acquisition and release requires an instruction fence before the critical section to pre-

⁴It appears that libc malloc as well as Hoard use a technique where the parent thread bypasses synchronization if it knows that it has not spawned any threads yet. We applied the same technique to our allocator and the average single-thread latency for our allocator was lower than those for libc malloc and Hoard. However, in order to measure true contention-free latency under multithreading, in our experiments, the parent thread creates an additional thread at initialization time which does nothing and exits immediately before starting time measurement.

vent reads inside the critical section from reading stale data before lock acquisition, and a memory fence after the end of the critical section to ensure that the lock is not observed to be free before the writes inside the critical sections are also observed by other processors. In the common case, a pair of malloc and free using Ptmalloc and Hoard need to acquire and release two and three locks, respectively.

Interestingly, when we conducted experiments with a lightweight test-and-set mutual exclusion lock on the POWER4 system, we found that the average contention-free latency for a pair of lock acquire and release is 165 ns. On the other hand, the average contention-free latency for a pair of malloc and free in *Linux Scalability* using our allocator is 282 ns., i.e., it is less than twice that of a minimal critical section protected by a lightweight test-and-set lock. That is, on that architecture, it is highly unlikely—if not impossible—for a lock-based allocator (without per-thread private heaps) to have lower latency than our lock-free allocator, even if it uses the fastest lightweight lock to protect malloc and free and does nothing in these critical sections.

4.2.2 Scalability and Avoiding False Sharing

Figure 8(a) shows speedup results relative to contention-free libc malloc for *Linux scalability*. Our allocator, Ptmalloc, and Hoard scale well with varying slopes proportional to their contention-free latency. Libc malloc does not scale at all, its speedup drops to 0.4 on two processors and continues to decline with more processors. On 16 processors the execution time of libc malloc is 331 times as much as that of our allocator.

The results for *Threadtest* (Figure 8(b)) show that our allocator and Hoard scale in proportion to their contention-free latencies. Ptmalloc scales but at a lower rate under high contention, as it becomes more likely that threads take over the arenas of other threads when their own arenas have no free blocks available, which increases the chances of contention and false sharing.

Figures 8(c–d) show the results for *Active-false* and *Passive-false*. The latency of malloc itself plays little role in these results. The results reflect only the effect of the allocation policy on inducing or avoiding false sharing. Our allocator and Hoard are less likely to induce false sharing than Ptmalloc and libc malloc.

In *Larson* (Figure 8(e)), which is intended to simulate server workloads, our allocator and Hoard scale, while Ptmalloc does not, probably due to frequent switching of threads between arenas, and consequently more frequent cases of freeing blocks to arenas locked by other threads. We also noticed, when running this benchmark, that Ptmalloc creates more arenas than the number of threads, e.g., 22 arenas for 16 threads, indicating frequent switching among arenas by threads. Even though freeing blocks to remote heaps in Hoard can degrade performance, this effect is eliminated after a short time. Initially threads free blocks that were allocated by another thread, but then in the steady state they free blocks that they have allocated from their own processor heaps.

4.2.3 Robustness under Producer-Consumer

For *Producer-consumer* we ran experiments with various values for *work* (parameter for local work per task). Figures 8(f–h) show the results for *work* set to 500, 750, and 1000, respectively. The results for all the allocators are virtually identical under no contention, thus the latency of the allocator plays a negligible role in the results for this bench-

mark. The purpose of this benchmark is to show the robustness of the allocators under the producer-consumer sharing pattern when the benchmark is scalable. The case where the benchmark cannot scale even using a perfect allocator is not of interest. We focus on the knee of the curve, where the differences in robustness between allocators impact the scalability of the benchmark.

Our allocator scales perfectly with work set to 1000 and 750, and up to 13 processors with work set to 500. With more than 13 processors (and with work set to 500), we found that the producer could not keep up with the consumers (as the queue was always empty at the end of each experiment), which is not an interesting case as the application would not be scalable in any case. Our allocator’s scalability is limited only by the scalability of the application.

Ptmalloc scales to a lesser degree, but at the cost of higher external memory fragmentation, as the producer keeps creating and switching arenas due to contention with consumers, even though most arenas already have available blocks.

Hoard’s scalability suffers due to high contention on the producer’s heap, as 75% of all malloc and free operations are targeted at the same heap. Our allocator’s performance does not suffer, although it faces exactly the same situation. The main reason is that in Hoard, even in the common case, free operations need to acquire either the processor heap’s lock or the global heap’s lock. In our allocator typical free operations are very simple and operate only on the superblock descriptor associated with the freed block, thus allowing substantially more concurrency than Hoard. Other minor reasons for our allocator’s ability to perform well even under contention on the same superblock are: (a) In our allocator, read-modify-write code segments are shorter in duration, compared with critical sections in Hoard. (b) Successful lock-free operations can overlap in time, while mutual exclusion locks by definition must strictly serialize critical sections.

4.2.4 Optimization for Uniprocessors

With uniprocessors in mind, we modified a version of our allocator such that threads use only one heap, and thus when executing malloc, threads do not need to know their id. This optimization achieved 15% increase in contention-free speedup on *Linux scalability* on POWER3. When we used multiple threads on the same processor, performance remained unaffected, as our allocator is preemption-tolerant. In practice, the allocator can determine the number of processors in the system at initialization time by querying the system environment.

4.2.5 Space Efficiency

We tracked the maximum space used by our allocator, Hoard, and Ptmalloc when running the benchmarks that allocate a large number of blocks: *Threadtest*, *Larson*, and *Producer-consumer*. The maximum space used by our allocator was consistently slightly less than that used by Hoard, as in our allocator each processor heap holds at most two superblocks, while in Hoard each processor heap holds a variable number of superblocks proportional to allocated blocks. The maximum space allocated by Ptmalloc was consistently more than that allocated by Hoard and our allocator. The ratio of the maximum space allocated by Ptmalloc to the maximum space allocated by ours, on 16 processors, ranged from 1.16 in *Threadtest* to 3.83 in *Larson*.

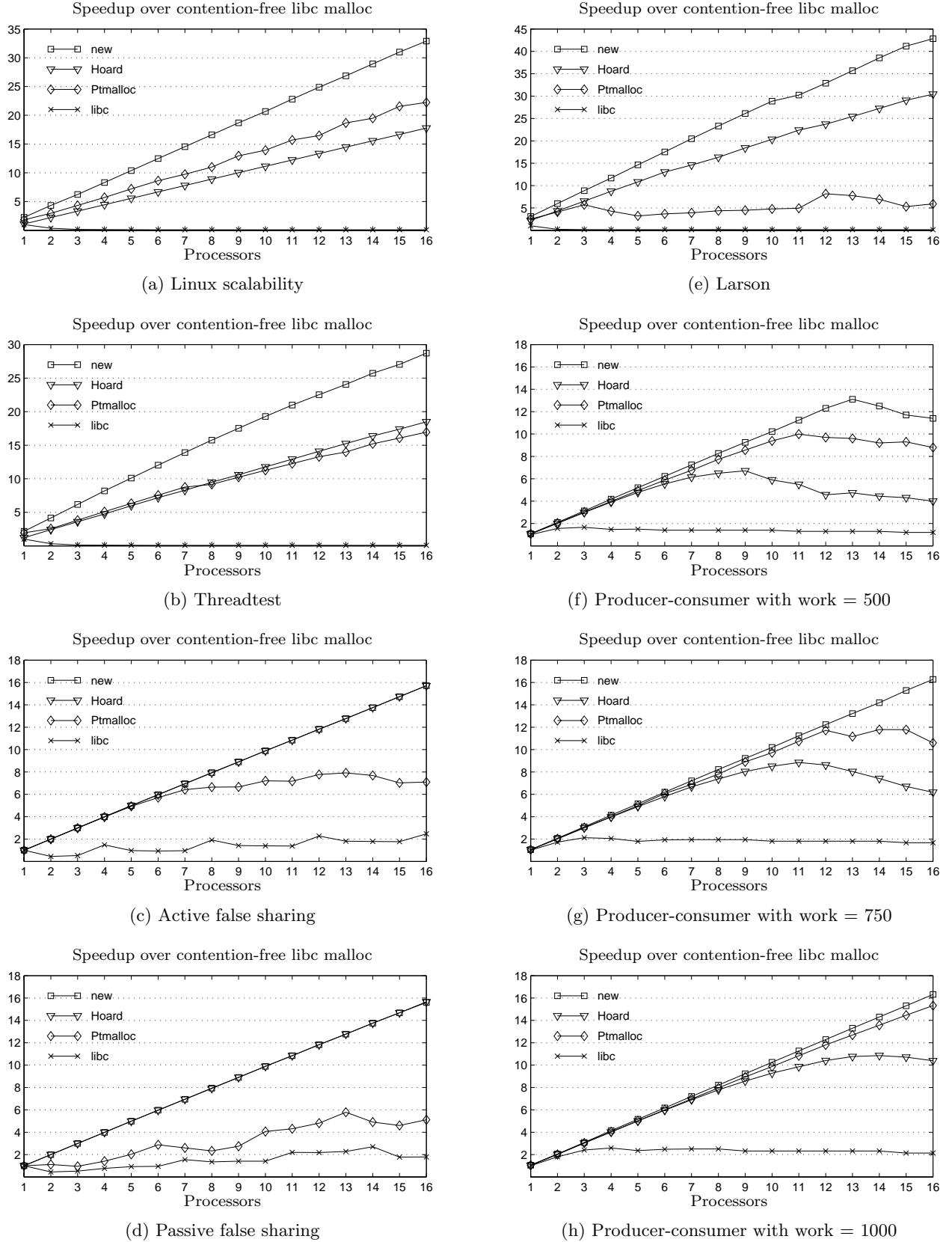


Figure 8: Speedup results on 16-way 375 MHz POWER3.

5. SUMMARY

In this paper we presented a completely lock-free dynamic memory allocator. Being completely lock-free, our allocator is immune to deadlock regardless of scheduling policies and even when threads may be killed arbitrarily. Therefore, it can offer async-signal-safety, tolerance to priority inversion, kill-tolerance, and preemption-tolerance, without requiring any special kernel support or incurring performance overhead. Our allocator is portable across software and hardware platforms, as it requires only widely-available OS support and hardware atomic primitives. It is general-purpose and does not impose any unreasonable restrictions regarding the use or initialization of the address space. It is space efficient and limits space blowup [3] to a constant factor.

Our experimental results compared our allocator with the default AIX 5.1 libc malloc, and two of the best multithread allocators, Hoard [3] and Ptmalloc [6]. Our allocator outperformed the other allocators in all cases, often by significant margins, under various levels of parallelism and allocation patterns. Our allocator showed near perfect scalability under various allocation and sharing patterns. Under maximum contention on 16 processors, it achieved a speedup of 331 over libc malloc.

Equally significant, our allocator offers substantially lower latency than the other allocators. Under no contention, it achieved speedups of 2.75, 1.99, and 1.43 over libc malloc, and highly-optimized versions of Hoard and Ptmalloc, respectively. Scalable allocators are often criticized that they achieve their scalability at the cost of higher latency in the more common case of no contention. Our allocator achieves both scalability and low latency, in addition to many other performance and qualitative advantages.

Furthermore, this work, in combination with recent lock-free methods for safe memory reclamation [17, 19] and ABA prevention [18] that use only single-word CAS, allows lock-free algorithms including efficient algorithms for important object types—such as LIFO stacks [8], FIFO queues [20], and linked lists and hash tables [16, 21]—to be both completely dynamic and completely lock-free, including in 64-bit applications and on systems without support for automatic garbage collection, all efficiently without requiring special OS support and using only widely-available 64-bit atomic instructions.

Acknowledgments

The author thanks Emery Berger, Michael Scott, Yefim Shuf, and the anonymous referees for valuable comments on the paper.

6. REFERENCES

- [1] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.
- [2] Emery D. Berger. *Memory Management for High-Performance Applications*. PhD thesis, University of Texas at Austin, August 2002.
- [3] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 117–128, November 2000.
- [4] Bruce M. Bigler, Stephen J. Allan, and Rodney R. Oldhoeft. Parallel dynamic storage allocation. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 272–275, August 1985.
- [5] Dave Dice and Alex Garthwaite. Mostly lock-free malloc. In *Proceedings of the 2002 International Symposium on Memory Management*, pages 269–280, June 2002.
- [6] Wolfram Gloger. *Dynamic Memory Allocator Implementations in Linux System Libraries*. <http://www.dent.med.uni-muenchen.de/~wmglo/>.
- [7] Maurice P. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.
- [8] IBM. *IBM System/370 Extended Architecture, Principles of Operation*, 1983. Publication No. SA22-7085.
- [9] IEEE. *IEEE Std 1003.1, 2003 Edition*, 2003.
- [10] Arun K. Iyengar. *Dynamic Storage Allocation on a Multiprocessor*. PhD thesis, MIT, 1992.
- [11] Arun K. Iyengar. Parallel dynamic storage allocation algorithms. In *Proceedings of the Fifth IEEE Symposium on Parallel and Distributed Processing*, pages 82–91, December 1993.
- [12] Leslie Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, November 1977.
- [13] Per-Åke Larson and Murali Krishnan. Memory allocation for long-running server applications. In *Proceedings of the 1998 International Symposium on Memory Management*, pages 176–185, October 1998.
- [14] Doug Lea. *A Memory Allocator*. <http://gee.cs.oswego.edu/dl/html/malloc.html>.
- [15] Chuck Lever and David Boreham. Malloc() performance in a multithreaded Linux environment. In *Proceedings of the FREENIX Track of the 2000 USENIX Annual Technical Conference*, June 2000.
- [16] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 73–82, August 2002.
- [17] Maged M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Distributed Computing*, pages 21–30, July 2002.
- [18] Maged M. Michael. ABA prevention using single-word instructions. Technical Report RC 23089, IBM T. J. Watson Research Center, January 2004.
- [19] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 2004. To appear. See www.research.ibm.com/people/m/michael/pubs.htm.
- [20] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 267–275, May 1996.
- [21] Ori Shalev and Nir Shavit. Split-ordered lists: Lock-free extensible hash tables. In *Proceedings of the Twenty-Second Annual ACM Symposium on Principles of Distributed Computing*, pages 102–111, July 2003.
- [22] Josep Torrellas, Monica S. Lam, and John L. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651–663, June 1994.
- [23] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *Proceedings of the 1995 International Workshop on Memory Management*, pages 1–116, September 1995.

COMPARATIVE PERFORMANCE OF MEMORY RECLAMATION
STRATEGIES FOR LOCK-FREE AND CONCURRENTLY-READABLE DATA
STRUCTURES

by

Thomas Edward Hart

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Copyright © 2005 by Thomas Edward Hart

Abstract

Comparative Performance of Memory Reclamation Strategies for Lock-free and
Concurrently-readable Data Structures

Thomas Edward Hart

Master of Science

Graduate Department of Computer Science

University of Toronto

2005

Despite their advantages, lock-free algorithms are often not adopted in practice, partly due to the perception that they perform poorly relative to lock-based alternatives in common situations when there is little contention for objects or the CPUs.

We show that memory reclamation can be a dominant performance cost for lock-free algorithms; therefore, choosing the most efficient memory reclamation method is essential to having lock-free algorithms perform well. We compare the costs of three memory reclamation strategies: *quiescent-state-based reclamation*, *epoch-based reclamation*, and *safe memory reclamation*. Our experiments show that changing the workload or execution environment can change which of these schemes is the most efficient. We therefore demonstrate that there is, to date, no panacea for memory reclamation for lock-free algorithms.

Using a common reclamation scheme, we fairly compare lock-free and concurrently-readable hash tables. Our evaluation shows that programmers can choose memory reclamation schemes mostly independently of the target algorithm.

Acknowledgements

First, I'd like to thank my supervisor, Angela Demke-Brown, for helping me shape my vague ideas into coherent research, and helping me to take a step back and see which questions are important and which are not.

Second, I want to thank Paul McKenney, with whom I have enjoyed a very helpful collaboration during the course of this project. Having such an experienced colleague has been immensely helpful as I start my research career.

I wish to thank Maged Michael and Keir Fraser for so readily answering questions about their respective work.

I also want to thank Vassos Hadlizacos for introducing me to practical lock-free synchronization through Greenwald's work on Cache Kernel, and my colleagues in the Systems Software Reading Group for introducing me to Read-Copy Update. Without these colleagues, I never would have become aware of this research topic.

During the course of this project, I solicited advice from many, many colleagues here at the University of Toronto, all of whom have been very generous in their help. In particular, I would like to thank Faith Fich, Alex Brodsky, Cristiana Amza, Reza Azimi, David Tam, Marc Berndl, Mathew Zaleski, Jing Su, and Gerard Baron.

Finally, I would like to thank my lab's system administrator, Norman Wilson, for keeping the machines I needed for my experiments running smoothly, and for forgiving me when I occasionally crashed them.

Contents

1	Introduction	1
1.1	Problems with Locking	1
1.2	Memory Reclamation	3
1.3	Contributions	4
1.4	Organization of Thesis	5
2	Fundamentals	7
2.1	Terminology	7
2.1.1	Threads	8
2.1.2	Shared Objects	8
2.1.3	Hardware Operations	12
2.2	Memory Consistency Models	15
3	Lock-free and Concurrently-readable Algorithms	17
3.1	Theory of Non-blocking Synchronization	17
3.2	Practical Non-blocking Algorithms	18
3.2.1	Higher-level primitives	19
3.2.2	Algorithms using CAS or LL/SC	20
3.3	Concurrently-Readable Algorithms	32
4	Memory Reclamation Schemes	35

4.1	Descriptions of Schemes	35
4.1.1	Blocking Methods	35
4.1.2	Lock-free Methods	39
4.2	Applying the Schemes	43
4.3	Analytic Comparison of Methods	49
5	Experimental Evaluation	53
5.1	Experimental Setup	53
5.1.1	Algorithms Compared	53
5.1.2	Test Program	54
5.1.3	Operating Environment	55
5.1.4	Limitations of Experiment	57
5.2	Performance Analysis	57
5.2.1	Effects of Traversal Length	60
5.2.2	Effects of CPU Contention	63
5.2.3	Relative Severity	68
5.2.4	Low Overhead of QSBR	69
5.2.5	Lock-free Versus Concurrently-readable Linked List Algorithms . .	71
5.2.6	Summary of Recommendations	75
6	Related Work	79
6.1	Blocking Memory Reclamation for Non-blocking Algorithms	79
6.2	Vulnerabilities of Blocking Memory Reclamation Schemes	81
6.3	Performance Comparisons	82
7	Conclusions and Future Work	84
Bibliography		87

List of Tables

5.1 Characteristics of Machines	56
7.1 Properties of Memory Reclamation Schemes	85

List of Figures

2.1	Hierarchy of properties of concurrent objects, represented as a partial order. Any property in the hierarchy is strictly stronger than all properties below it; for example, <i>obstruction-free</i> and <i>almost non-blocking</i> are weaker than <i>lock-free</i> , but in different ways.	11
2.2	Pseudocode definition of CAS.	12
2.3	Pseudocode definitions of LL and SC.	13
2.4	Pseudocode for implementing CAS using LL/SC.	13
2.5	Illustration of the ABA problem.	14
3.1	The consensus hierarchy.	18
3.2	Pseudocode definition of DCAS.	19
3.3	Example operation of search function for lock-free linked list.	21
3.4	Pseudocode for search function for lock-free list, stripped of memory reclamation code.	22
3.5	Example operation of insert function for lock-free linked list.	24
3.6	Error which can occur in a naïve lock-free linked list implementation when insertions and deletions are interleaved. To prevent such errors, the lock-free linked list must mark a node’s <i>next</i> pointer before deleting the node (Figure 3.7).	26
3.7	Example operation of delete function for lock-free linked list.	27
3.8	Dequeue from non-empty lock-free queue.	28

3.9	Pseudocode for dequeue function for lock-free queue, stripped of memory reclamation code.	29
3.10	Enqueue to non-empty lock-free queue.	30
3.11	Pseudocode for enqueue function for lock-free queue, stripped of memory reclamation code.	31
3.12	Concurrently-readable node modification example from which <i>read-copy update</i> derives its name.	33
3.13	Concurrently-readable insertion.	33
3.14	Concurrently-readable deletion.	34
4.1	Illustration of EBR. Threads follow the global epoch. If a thread observes that all other threads have seen the current epoch, then it may update the global epoch. Hence, if the global epoch is e , threads in critical sections can be in either epoch $e + 1$ or e , but not $e - 1$ (all mod 3). The time period $[t_1, t_2]$ is thus a grace period for thread $T1$	36
4.2	Illustration of QSBR. Thick lines represent quiescent states. The time interval $[t_1, t_2]$ is a grace period: at time t_2 , each thread has passed through a quiescent state since t_1 , so all nodes logically removed before time t_1 can be physically deleted.	38
4.3	Illustration of SMR. q is logically removed from the linked list on the right of the diagram, but cannot be physically deleted while $T3$'s hazard pointer $HP[4]$ is still pointing to it.	41
4.4	Lock-free queue's <code>dequeue()</code> function, using SMR.	45
4.5	Lock-free queue's <code>dequeue()</code> function, using QSBR.	46
4.6	Lock-free queue's <code>dequeue()</code> function, using EBR.	47

4.7	Illustration of why QSBR is inherently blocking. Here, thread T_2 does not go through a quiescent state for a long period of execution time; hence, threads T_1 and T_3 must wait to reclaim memory. A similar argument holds for EBR.	51
5.1	High-level pseudocode for the test loop of our program. Each thread executes this loop. The call to QUIESCENT_STATE() is ignored unless we are using QSBR.	55
5.2	Single-threaded memory reclamation costs on PowerPC. Hash table statistics are for a 32-bucket hash table with a load factor of 1. Queue statistics are for a single non-empty queue.	58
5.3	Hash table, 32 buckets, load factor 1, read-only workload. Spinlocks scale poorly as the number of threads increases.	59
5.4	Hash table, 32 buckets, one thread, read-only workload, varying load factor.	60
5.5	Hash table, 32 buckets, one thread, write-only workload, varying load factor.	61
5.6	Hash table, 32 buckets, one thread, write-only workload, varying load factor.	62
5.7	100 queues, variable number of threads, Darwin/PPC.	63
5.8	Hash table, 32 buckets, load factor 1, write-only workload, variable number of threads, Darwin/PPC.	64
5.9	100 queues, variable number of threads, Linux/IA-32.	64
5.10	Hash table, 32 buckets, 16 threads, write-only workload, varying load factor.	66

5.11 Hash table, 32 buckets, load factor 10, write-only workload, varying number of threads.	66
5.12 Hash table, 32 buckets, load factor 20, write-only workload, varying number of threads.	67
5.13 Queues, two threads, varying number of queues.	69
5.14 Hash table, 32 buckets, two threads, load factor 5, varying update fraction.	70
5.15 Hash table, 32 buckets, two threads, load factor 5, varying update fraction. QSBR allows the lock-free algorithm to out-perform RCU for almost any workload; neither SMR nor EBR achieve this.	71
5.16 Code for fast searches of lock-free list; compare to pseudocode of Figure 3.4.	72
5.17 Hash table, 32 buckets, two threads, load factor 5, varying update fraction between 0 and 0.1.	72
5.18 Hash table, 32 buckets, two threads, read-only workload, varying load factor.	73
5.19 Hash table, 32 buckets, two threads, write-only workload, varying load factor.	73
5.20 Decision tree for choosing a memory reclamation scheme.	76
5.21 Decision tree for choosing whether to use the lock-free or concurrently- readable linked list algorithm.	78

Chapter 1

Introduction

Shared memory multiprocessing is becoming important outside the traditional areas of high-performance computing such as scientific computation, graphics rendering, and web servers. New technologies such as *simultaneous multithreading (SMT)* and *chip multiprocessing (CMP)* are bringing multiprocessing to commodity desktops. Furthermore, many applications running on these systems, such as web browsers and operating system kernels, are already multithreaded. It is essential that these concurrent applications perform well; we expect that technologies like SMT and CMP will make this requirement even more important in the future.

The rest of this chapter describes the problems associated with locking, explains the need for memory reclamation, summarizes the contributions of this thesis, and outlines the organization of the remainder of this document.

1.1 Problems with Locking

Concurrent applications require a means to coordinate accesses to shared data structures. Mutual exclusion, implemented via locks or semaphores, is the most common solution to the synchronization problem. Mutual exclusion is intuitive; however, it has several drawbacks. Locks, in particular, can be bottlenecks in high-performance shared memory

programs, and suffer from several problems:

- **Poor reliability:** a thread that crashes while holding a lock will make the resource associated with the lock unavailable to all other threads. This is the most cited problem with locks in theoretical literature, since fault tolerance is a favorite problem in the theory of distributed computing; however, few systems researchers consider this the most compelling disadvantage of locks.
- **Poor performance in the face of preemption:** a thread that is pre-empted while holding a lock will make the associated resource unavailable until the thread is rescheduled and can complete its work. This can result in *convoying*, which occurs when a pre-empted or otherwise stalled thread holds a lock, and other threads form a “convoy” waiting for the lock.
- **Priority inversion:** a low priority thread may be pre-empted while holding a lock on a shared resource; a high priority thread requiring the shared resource may then be scheduled, and have to wait for the lower priority thread.
- **Vulnerability to deadlock:** if threads must acquire locks on two or more resources, deadlock is possible if those locks are not acquired in the same order. A thread may even deadlock with itself if it attempts to acquire a lock which it already holds..

In addition to these deficiencies, lock-based approaches require expensive operations, such as compare-and-swap, to acquire and release locks — even when the locks are not contended.

Many researchers are therefore interested in ways to avoid using locks [42, 18, 54, 55]. Using reader-writer locks instead of normal spinlocks increases concurrency by allowing either multiple readers or a single writer at any given time. Using a *concurrently-readable* [34] data structure also permits multiple readers, but in this case they can run concurrently with a single writer, and do not need to acquire a lock. Using a *lock-free* data

structure is more complicated, but allows readers and writers both to run concurrently when logically possible, and sidesteps the above-mentioned disadvantages of locks; however, lock-free data structures typically require expensive atomic operations. Some recent lock-free algorithms, however, require few such expensive operations, and therefore provide an attractive alternative to lock-based designs, as they have been shown to have lower overhead, even when contention for objects and the CPUs is low [47].

1.2 Memory Reclamation

Memory reclamation is required for all dynamic lock-free and concurrently-readable data structures, such as linked lists and queues. We distinguish logical deletion of a node, N (removing it from a shared data structure so that no new references to N may be created) from physical deletion of that node (allowing the memory used for N to be reclaimed for arbitrary reuse). If a thread T_1 logically deletes a node N from a lock-free data structure, it cannot physically delete N until no other thread T_2 holds a reference to N , since physically deleting N may cause T_2 to crash or execute incorrectly. Never physically deleting logically deleted nodes is also unacceptable, since this will eventually lead to out-of-memory errors which will stop threads from making progress.

Choosing an inefficient memory reclamation scheme can ruin the performance of a lock-free or concurrently-readable algorithm. Reference counting [61, 12], for example, has high overhead in the base case, and scales poorly in structures for which long chains of nodes must be traversed [47]. Little work has been done on comparing different memory reclamation strategies; we address this deficiency, showing that the performance of memory reclamation schemes depends both on their base costs, and on the target workload and execution environment. We expect our results can provide some guidance to implementers of lock-free and concurrently-readable algorithms in choosing an appropriate scheme for their specific applications.

Current memory reclamation strategies suggested for lock-free data structures [44, 47, 28, 17, 11] have high per-operation runtime overhead. We believe that imposing large overhead on an algorithm in order to manage memory reclamation is unacceptable. We therefore propose using *quiescent-state-based reclamation* (QSBR) [42, 39, 18, 6, 38, 40], an efficient scheme with negligible overhead pioneered in the domain of operating system kernels, to manage memory for lock-free data structures whenever CPU contention is low. QSBR is normally used with concurrently-readable algorithms; however, we have found that it also works well with lock-free ones. We show by example that QSBR can be implemented in applications other than operating system kernels, and that using this memory reclamation scheme can make lock-free algorithms significantly more efficient. The cost of QSBR’s increased performance is an increased burden on the application programmer, and the risk of out-of-memory errors. We therefore believe that, despite the performance advantage of QSBR, these other memory reclamation algorithms have a place in situations where this trade-off is unacceptable.

The other memory reclamation schemes, safe memory reclamation (SMR) and epoch-based reclamation (EBR), are intended for use with lock-free algorithms. We demonstrate that just as QSBR can be used with lock-free schemes, SMR and EBR can be used with concurrently-readable ones, therefore showing that the choice of memory reclamation scheme is mostly independent of the target algorithm.

1.3 Contributions

Although lock-free and concurrently-readable algorithms each have their respective adherents, we are not aware of any work which has examined the tradeoffs between comparable lock-free and concurrently-readable algorithms. Were one to make such a comparison, one might naïvely use each algorithm with the reclamation scheme suggested by its creator, and compare, for example, a lock-free algorithm using SMR to a concurrently-

readable algorithm using QSBR. One might even be unaware that the algorithm and its reclamation scheme *can* be separated. Since the choice of memory reclamation scheme has a profound impact on performance, any such comparison would be unfair. Our decoupling of memory reclamation schemes from the algorithms for which they were originally designed therefore allows us to make the first fair and detailed comparison between lock-free and concurrently-readable chaining hash tables.

In summary, the contributions of this thesis are as follows:

- We demonstrate that the choice of algorithm and memory reclamation scheme are mostly independent.
- We analyze the strengths and weaknesses of each of three memory reclamation strategies — QSBR, SMR, and EBR.
- We make the first comparison of the performance of a lock-free algorithm to a concurrently-readable alternative.

Our experiments were conducted on commodity dual-processor PowerPC and IA-32 machines. Our results show that changing the workload or execution environment can change which memory reclamation scheme is the most efficient; we therefore demonstrate that there is, to date, no panacea for memory reclamation for lock-free and concurrently-readable algorithms.

1.4 Organization of Thesis

This thesis is organized as follows. Chapter 2 discusses the terminology we use and our system model. Chapter 3 provides background on lock-free and concurrently-readable algorithms, with an emphasis on the algorithms on which we performed our experiments. Chapter 4 presents the memory reclamation schemes we consider, as well as some which

we did not. Chapter 5 presents our experimental results, and Chapter 6 discusses related work. Chapter 7 concludes the thesis.

Chapter 2

Fundamentals

In this chapter, we define terminology for discussing lock-free and concurrently-readable algorithms. We then discuss weak memory consistency models, which, we show in chapter 5, have an important effect on the performance of memory reclamation schemes.

2.1 Terminology

This work attempts to use results from the theory of distributed computing, specifically non-blocking synchronization, in a practical setting. As such, we shall use terminology which allows us to discuss non-blocking synchronization in a cogent matter. We shall define our concepts of threads, shared objects and their properties, and the hardware operations required to implement these shared objects.

The level of formality of this terminology is lower than that used in pure theory literature, but higher than that of systems literature; we hope that the terminology will be acceptable to people in both groups.

2.1.1 Threads

The terms *process* and *thread* are, for the purposes of our discussion, synonymous. Both refer to a unit of execution which may access shared objects. Since the additional denotations of process in operating systems literature have no bearing on our analysis of the algorithms under consideration, we prefer the term “thread.”

Typically, for the purposes of any theoretical analysis, a thread may *crash*. This means only that the crashed thread ceases to perform any actions, and does not include any of the other denotations of the term “crash” in systems literature. A thread that does not crash is said to be *correct*. We assume that the actions of any thread occur over a sequence of discrete *time steps*.

2.1.2 Shared Objects

A *shared object* is an entity in memory which can be accessed and modified only through a pre-defined set of *operations*. Conceptually, these operations correspond to *methods* in object-oriented programming; for example, the operations on a shared queue are *enqueue* and *dequeue*. Multiple threads may access a shared object. A *dynamic shared object* is a shared object which is made up of *nodes* which are dynamically allocated in memory.

In discussing the various forms of non-blocking synchronization, we shall use terminology which is becoming standard in the literature [17]. A shared object is *non-blocking* if and only if it is *wait-free*, *lock-free*, or *obstruction-free*. A shared object O is wait-free, lock-free, or obstruction-free, respectively, if and only if the following properties hold:

- *wait-free*: if a correct thread is performing an operation on O , this operation must complete after a finite number of time steps.
- *lock-free*: if a correct thread is performing an operation on O , *some* operation invoked by *some* thread on O must complete after a finite number of time steps.

- *obstruction-free*: if a correct thread is performing an operation on O , *some* operation invoked by *some* thread on O must complete after a finite number of time steps *unless* another thread’s operation conflicts with it.

Those who are used to the terminology of operating systems may find the following summary of the different non-blocking synchronization properties more helpful. A lock-free shared object guarantees that it will not, under any circumstances, cause any of the threads accessing it to deadlock, even if some of those threads crash. An obstruction-free shared object makes almost the same guarantee, except that it can allow livelock. A wait-free shared object is a lock-free shared object that also guarantees that it will not starve any thread accessing it.

We note that designating wait-freedom, lock-freedom, and obstruction-freedom as the only non-blocking properties is only a matter of convention. Originally, *lock-free* was a synonym for *non-blocking*. The intuition behind the latter term is that threads may block while waiting to access a lock-protected object if another thread holds the object’s lock, but threads will never block on accessing a non-blocking object. Obstruction-freedom was later introduced as a non-blocking property, and the literature continues to refer to it as such.

Shared objects may also be *almost non-blocking* or *concurrently-readable*. These properties are similar to the three non-blocking properties, but are not defined as being non-blocking. Roughly, a concurrently-readable shared object allows concurrent reads, but may use locks for updates, and an almost non-blocking shared object allows concurrent reads, but only a bounded number of threads may simultaneously attempt to update the object. In both cases, writers could block on attempting to update the shared object, so it intuitively makes sense not to consider these properties non-blocking.

We owe the term *concurrently-readable* to Lea [34], whose definition we attempt to formalize. A shared object O is *concurrently-readable* if multiple threads may read O concurrently, and any read operation on O begun by a correct thread must complete

after a finite number of time steps. The definition of *concurrently-readable* does not rule out the use of locks for updates; however, a conventional design based on reader-writer locks is not concurrently-readable, since a thread that crashes after write-acquiring the lock will stop other threads from reading the shared object.

By our definition, any wait-free object is concurrently-readable, but a lock-free or obstruction-free object is *not* necessarily concurrently-readable. These non-blocking properties both allow readers to be starved, which violates the definition of concurrently-readable. Furthermore, technically, the definition of obstruction-freedom allows livelock between concurrent reads (although we would consider a design which allows this to be strange), while our definition of concurrently-readable does not.

We note that the literature on concurrently-readable algorithms often refers to the algorithms' read operations as *lock-free*. This use of the term is different than the definition of lock-free concurrent objects. Therefore, we instead refer to operations which do not use locks as *lockless*.

The idea of almost non-blocking objects is due to Boehm [8]. An *almost non-blocking* shared object is one that is *N-non-blocking* for some $N > 0$. The definition of *N-non-blocking* requires the concept of *inactive* threads. We say that a thread is *inactive* if it fails to execute instructions at some pre-determined minimum rate while trying to update a data structure; this is an attempt to model threads which are descheduled, blocked, or crashed altogether. Then, a shared object O is *N-non-blocking* if and only if:

- any number of processes may concurrently access O , and,
- if at most N *inactive* threads are trying to update O , and at least one active thread is trying to access or update O , then some thread will succeed in accessing or updating O in a bounded amount of time.

The relationship between these five properties may be confusing. We show the hierarchy of these properties in Figure 2.1, along with their relation to conventional reader-

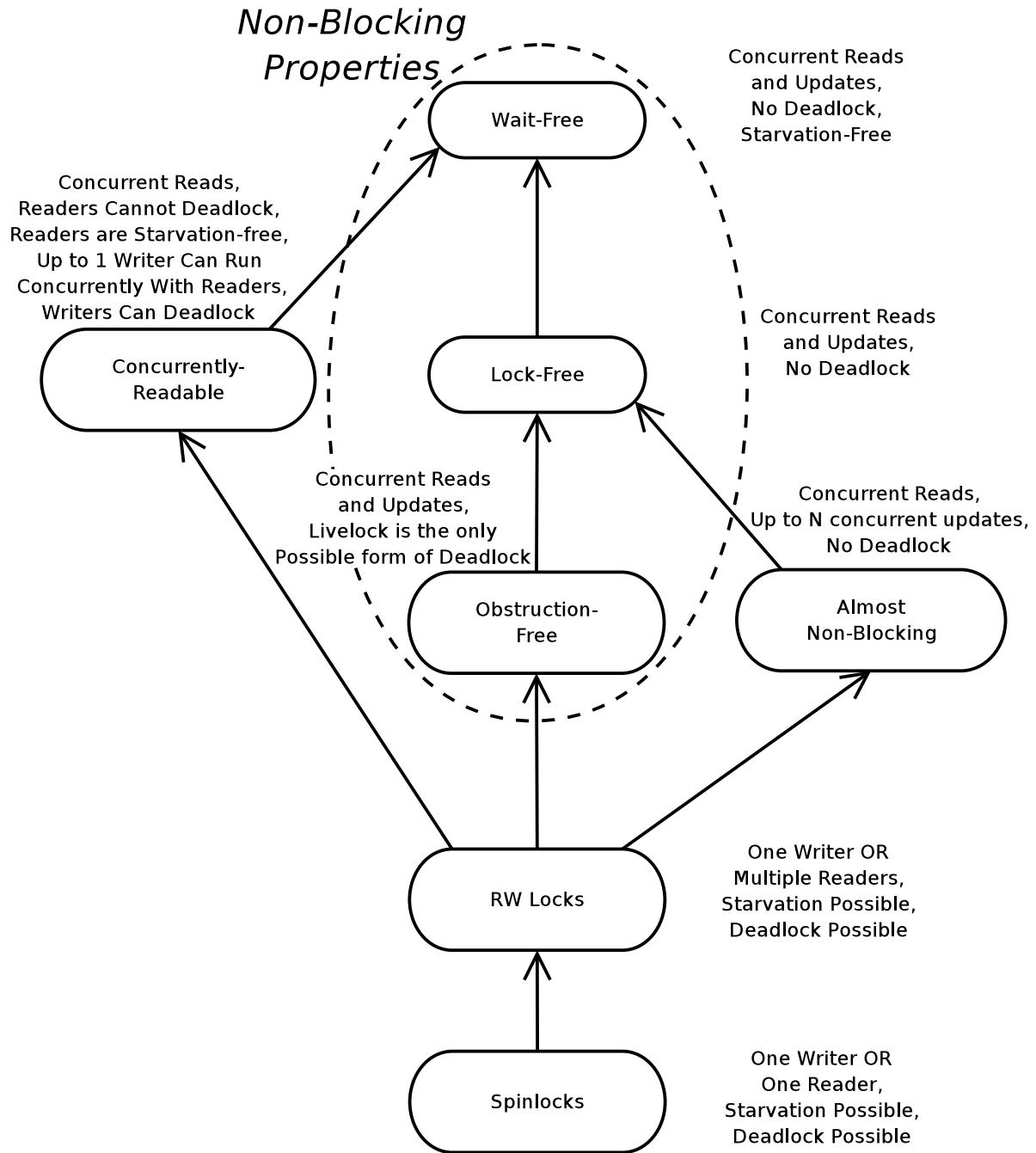


Figure 2.1: Hierarchy of properties of concurrent objects, represented as a partial order. Any property in the hierarchy is strictly stronger than all properties below it; for example, *obstruction-free* and *almost non-blocking* are weaker than *lock-free*, but in different ways.

```
BOOL CAS (void *A, long B, long C)
{
    atomically do {
        if (*A == B) {
            *A = C;
            return TRUE;
        } else {
            return FALSE;
        }
    }
}
```

Figure 2.2: Pseudocode definition of CAS.

writer locking and spinlocking, both of which may be more familiar to the reader.¹ We are primarily interested in lock-free and concurrently-readable shared objects; the other properties are noted only for completeness.

2.1.3 Hardware Operations

Non-blocking shared object implementations typically use either the *compare-and-swap* (*CAS*) operation or the *load-linked* and *store-conditional* (*LL/SC*) pair of instructions to update shared pointers. Both of these instructions are conditional synchronization primitives which atomically both check if a certain condition has been met, and update a word in memory if the check succeeds. CAS and LL/SC are defined as shown in Figures 2.2 and 2.3, respectively.

Typically, processors built according to the complex instruction set (CISC) paradigm will provide a CAS operation, while those built according to the reduced instruction set (RISC) paradigm will provide (restricted) LL/SC. Each of CAS and LL/SC can be used to implement the other, and both may be desirable. LL/SC is often used to implement CAS,

¹This is a minor abuse of terminology, since reader-writer locks and spinlocks are mechanisms, not properties.

```
WORD LL (void *A)
{
    return *A;
}

BOOL SC (void *A, WORD w)
{
    atomically do {
        if (A has not been written to since this thread last called LL(A)) {
            *A = w;
        }
    }
}
```

Figure 2.3: Pseudocode definitions of LL and SC.

```
BOOL CAS (void *A, long B, long C)
{
    do {
        if (LL(A) ≠ B) {
            return FALSE;
        }
    } while (SC(A,C) == FALSE);
    return TRUE;
}
```

Figure 2.4: Pseudocode for implementing CAS using LL/SC.

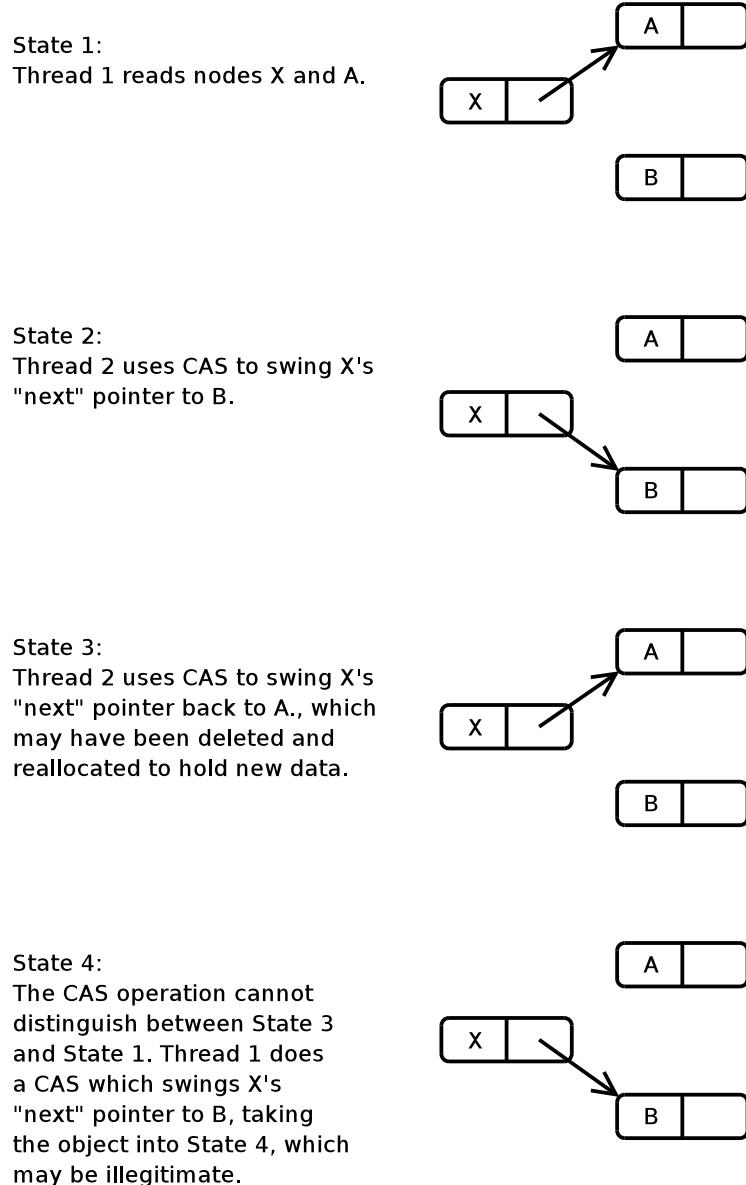


Figure 2.5: Illustration of the ABA problem.

since CAS is intuitive, and convenient for implementing many non-blocking algorithms. Implementing CAS using LL/SC is quite trivial, as shown in Figure 2.4.

Implementations of LL/SC using CAS are more involved [14, 48], but may be desirable in order to avoid the ABA problem. This problem occurs when CAS is used to implement a lock-free algorithm, and is illustrated in Figure 2.5. The effect of two CAS operations can make two states of a data structure indistinguishable to a third CAS operation, which could then succeed and take the data structure into an illegal state.

2.2 Memory Consistency Models

Current literature on lock-free algorithms generally assumes a sequentially-consistent [33] memory model. However, for performance reasons, modern architectures provide a weakly-consistent memory model; sequential consistency can be enforced when needed by using special *fence* instructions. We formalize the key properties of the weakly-consistent memory models used by the processor architectures in this study, using the terminology given in [17].

Let A and B be instructions, let M be a word in memory, and let $<_p$ and $<_m$ represent partial orders. We say that $A <_p B$ if and only if A and B are executed on the same processor, and A precedes B in the program order, and that $A <_m B$ if and only if A is executed before B in all valid program execution orders. The processors in this study provide guarantees of *coherency*, *self-consistency*, and *dependency consistency*, defined as follows²:

- Coherency: At any given point in execution time, M has only one value, and this value is eventually visible to all processors.
- Self-consistency: If A and B both access M , and $A <_p B$, then $A <_m B$.

²Most current processors provide these guarantees; we do not consider processors such as the DEC Alpha 21264 that do not provide dependency consistency.

- Dependency consistency: If A and B are executed on the same processor and B depends on a control decision by A or a state written by A , then $A <_m B$.

Fence instructions can enforce particular orderings when necessary as follows. If A and B are instructions executed on processor P , X is a fence executed on P , and $A <_p X <_p B$, then $A <_m B$. Distinct *write fences* and *read fences*, which impose orderings on writes and reads, respectively, may also be provided.

Our work involves performance measurement on current commodity machines; specifically, IA-32 and PowerPC processors. Since fence instructions are expensive operations, we cannot ignore them in our analysis. Indeed, we will show that the base costs of the reclamation strategies considered depend almost entirely on the number of fences they require. In addition, they are an often-hidden factor in the performance of several lock-free algorithm implementations.

Chapter 3

Lock-free and Concurrently-readable Algorithms

In this chapter, we present an overview of the literature on lock-free and concurrently-readable algorithms. We present in more depth the algorithms used in our analysis.

3.1 Theory of Non-blocking Synchronization

Lamport [32] introduced the idea of concurrent computing without mutual exclusion. Herlihy deepened the theory of non-blocking synchronization by showing that the power of concurrent objects and operations can be characterized by their ability to solve the *consensus* problem [25]. There exists a consensus hierarchy, partially shown in Figure 3.1, such that any object or operation on level n of the hierarchy can, in combination with atomic read/write registers, be used to solve the consensus problem for n processes, but not $n+1$ processes. This result established that strong synchronization primitives such as CAS and LL/SC are required for implementing practical non-blocking synchronization, while weaker primitives such as Test&Set are insufficient.

Herlihy presented a *universal* method to transform any sequential object into an n -process wait-free concurrent one [25] using n -process consensus objects. He later pre-

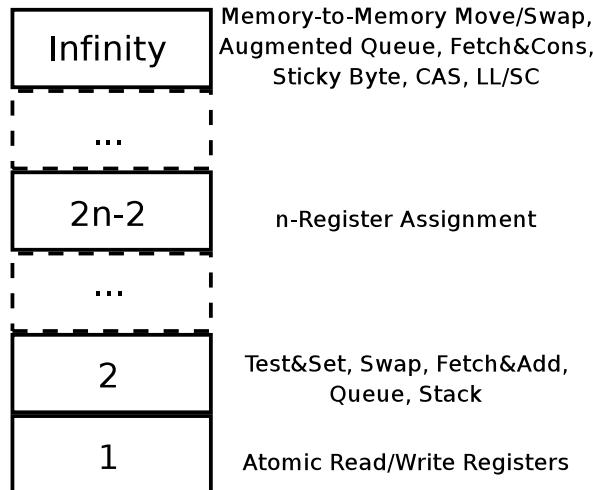


Figure 3.1: The consensus hierarchy.

sented a more efficient transformation based on LL/SC; however, even this more efficient transformation yields implementations which usually have higher overhead than their lock-based counterparts [26]. This poor performance of universal transformations provided a motivation for less general, but more high-performance, non-blocking object implementations.

3.2 Practical Non-blocking Algorithms

Herlihy's universal transformations are too inefficient to be practical. Alemany and Felten [2] identified useless parallelism and unnecessary copying as partial causes of this inefficiency. Several others have attempted to design more efficient universal transformations [4, 3, 7, 52]. However, no universal transformation yet devised can match the performance of custom non-blocking algorithms [19].

```
BOOL DCAS (void *A1, void *A2, long B1, long B2, long C1, long C2)
{
    atomically do {
        if (*A1 == B1 && *A2 == B2) {
            *A1 = C1;
            *A2 = C2;
            return TRUE;
        } else {
            return FALSE;
        }
    }
}
```

Figure 3.2: Pseudocode definition of DCAS.

3.2.1 Higher-level primitives

Many early attempts to build lock-free data structures using techniques more efficient than expensive universal constructions used the *double compare-and-swap (DCAS)* operation shown in Figure 3.2. This operation was useful, for example, to update both a pointer and a version number simultaneously (see example given in [20]). The only two lock-free operating system kernels yet developed, Synthesis and Cache Kernel, both used DCAS [37, 20]. Unfortunately, DCAS was not supported by any processor other than the Motorola 680x0 series, and hardware implementations of DCAS are considered impractical with current processor technology.

DCAS is not a convenient enough primitive to make the design of lock-free algorithms easy [13]. Although it is easier to design a lock-free algorithm using DCAS than it is with CAS or LL/SC, constructing DCAS-based lock-free algorithms is still difficult. Many researchers believe that *multi-word compare-and-swap (MCAS)*, which operates on an arbitrary number of words independently, is the correct primitive for easy construction of arbitrary non-blocking shared objects [17, 19, 23]. Fraser showed that MCAS is substantially easier to use than CAS, but has moderate overhead and performs poorly

when contention for a data structure is high [17].

Another approach to easy non-blocking synchronization is *transactional memory*, implemented either in software [30, 17], or hardware [55, 21]. Transactional memory allows operations to be grouped into transactions which atomically succeed or fail. Fraser [17] and Herlihy et. al. [30] showed that transactional memory makes lock-free algorithm design relatively simple. However, software transactional memory has very high overhead [17], and whether or not hardware transactional memory will be practical in future processors is still unknown.

The data structures we consider can be implemented efficiently using CAS, so MCAS and transactional memory are unnecessary. Nevertheless, we believe that our results would also be applicable to more complex data structures which currently require these mechanisms.

3.2.2 Algorithms using CAS or LL/SC

Lock-free algorithms using CAS or LL/SC out-perform versions using MCAS or software transactional memory, but are more difficult to construct. Only a handful of data structures have known lock-free implementations that do not require higher-level abstractions or universal transformations. Among them are linked lists [22, 43], chaining hash tables [43], skip lists [15, 17], doubly-linked lists [58], queues [51, 47], priority queues [57], stacks [60, 47, 24], and deques [45, 58]. More complex data structures, such as binary search trees and red-black trees, currently require higher-level primitives [17, 30].

We note that, despite the relatively-small number of lock-free algorithms based on CAS or LL/SC, many of them appeared after DCAS-based versions (compare [22] to [20], and [45] to [10], for example). In the future, we may see lock-free implementations of other data structures requiring only CAS or LL/SC.

In our experiments, we focus on the lock-free queue and chaining hash table implementations presented in [47]; the latter is simply an array of lock-free linked lists. We

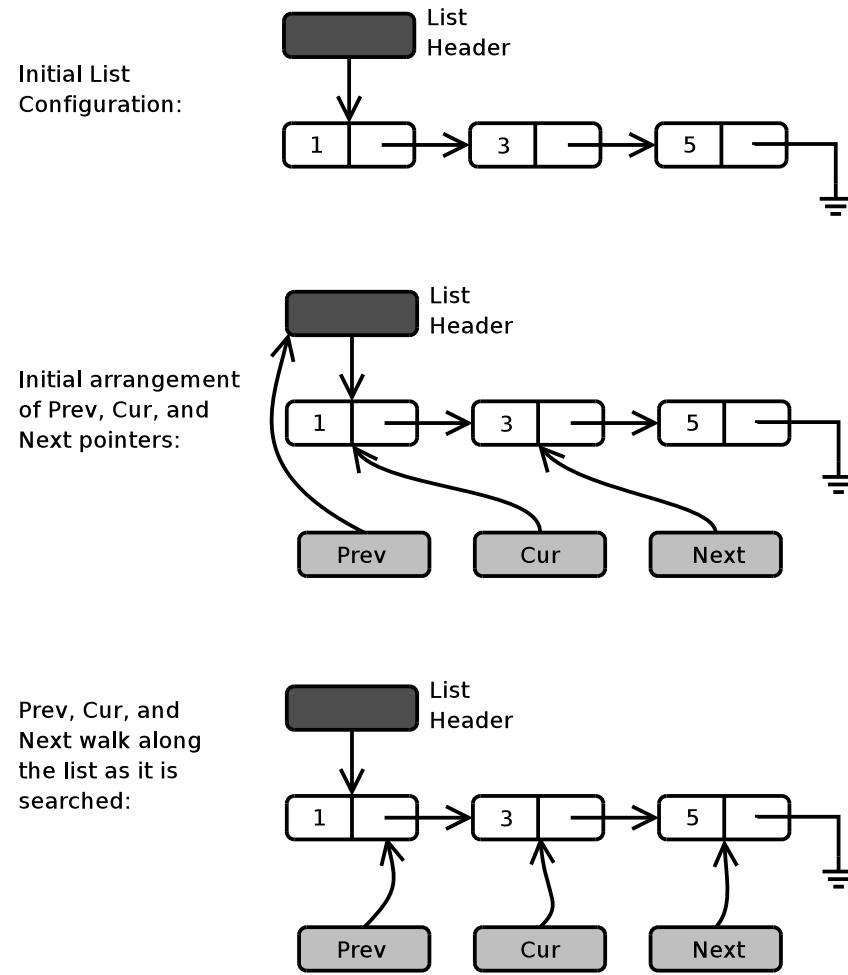


Figure 3.3: Example operation of search function for lock-free linked list.

therefore present outlines of the algorithms for lock-free linked lists and queues.

Lock-free Linked Lists

We use the lock-free linked list presented in [47], which is an improvement by Michael on an original design by Harris [22]. The list stores its keys in sorted order, and does not allow duplicate keys. It is singly-linked, *NULL*-terminated, and has a head node. The supported operations are searching the list for a node with a given key, deleting a node with a given key, and inserting a new node with a given key.

Figure 3.3 shows an example of a search for the key 2 in such a list, and Figure 3.4

```

node **prev;
node *next;
node *cur;

int search (node **head, long key)           5
{
    try_again:
        prev = head;
        cur = *prev;
        while (cur != NULL) {                  10
            if (*prev != cur) goto try_again;
            next = cur->next;
            /* If the low-order bit is a 1, the node is marked to be logically deleted. */
            if (next & 1) {
                /* Update the link and logically delete the node. */
                if (!CAS(prev, cur, next-1)) goto try_again;
                schedule_for_deletion(cur);
                cur = next-1;
            } else {                           15
                if (*prev != cur) goto try_again;
                if (cur->key >= key) {
                    return (cur->key == key);
                }
                prev = &cur->next;
            }
        }
    return (0);                           20
}

```

Figure 3.4: Pseudocode for search function for lock-free list, stripped of memory reclamation code.

shows the associated pseudocode. The searching thread walks the list, and as it does so, it keeps track of the current node, the *next* pointer of the current node's predecessor (or the *head* pointer if the current node is the first in the list), and the current node's successor. These three references are used by CAS operations in the insertion and deletion routines. The thread stops once it encounters the first key greater than or equal to the key for which it is searching (see lines 21-23 of Figure 3.4).

If a searching thread encounters a partially-deleted node, it must attempt to help complete this deletion, and then restart its traversal (lines 13-18; see the explanation of the deletion function, below). The thread also restarts its traversal if it detects that a node has been inserted between the current node and its predecessor (line 20). If such an insertion takes place after the check in line 20, and the caller of the search function is an inserting or deleting thread, the calling thread will have to invoke the search function again; the check serves to decrease the chances of this occurring.

More formally, the search function finds a pointer to a pointer to a node, *prev*, and pointers to nodes *cur* and *next* such that:

- If list is empty, *prev* = *head*, *cur* = *NULL*, and the search returns *false*.
- If the key is not in the list, then *prev* points to the *next* pointer of the last node in the list, *cur* = *NULL*, *next* = *NULL*, and the search function returns *false*.
- If there exists some node *q* in the list such that:
 - *q* has not been marked for deletion (see the explanation of the delete function, below), and
 - $q \rightarrow key \geq key$ and either *q* is the first node in the list or $q \rightarrow prev \rightarrow key < key$,

then $*prev = cur$, $cur = q$, $next = q \rightarrow next$, and the search returns *true* if and only if $q \rightarrow key = key$.

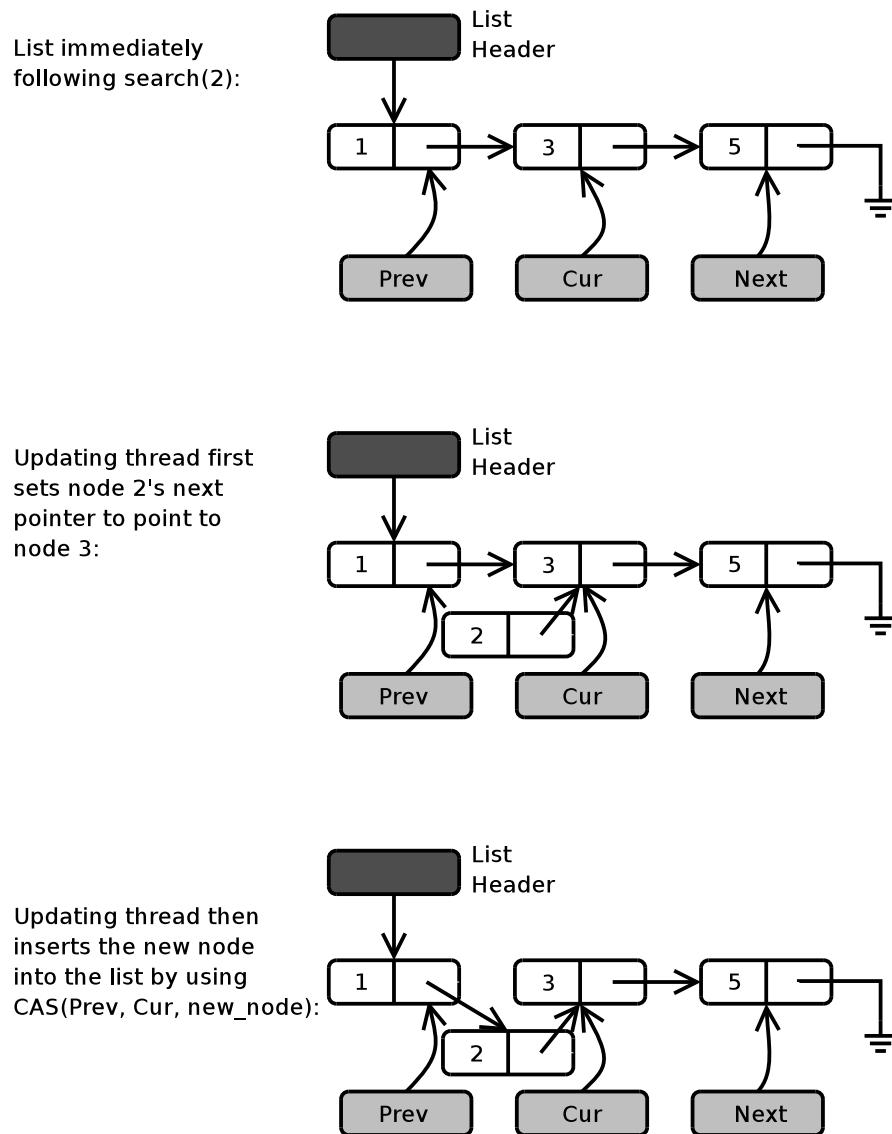


Figure 3.5: Example operation of insert function for lock-free linked list.

An insertion of a node with key 2 is shown in Figure 3.5. The insertion begins by searching for a node with key 2, as shown in Figure 3.3. This search ensures that we do not insert a duplicate key; furthermore, if no node with key 2 is found, it gives us the position in which to insert the new node with key 2 in order to keep the list sorted. We then initialize the new node’s *next* pointer to point to the node with key 3 in Figure 3.3, and update the *next* pointer of the node with key 1 using CAS, thereby linking the new node into the list. If the CAS operation fails, then we must restart the insertion from the beginning. Note that the steps of the insert operation *must* be performed in this order, or else readers may follow the *next* pointer of the node with key 2 before it has been initialized, leading to indeterminate behavior.

Figure 3.7 shows a deletion of the node with key 3 from the linked list. Deletion is slightly more complex than insertion, since we must prevent the possibility of concurrent insertions and deletions corrupting the list, as shown in Figure 3.6. To do so, we first search for the node with the key we wish to delete, and mark the low-order bit of this node’s *next* pointer using CAS. This is possible on current architectures because words lie on 4-byte boundaries; hence, the low-order two bits of any pointer are always zero. Marking the low-order bit using CAS will either fail or cause concurrent insertions which would otherwise corrupt the list, such as that shown in Figure 3.6, to fail. After marking the low-order bit, we then use CAS again to unlink the node from the list, thus completing the logical deletion.

Full details on the algorithm, including proofs of correctness, are available in [22], [43], and [47].

Lock-free Queues

Our lock-free queue is that presented in [50] and [47]; our code is structured according to the pseudocode given in the latter. A queue is represented by a singly-linked list with *head* and *tail* pointers. The implementation of the queue uses a dummy node which is

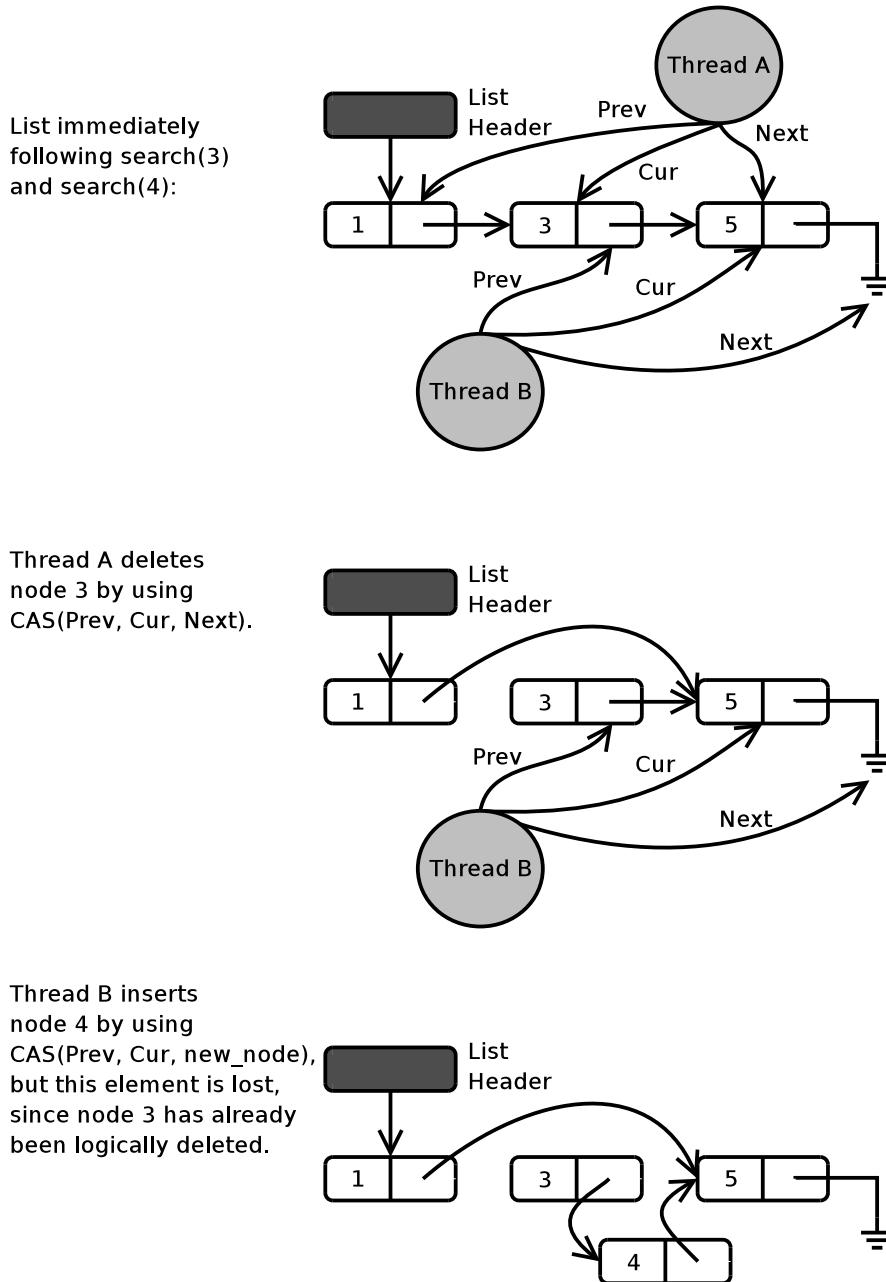


Figure 3.6: Error which can occur in a naïve lock-free linked list implementation when insertions and deletions are interleaved. To prevent such errors, the lock-free linked list must mark a node's *next* pointer before deleting the node (Figure 3.7).

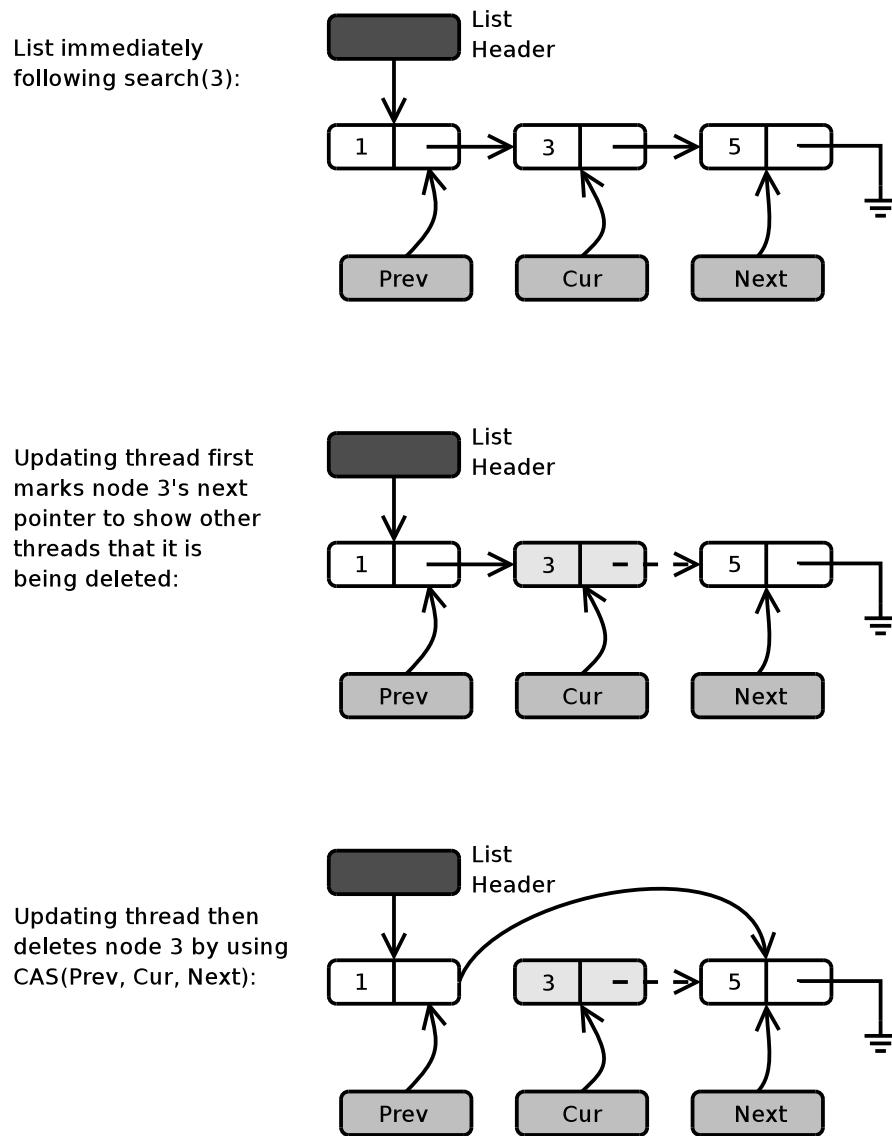


Figure 3.7: Example operation of delete function for lock-free linked list.

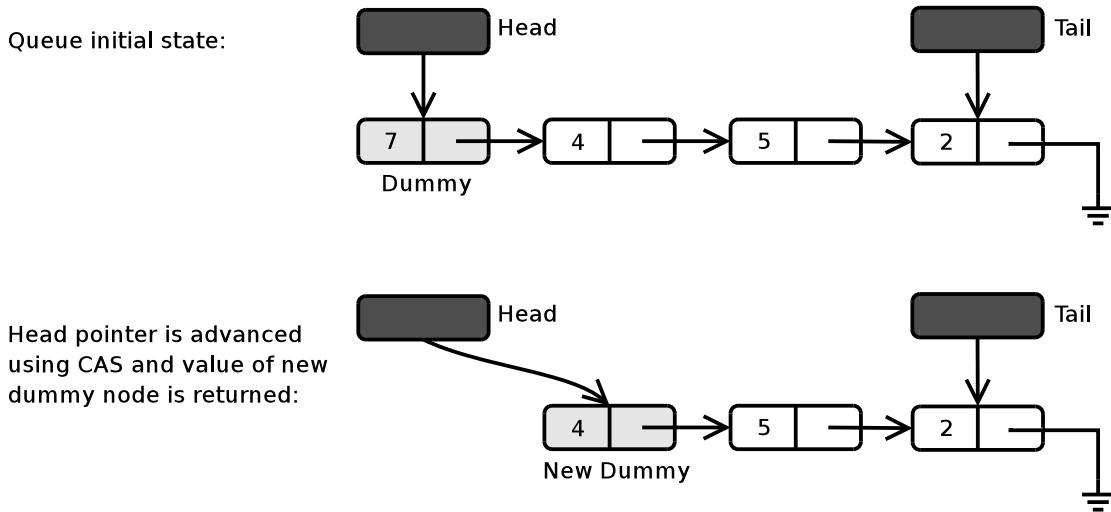


Figure 3.8: Dequeue from non-empty lock-free queue.

not logically part of the queue; hence, the *head* and *tail* pointers always have something to which to point. The dummy node is always either a node created for this purpose when the queue is initialized, or the most recently dequeued node. The *head* pointer must always point to the dummy node, and the *tail* pointer must always point to either the last or second-last node in the queue. Enqueue operations append nodes to the *tail* of the list, and dequeue operations remove nodes from the *head* of the list.

Figure 3.8 illustrates a dequeue operation, and Figure 3.9 shows the associated pseudocode. The dequeue operation begins by taking a consistent snapshot of pointers to the *head* (dummy) node and its successor, and to the *tail* node (lines 7-15 of Figure 3.9). If the only node in the queue is the dummy node, then the queue is empty; otherwise, if the *tail* pointer points to the dummy node, the dequeuing thread must attempt to advance the *tail* pointer using CAS, and then retry (lines 17-25). The thread then advances the *head* pointer to point to the dummy node's successor (lines 30-33); this node then becomes the new dummy node, and the old dummy node can be logically deleted. The new dummy node's key is then returned to the calling function.

An enqueue operation is shown in Figure 3.10, with the associated pseudocode shown

```

long dequeue(struct queue *Q)
{
    node *h, *t, *next;
    long data;
while (1) {
    /* Get the old head and tail nodes. */
    h = HEAD(Q);
    t = TAIL(Q);
    /* Get the head node's successor. */
    next = h->next;
    memory_barrier();
    if (HEAD(Q) != h)
        continue;
    /* If the head (dummy) node is the only one, return EMPTY. */
    if (next == NULL)
        return EMPTY_SENTINEL;
    /* There are multiple nodes. Help update tail if needed. */
    if (h == t) {
        CAS(&TAIL(Q), t, next);
        continue;
    }
    /* Save the data of the head's successor. It will become the new dummy node. */
    data = next->key;
    /* Attempt to update the head pointer so that it points to the new dummy node. */
    if (CAS(&HEAD(Q), h, next))
        break;
}
/* The old dummy node has been unlinked, so reclaim it. */
schedule_for_deletion(h);

return data;
}

```

Figure 3.9: Pseudocode for dequeue function for lock-free queue, stripped of memory reclamation code.

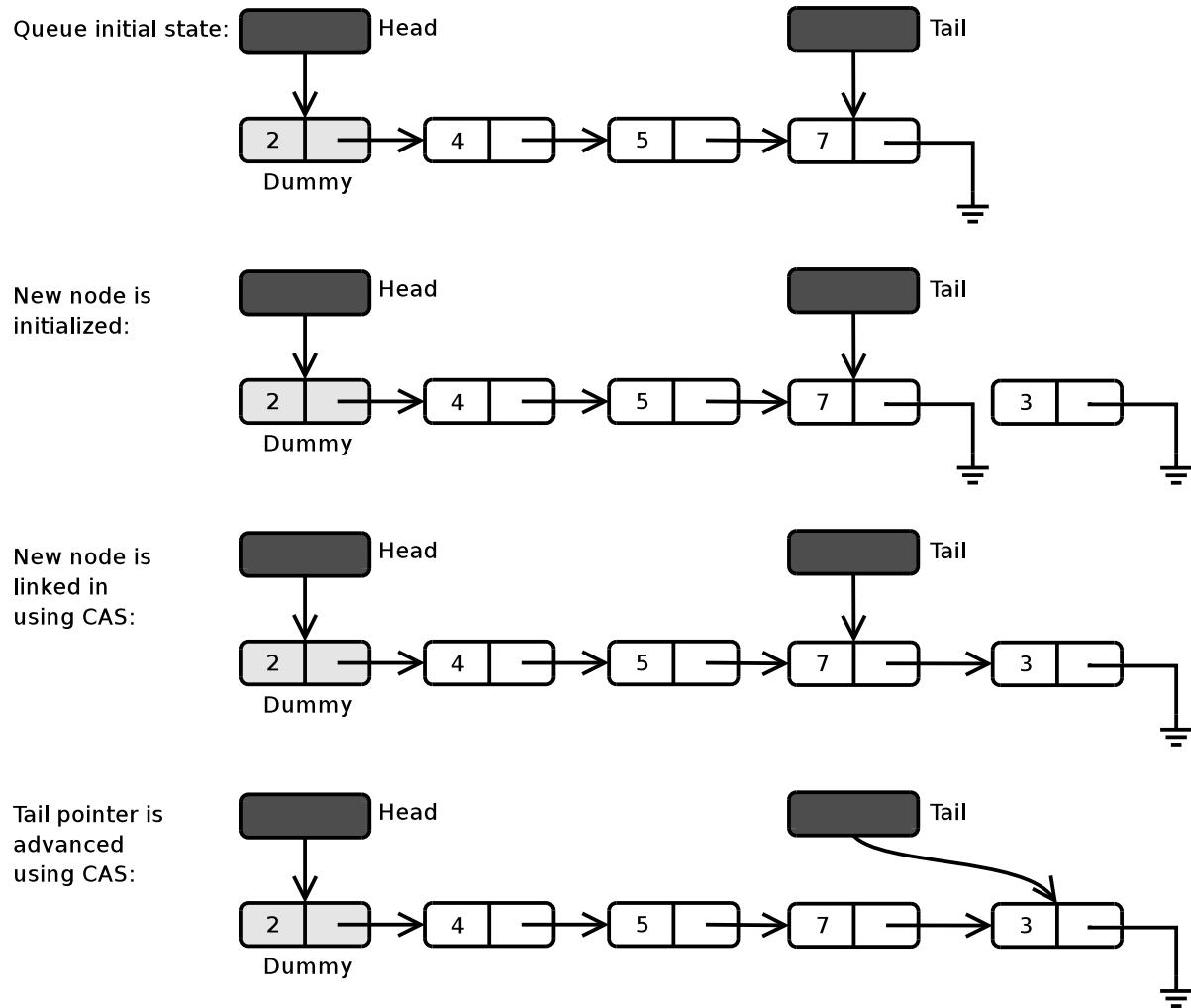


Figure 3.10: Enqueue to non-empty lock-free queue.

```

void enqueue(long data, struct queue *Q)
{
    node *newnode = allocate_new_node();
    node *t, *next;
    /* Initialize the new node. */
    newnode->key = data;
    newnode->next = NULL;
    /* Ensure that newnode->next = NULL before inserting it. */
    write_barrier();                                10

    while(1) {
        /* Snapshot the old tail pointer and its successor. */
        t = TAIL(Q);
        next = t->next;
        if (TAIL(Q) != t)
            continue;
        /* Help update the tail pointer if needed. */
        if (next != NULL) {
            CAS(&TAIL(Q), t, next);
            continue;
        }                                              15
        /* Attempt to link in the new node. */
        if (CAS(&t->next, NULL, &newnode))
            break;
    }                                              25
    /* Swing the tail to the new node. */
    CAS(&TAIL(Q), t, &newnode);
}

```

Figure 3.11: Pseudocode for enqueue function for lock-free queue, stripped of memory reclamation code.

in Figure 3.9. First, the enqueueing thread allocates a new node, stores the key to be enqueued in it, initializes its *next* pointer to *NULL*, and executes a write fence (lines 3-11 of Figure 3.9). The thread then takes a snapshot of pointers to the *tail* node and its successor (lines 14-18). If the *tail* node's successor is not *NULL*, then the *tail* pointer must be pointing to the second-last node in the queue, so the thread attempts to advance the *tail* pointer, and then retries (lines 20-24). Next, the thread uses CAS to insert the new node at the end of the list; if this CAS fails, the thread must retry (lines 26-28). Once the CAS succeeds, a second CAS operation attempts to advance the queue's *tail* pointer to point to the new node (line 32); no failure condition is needed on this latter CAS, since it fails only if another thread has already succeeded in updating the *tail* pointer.

As with the lock-free list, full details, including a proof of correctness, are available in [51] and [47].

We note that both of these lock-free algorithms are quite complex implementations of relatively simple data structures. Much of this complexity is due to the fact that lock-free algorithms must coordinate multiple concurrent updates. Algorithms accommodating multiple readers, but only a single writer, can be significantly simpler.

3.3 Concurrently-Readable Algorithms

The most well-known use of concurrently-readable algorithms is in implementing read-copy update [38, 42, 39, 40, 41, 6]. These algorithms focus on concurrently-readable linked lists and chaining hash tables, although other applications of read-copy update exist [38]. We note that we examine concurrently-readable chaining hash tables in our experiments, and that these hash tables are simply arrays of concurrently-readable linked lists; therefore, we present an outline of these lists.

Figure 3.12 shows an example from [42] of an update to a node of a linked list which

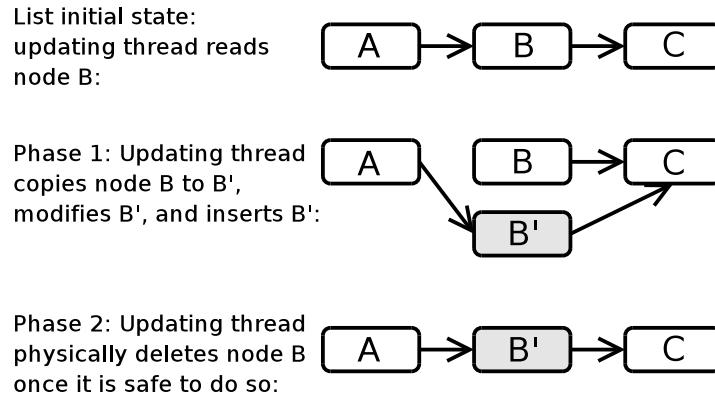


Figure 3.12: Concurrently-readable node modification example from which *read-copy update* derives its name.

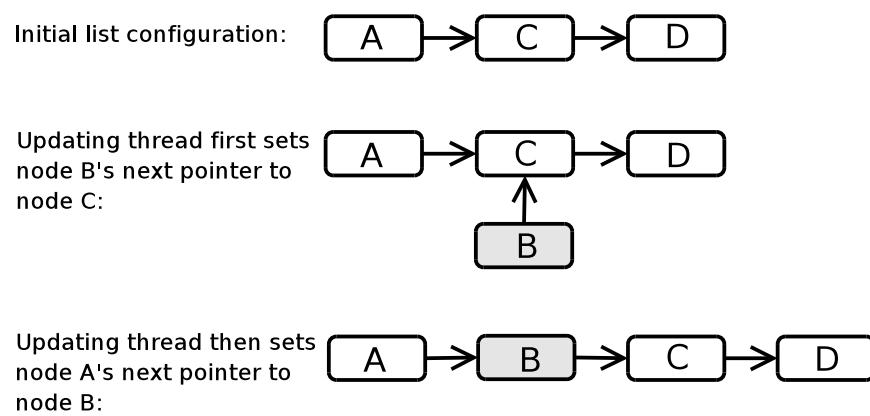


Figure 3.13: Concurrently-readable insertion.

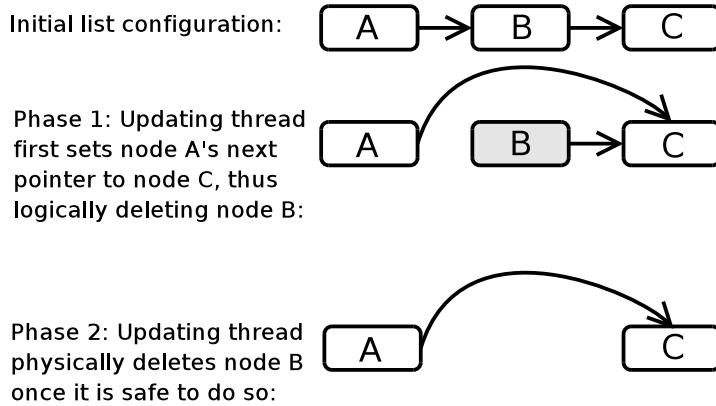


Figure 3.14: Concurrently-readable deletion.

can run concurrently with lockless reads; insertions and deletions are slightly simpler, and are shown in Figures 3.13 and 3.14. We focus on the example of the update shown in Figure 3.12. Readers merely traverse the list as they would if the program were single-threaded. A writer acquires a per-list spinlock, so writers need not deal with concurrent writes. Writes then proceed in two phases. First, the updating thread makes a copy of the node to be updated, and performs all needed modifications to the copy. The copy's *next* pointer is then made to point to the original node's successor, then, the *next* pointer of the original's predecessor must be updated to point to the modified node. As with the lock-free linked list algorithm, the updates *must* be performed in this order, or else lockless reads may follow the updated node's *next* pointer before it has been initialized. In the second phase of the update, the writer physically deletes the original node once it is safe to do so.

In a theoretical model with infinite memory, the second phase would be unnecessary and the algorithm would be trivial. The algorithm becomes more interesting when examined in a practical setting and combined with a high-performance memory reclamation scheme [42].

Chapter 4

Memory Reclamation Schemes

In this chapter, we present our three memory reclamation schemes — EBR, QSBR, and SMR, along with other schemes which we did not consider. For the schemes which we do consider, we show how they can be applied, and compare them analytically. The next chapter provides experimental validation of this analysis.

4.1 Descriptions of Schemes

This section details the three memory reclamation schemes under consideration: quiescent-state-based reclamation (QSBR), safe memory reclamation (SMR), and epoch-based reclamation (EBR). We also discuss reference counting, Greenwald’s type-stable memory, and Pass the Buck, and explain why we did not consider these schemes. Since all these methods have been published elsewhere [42, 6, 44, 47, 17], we discuss them in only enough detail for the reader to understand our work.

4.1.1 Blocking Methods

We describe three blocking memory reclamation schemes: epoch-based reclamation, quiescent-state based reclamation, and Greenwald’s type-stable memory. These methods

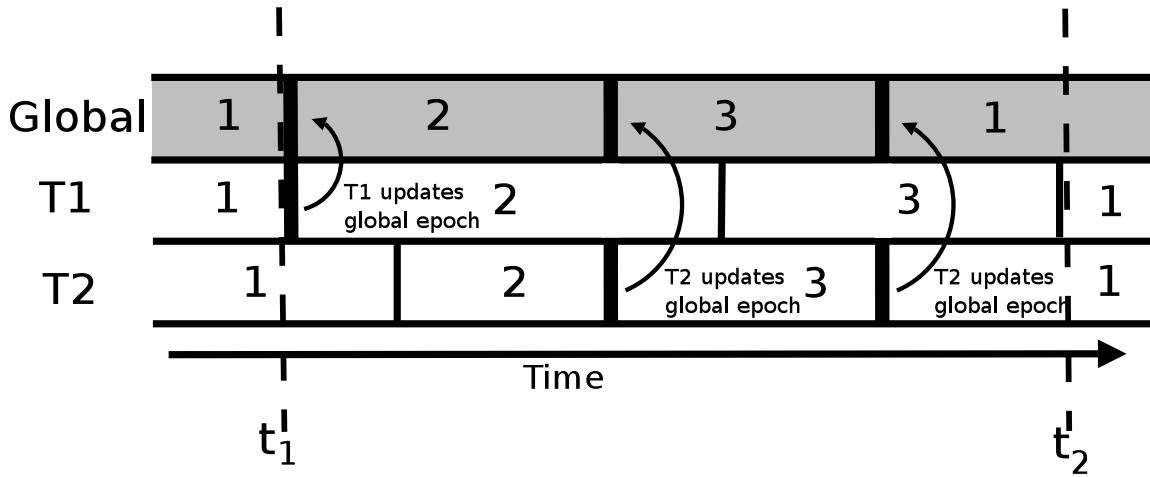


Figure 4.1: Illustration of EBR. Threads follow the global epoch. If a thread observes that all other threads have seen the current epoch, then it may update the global epoch. Hence, if the global epoch is e , threads in critical sections can be in either epoch $e + 1$ or e , but not $e - 1$ (all mod 3). The time period $[t_1, t_2]$ is thus a grace period for thread T_1 .

are all blocking, because they force threads to wait for some condition, which could be delayed arbitrarily, to become true, and therefore place no upper bound on the amount of unfreed memory at any given time. Since the amount of unfreed memory is unbounded, the system may run out of memory, thus causing threads to block on memory allocation and therefore fail to make progress. Figure 4.7 of section 4.3, below, shows how blocked threads can obstruct memory reclamation.

Epoch-Based Reclamation

Epoch-based reclamation (EBR) was introduced by Fraser [17], but builds on earlier ideas [31, 36, 53]. At any point in time, each thread is executing in one of three logical epochs. For each of the three epochs, the thread has an associated *limbo list* which holds logically deleted nodes awaiting physical deletion. When a thread T is in epoch e , it places all nodes that it logically deletes in limbo list e . T may physically delete these

nodes once a *grace period* has passed. A grace period $[a, b]$ is an interval of program execution time such that, after point b , all nodes logically deleted before point a can be physically deleted safely. EBR uses epochs to detect grace periods, as explained below.

EBR is illustrated in Figure 4.1. At the start of any lock-free operation, a thread enters a *critical section* with respect to memory reclamation (note that this use of the term *critical section* has nothing to do with mutual exclusion). Upon entering a critical section, the thread updates its local epoch to match the global epoch if the two epochs differ, as indicated by the thinner lines in Figure 4.1. It also sets a per-thread flag indicating to other threads that it is in a critical section. Upon exit of a critical section, a thread clears its flag. No thread is allowed to access an EBR-protected object outside of a critical section.

Upon entering a programmer-determined number of critical sections since seeing the global epoch change, a thread may attempt to update the global epoch. If any thread which is in a critical section has not updated its local epoch to match the global epoch, then this attempt to update the global epoch must fail. EBR therefore guarantees that at any time t , if the global epoch is e , the local epoch of each thread in a critical section is either e or $e + 1$, but not $e - 1$ (all mod 3). As a result, whenever a thread sets its local epoch to e , it can physically delete all nodes logically deleted the last time that it was in epoch e , since all operations which could have held a reference to the logically deleted node have completed.

EBR is completely encapsulated within a library, and is invisible to the application programmer. Further, threads that are not in critical sections can not obstruct the progress of EBR. These factors make EBR very generally applicable, and easy for a programmer to use.

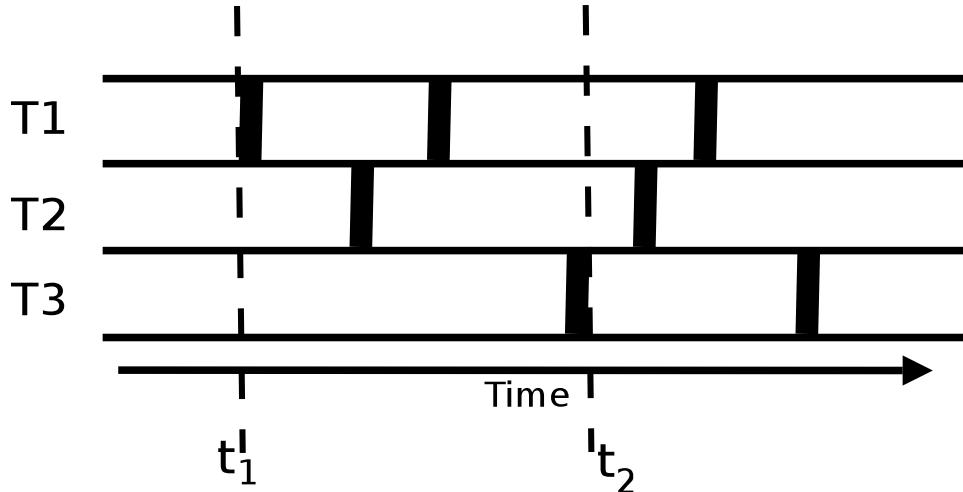


Figure 4.2: Illustration of QSBR. Thick lines represent quiescent states. The time interval $[t_1, t_2]$ is a grace period: at time t_2 , each thread has passed through a quiescent state since t_1 , so all nodes logically removed before time t_1 can be physically deleted.

Quiescent-State-Based Reclamation

Instead of dividing time into epochs, QSBR has the programmer identify quiescent states in the application code. A *quiescent state* for thread T is a point in T 's program code at which T can hold no reference to any shared node; hence, from T 's point-of-view, all nodes logically deleted by other threads can safely be physically deleted. A *grace period* for QSBR is an interval of execution time during which each thread passes through at least one quiescent state. QSBR is illustrated in Figure 4.2.

QSBR must enable threads to detect grace periods so that they can physically delete logically deleted nodes. However, no QSBR implementation is required to detect the smallest grace periods possible. Furthermore, unlike with EBR, the definition of a quiescent state is application-dependent. Natural and convenient quiescent states exist for many operating system kernels — the domain in which quiescent-state-based reclamation is used to implement read-copy update [42, 40, 18].

The fact that QSBR is application-dependent is the fundamental difference between QSBR and EBR. EBR, by definition, detects grace periods at the library level. QSBR,

by contrast, requires that the application report quiescent states to the QSBR library. As we show in Section 5.2, this gives QSBR a significant performance advantage over EBR.

Type-Stable Memory Management

EBR and QSBR both guarantee that a node is never physically deleted unless no thread can hold a reference to it. Type-stable memory (TSM) [20, 19] makes a weaker guarantee: a node’s memory *cannot be re-used for an object of another type* until no thread can hold a reference to it.

Greenwald [19] outlines both kernel-level and user-level implementations of TSM. The kernel-level implementation relies on “safe points” which are equivalent to quiescent states. The user-level version uses per-type reference counters.

Like EBR and QSBR, Greenwald’s TSM implementations are blocking, and hence suffer from the same drawbacks. Furthermore, TSM places additional burdens on the programmer, such as having to check after finding a node that the node has not been reallocated and inserted into another data structure. Such checks would add programming complexity and performance overhead to lockless linked list searches. Due to this disadvantage, and Greenwald’s TSM’s lack of any apparent advantages relative to EBR and QSBR, we do not consider it in our experiments.

4.1.2 Lock-free Methods

Here, we present the three lock-free memory reclamation schemes of which we are aware: reference counting, safe memory reclamation, and Pass the Buck.

Reference Counting

Implementations of lock-free reference counting have been proposed by Valois [61] (corrected by Michael and Scott [50]), Sundell [59], and Detlefs et al. [11, 12]. Valois’

scheme uses compare-and-swap (CAS) and fetch-and-add instructions to manage reference counts, and requires that nodes retain their type after deletion. Sundell’s scheme is based on Valois’, but is wait-free. The method of Detlefs et al. allows the memory used by nodes to be re-used for structures of other types, but requires the double-compare-and-swap operation, which no current architecture supports in hardware.

Reference counting has been shown by Michael [47] to introduce performance overhead which makes lock-free algorithms perform worse than their lock-based counterparts in most situations. We thus omit reference counting from our experiments.

Safe Memory Reclamation

Safe memory reclamation was introduced by Michael [44]. It provides a simple and intuitive existence locking mechanism for dynamic nodes. Each thread which accesses a lock-free or concurrently-readable data structure has K hazard pointers which it uses to protect nodes from deletion by other threads. The required number of hazard pointers, K , is algorithm-dependent, and is typically very small: queues and linked lists need $K = 2$ hazard pointers, while stacks require only $K = 1$. If the total number of threads in the system that may access an SMR-protected data structure is N , then we need $H = NK$ hazard pointers in total.

When a thread T removes a node from a dynamic data structure, it places a reference to that node in a private list. When the list grows to size R , the thread attempts to physically delete all nodes in the list. R is a constant chosen by the programmer; a higher value of R will mean that memory remains unfreed for a longer period of time, but memory reclamation overhead will be amortized over a larger number of operations. However, to ensure that the expected amortized processing time per reclaimed node is kept constant, R must be chosen such that $R = H + \Omega(H)$.¹

¹The terminology $R = H + \Omega(H)$ is confusing to some. Roughly, this means that R should be parameterized by H , the total number of hazard pointers, and that R must always be greater than H by an amount which is at least linear in H . Hence choosing $R = aH + b$ where $a > 1$ and b is a constant

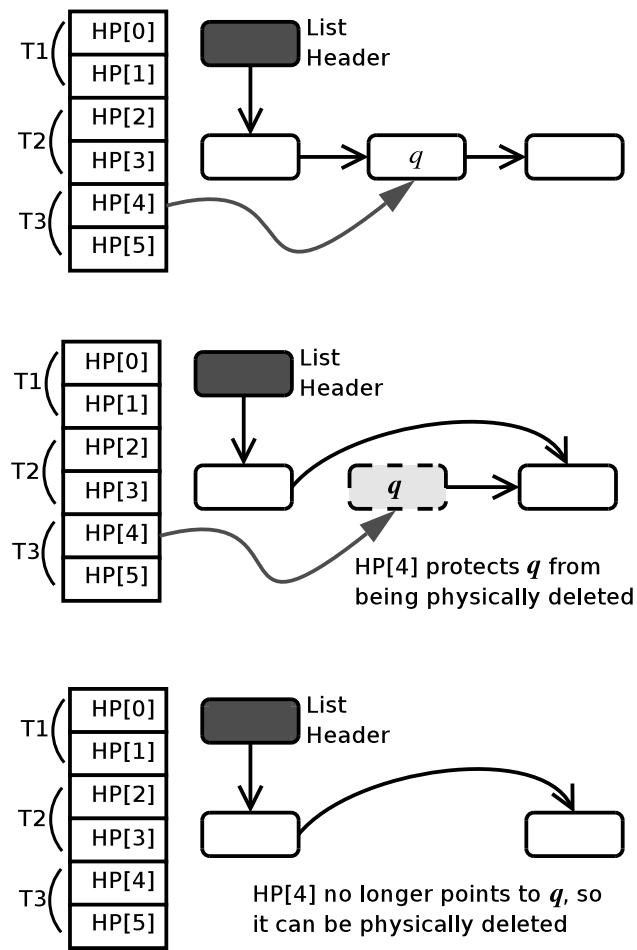


Figure 4.3: Illustration of SMR. q is logically removed from the linked list on the right of the diagram, but cannot be physically deleted while $T3$'s hazard pointer $HP[4]$ is still pointing to it.

To physically free nodes, T copies all non-NULL hazard pointers of all threads into a private array, which it then sorts. For each node n in T 's private list of nodes to be reclaimed, T does a binary search for a pointer to n , and physically deletes n if no such pointer is found.

To use hazard pointers, an algorithm must identify all *hazardous references* — references to shared objects that may have been deleted by other threads (or are vulnerable to the ABA problem if hazard pointers are being used for ABA-protection) [47] — in all lockless operations. Before using any such reference, a hazard pointer must be made to point to the target object. After setting the hazard pointer, an algorithm-specific check must be made in order to ensure that the protected object has not been deleted; the rules of the SMR deletion routine outlined above guarantee that the object will never be deleted so long as the hazard pointer continues to point to it. Figure 4.3 illustrates the use of SMR.

One cited advantage of SMR is that it requires only atomic reads and writes, and is therefore usable on hardware platforms which do not support CAS or LL/SC [47]. Since most current hardware platforms support one of these strong synchronization primitives, the advantage of not requiring these operations lies mostly in avoiding their significant performance cost.

Pass the Buck

Herlihy et. al. [28, 27] present Pass the Buck, a solution to the *Repeat Offender Problem*, which is an attempt to formalize the problem of lock-free memory reclamation. Pass the Buck is similar to SMR, but uses expensive CAS operations while physically deleting nodes, and lacks SMR's amortized bound on the memory reclamation overhead per physically deleted node. On the upside, Pass the Buck has a property called *value progress*, which guarantees that logically deleted nodes will eventually be freed, even if there are

suffices.

thread failures.

We do not consider value progress attractive enough to justify Pass the Buck's higher overhead relative to SMR, so we do not include Pass the Buck in our experiments; however, due to Pass the Buck's similarity to SMR, we believe that our experimental analysis of SMR relative to QSBR and EBR would be applicable to Pass the Buck as well.

4.2 Applying the Schemes

As explained in Chapter 3, we examined three algorithms requiring deferred memory reclamation: a concurrently-readable chaining hash table, a lock-free chaining hash table, and a lock-free queue. We found that all three of these algorithms were compatible with each of our three memory reclamation schemes.

We illustrate this compatibility by way of the lock-free queue's `dequeue()` method. We chose this method because it is the simplest method which demonstrates the use of these schemes. Figures 4.4, 4.5, and 4.6 demonstrate the use of SMR, QSBR, and EBR, respectively. Since this is actual code and not pseudocode, some conventions must be explained. Our nodes are of type `struct el`. Lists are implemented using the doubly-linked list interface of the Linux kernel [35]: each node contains an instance of `struct list_head`, which contains two pointers: `struct list_head *prev` and `struct list_head *next`. The function `list_entry()` maps an instance of `struct list_head*` to a pointer to the `struct el` which contains it. Hazard pointers are implemented using a cacheline-aligned structure, `struct hazard_pointer`, which has one member: `struct el *p`.

SMR, QSBR, and EBR all register *callbacks* for logically deleted nodes. A callback is simply a record of the logically deleted node, and the function to be used to physically delete it once it is safe to do so; in our experiments, this function is always `kfree()`. Our

SMR implementation hides our callback interface within the body of the `retire_node()` function; however, the interface is exposed in our QSBR and EBR implementations, as shown in line 39 of Figure 4.5 and line 40 of Figure 4.6, respectively. We ask the reader to forgive this minor inconsistency in our interfaces.

The code in Figure 4.4, which illustrates the use of SMR, is the most complex of the three versions of the `dequeue()` function. For convenience, it uses two pointers to hazard pointers, `hp0` and `hp1`, which are first set to point to the two hazard pointers owned by the calling thread (lines 7-9 of Figure 4.4). After making a copy of the value of the queue's head pointer, we protect the head from deletion using a hazard pointer (lines 12-14). We then execute a fence instruction, after which we ensure that the head has not changed - and hence possibly been deleted (lines 15-17). Once we are sure that this has not happened, we know that the head will not be physically deleted until our hazard pointer is unset. A similar step must be taken after acquiring a pointer to the head node's successor (lines 22-27). If, after setting either of these hazard pointers, we find that the head of the queue has changed, we must retry our dequeuing attempt (lines 17 and 27).

Once we have successfully logically deleted the dequeued node, we can retire it using the `retire_node()` function (line 50), and unset our hazard pointers so that the nodes they pointed to can be reclaimed if necessary (line 51).

The code illustrating QSBR in Figure 4.5 is much simpler. Since reclamation works by keeping track of quiescent states at the application level, the code for the `dequeue()` method is not burdened by any memory reclamation code. The code in Figure 4.6, which illustrates the use of EBR, is almost identical. The only differences are that the code for EBR places calls to `critical_enter()` and `critical_exit()` at the beginning and end of the method, respectively (lines 7 and 41 of Figure 4.6), and that it schedules logically deleted nodes for physical deletion using `call_epoch_kfree()` instead of `call_rcu_kfree()` (line 40 of Figure 4.6 and line 39 of Figure 4.5).

```

long dequeue(struct queue *Q)
{
    struct list_head *h, *t, *next;
    struct el *node;
    long data;                                5

    /* Initialize our hazard pointer pointers. */
    hp0 = &(HP[smp_processor_id()]*K).p;
    hp1 = &(HP[smp_processor_id()]*K+1]).p;      10

    while (1) {
        /* Protect the old head node. */
        h = HEAD(Q);
        *hp0 = list_entry(h, struct el, list);
        memory_barrier();                         15
        if (HEAD(Q) != h)
            continue;

        /* Get a pointer to the old tail node. */
        t = TAIL(Q);                            20

        /* Get and protect the head node's successor. */
        next = h->next;
        *hp1 = list_entry(next, struct el, list);
        memory_barrier();                         25
        if (HEAD(Q) != h)
            continue;

        /* If the head (dummy) node is the only one, return EMPTY. */
        if (next == NULL)                      30
            return -1; /* Empty. */

        /* There are multiple nodes. Help update tail if needed. */
        if (h == t) {                          35
            CAS(&TAIL(Q), t, next);
            continue;
        }

        /* Save the data of the head's successor. It will become the
         * new dummy node. */
        data = list_entry(next, struct el, list)->key;      40

        /* Attempt to update the head pointer so that it points to the
         * new dummy node. */
        if (CAS(&HEAD(Q), h, next))           45
            break;
    }

    /* The old dummy node has been unlinked, so reclaim it. */
    retire_node(list_entry(h, struct el, list));          50
    *hp0 = *hp1 = NULL;
    return data;
}

```

Figure 4.4: Lock-free queue’s `dequeue()` function, using SMR.

```

long dequeue(struct queue *Q)
{
    struct list_head *h, *t, *next;
    struct el *node;
    long data;                                5

    while (1) {
        /* Get the old head and tail nodes. */
        h = HEAD(Q);
        t = TAIL(Q);                           10

        /* Get the head node's successor. */
        next = h->next;
        memory_barrier();
        if (HEAD(Q) != h)                      15
            continue;

        /* If the head (dummy) node is the only one, return EMPTY. */
        if (next == NULL)
            return -1; /* Empty. */                  20

        /* There are multiple nodes. Help update tail if needed. */
        if (h == t) {                            25
            CAS(&TAIL(Q), t, next);
            continue;
        }

        /* Save the data of the head's successor. It will become the
         * new dummy node. */
        data = list_entry(next, struct el, list)->key;          30

        /* Attempt to update the head pointer so that it points to the
         * new dummy node. */
        if (CAS(&HEAD(Q), h, next))                35
            break;
    }

    /* The old dummy node has been unlinked, so reclaim it. */
    call_rcu	kfree(new_callback(), list_entry(h, struct el, list));
    return data;                                40
}

```

Figure 4.5: Lock-free queue’s `dequeue()` function, using QSBR.

```

long dequeue(struct queue *Q)
{
    struct list_head *h, *t, *next;
    struct el *node;
    long data;                                5

    critical_enter();

    while (1) {
        /* Get the old head and tail nodes. */
        h = HEAD(Q);                         10
        t = TAIL(Q);

        /* Get the head node's successor. */
        next = h->next;                      15
        memory_barrier();
        if (HEAD(Q) != h)
            continue;

        /* If the head (dummy) node is the only one, return EMPTY. */
        if (next == NULL)                    20
            return -1; /* Empty. */

        /* There are multiple nodes. Help update tail if needed. */
        if (h == t) {                        25
            CAS(&TAIL(Q), t, next);
            continue;
        }

        /* Save the data of the head's successor. It will become the
         * new dummy node. */
        data = list_entry(next, struct el, list)->key;          30

        /* Attempt to update the head pointer so that it points to the
         * new dummy node. */
        if (CAS(&HEAD(Q), h, next)) break;                  35
    }

    /* The old dummy node has been unlinked, so reclaim it. */
    call_epoch_kfree(new_callback(), list_entry(h, struct el, list));  40
    critical_exit();
    return data;
}

```

Figure 4.6: Lock-free queue’s `dequeue()` function, using EBR.

The fact that each of the three memory reclamation methods is compatible with each of the the three algorithms we considered is important. To date, the literature on QSBR has concentrated on using QSBR with locking methods, usually a variant of the concurrently-readable linked list described in section 3.3; furthermore, the literature concentrates on using QSBR to support lockless reads. We show, by using QSBR with lock-free queues, that it makes sense to use QSBR even with data structures which do not have read-only operations.

One caveat about using QSBR or EBR for memory reclamation concerns the ABA problem, which was explained in section 2.1.3. SMR can be used to make an algorithm ABA-safe [46]. To make an algorithm ABA-safe using QSBR or EBR, we must ensure that we do not re-insert a node which has been removed from a data structure until the node has been physically deleted and reallocated. Since most node implementations consist of only a pointer to the next node and a pointer to the node’s data, allocating new nodes to hold data which must be re-inserted into a data structure should be relatively-inexpensive; hence, this constraint should not cause programmers undue difficulties.

We note that not all algorithms are compatible with all memory management techniques. Some, such as the deques of [58] and Harris’ original version of our lock-free linked list [22], must reference logically-deleted nodes. For such algorithms, neither SMR, EBR, nor QSBR is usable; these algorithms typically use reference counting. Other algorithms, such as the deques of [10], are also SMR-incompatible and assume automatic garbage collection. These incompatibilities are detailed in [44]. The existence of such incompatibilities prevents us from claiming that the choice of memory reclamation scheme is completely independent of the target algorithm; instead, based on our successful implementations, we claim only that the two are *mostly* independent.

4.3 Analytic Comparison of Methods

Here we lay out an analytic comparison of the memory reclamation strategies under consideration, which we validate with performance data from our experiments in Section 5.2. We identify the following factors which could affect the performance of our memory reclamation schemes:

- **Object contention:** Many threads may attempt to access or modify the same object concurrently; in the case of updates, these operations may conflict with one another, thus forcing one or more threads to retry. Having more threads performing operations on the same number of objects will increase object contention.
- **CPU contention:** If there are more threads than there are physical CPUs, some threads will be descheduled for periods during which other threads may perform large numbers of operations. A descheduled thread may delay the progress of other threads under blocking memory reclamation schemes.
- **Workload:** Objects typically support several operations such as search, insert, delete, enqueue, and dequeue. The *workload*, in this paper, is the proportion of each type of operation in a given experiment. If an object has read-only operations, we use the term *update fraction* to refer to the proportion of the operations invoked on the object which are not read-only.
- **Traversal length:** In the case of structures such as linked lists and trees, a process performing a search operation will have to traverse several nodes; we refer to the number of nodes accessed as the *traversal length*.
- **Execution environment:** External factors such as the memory allocator and OS scheduler.

When object contention, CPU contention, and traversal length are low, and the workload is read-mostly, QSBR should have considerably less per-operation runtime overhead

than either SMR or EBR. This is not obvious until we recall that we are working in a weakly-consistent memory model in which fences are necessary. As we show in Section 5.2, these fence instructions make SMR and EBR more expensive than QSBR in most situations. In the case of SMR, a fence is necessary between the setting and the validation of any hazard pointer, since no hazard pointer can be validated until it has been set and is visible to all threads. EBR requires a flag to be set upon entry into any critical section, and cleared upon exit of the critical section. A fence is necessary after setting and before clearing the flag. QSBR has no per-operation code to manage quiescent states, and hence no fences are required. As a result, the per-operation overhead of using QSBR, in the best case, is very low.

We note that we could modify EBR so that critical sections are entered and exited at the application level instead of the library level, thus amortizing the overhead of the fence instructions across more operations. Doing so, however, would make EBR application-dependent. The point of comparing the performance of EBR to that of QSBR is to evaluate the performance benefits of using an application-dependent method to detect grace periods.

Traversal length will be the primary factor influencing the performance of SMR. While traversing a linked list, a thread must, for each node, set a hazard pointer, execute a fence instruction, and validate the hazard pointer. In a contention-free case in which the thread never has to restart its traversal, the number of fences needed will be $O(n)$, where n is the traversal length. If there is object contention on the linked list so that the thread may have to restart its traversal, the maximum number of fences required is unbounded. In contrast, EBR needs exactly two fences per operation, no matter how many times a thread may have to restart its traversal.

We can expect CPU contention to adversely affect QSBR and EBR. Descheduled threads can delay other threads' memory reclamation, as shown in Figure 4.7. In extreme cases, this could lead to out-of-memory errors, which could cause threads to block on

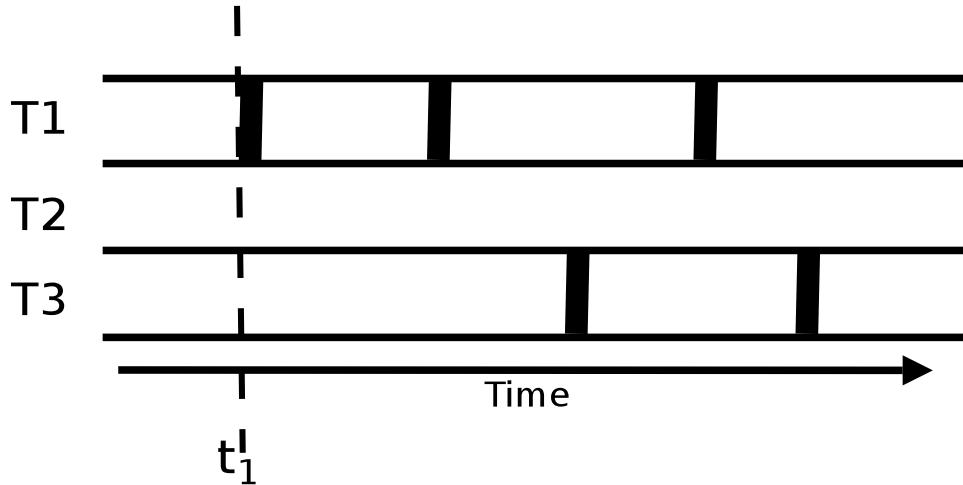


Figure 4.7: Illustration of why QSBR is inherently blocking. Here, thread T_2 does not go through a quiescent state for a long period of execution time; hence, threads T_1 and T_3 must wait to reclaim memory. A similar argument holds for EBR.

memory allocation, severely degrading performance. Furthermore, if there are locks in the memory allocator, these more memory-hungry methods may increase the contention on these locks. If a thread is descheduled while holding a lock on a global freelist, other threads will block on memory allocation. These effects will be most noticeable when the workload has a high percentage of operations which must allocate memory.

The choice of memory allocator and OS scheduler could also significantly impact QSBR and EBR. Although lock-free memory allocation is possible [49], most memory allocators use locking. The unbounded memory use of QSBR and EBR could cause an algorithm using these methods to need to allocate memory from a lock-protected global pool much more frequently than the same algorithm using SMR, therefore increasing the contention on the pool's lock. Furthermore, since thread delays can delay memory reclamation, the policy of the OS scheduler could play a huge part in how long memory is left unreclaimed. The only strategy that provides a provable bound on the amount of unfreed memory at any point in time is SMR [44]; we thus expect it should be less sensitive to the memory allocator and other factors in the external environment than QSBR and EBR.

Our analysis allows us to gain some intuition into the workings of these three memory reclamation schemes; however, we cannot analytically quantify the extent to which the factors outlined above impact each method. We must therefore evaluate these schemes experimentally.

Chapter 5

Experimental Evaluation

In this chapter, we describe our experiments. We first describe the setup we used, and then detail our results.

5.1 Experimental Setup

We evaluated our memory reclamation strategies on two data structures — a queue and a hash table — using systems based on PowerPC and IA-32 processors, while independently varying each of the factors outlined in Section 4.3. This section provides details on these aspects of our experiments.

5.1.1 Algorithms Compared

The hash table used in our experiments is an array of buckets, where each bucket has a linked list of keys. Duplicate keys are not allowed. The lock-free hash table implementation is that given in [47] and described in section 3.2.2, which, we reiterate, stores the keys in each hash chain in sorted order. We therefore stored the keys in the lists for the lock-based algorithms in sorted order as well, so that we could fairly compare the performance of these alternatives as we increased the load factor of the hash table.

We compared the lock-free hash tables to a version using per-bucket spinlocks, and the concurrently-readable version described in section 3.3. The lock-free queue is the version of Michael and Scott’s implementation given in [47] and described in section 3.2.2. We compare it to a simple spinlock-based queue.

The algorithms for lock-free queues, lock-free hash tables, and concurrently-readable hash tables were paired with each of the three memory reclamation schemes, for a total of nine combinations. We concentrate on the six combinations involving lock-free algorithms.

Spinlocks were implemented using the CAS operation and fence instructions. CAS is provided in hardware on IA-32, and implemented using LL/SC on PowerPC. No exponential back-off was used, since our primary interest is in cases of low CPU contention — that is, when the number of threads does not exceed the number of processors — in which back-off is unlikely to be useful.

5.1.2 Test Program

In our experiment, a parent thread creates n child threads, starts a timer, and stops the threads once that timer expires. Child threads keep track of the number of operations they perform, and report this value to the parent. The parent then calculates the average execution time per operation by dividing the total number of operations performed by all children by the duration of the test. Our tests performed seven trials, and reported the average of the median five.

Each thread runs repeatedly through a test loop. Once the loop completes, a quiescent state is identified if we are using QSBR, and the loop is begun again if the parent thread’s timer has not yet expired, as shown in Figure 5.1. For hash tables, on each iteration of the loop, the thread does either a search, an insertion, or a deletion. The probabilities of performing an insertion or a deletion are always equal, to keep the average load factor constant throughout the trial. For queues, the thread does either an enqueue or a dequeue

```

while (parent's timer has not expired) {
    for i from 1 to 100 do {
        key = random key;
        op = random operation;
        if (testing queues) {
            q = random queue;
            op(q, key);
        } else { /* Testing a hash table */
            op(key);
        }
    }
    QUIESCENT_STATE();
}

```

Figure 5.1: High-level pseudocode for the test loop of our program. Each thread executes this loop. The call to `QUIESCENT_STATE()` is ignored unless we are using QSBR.

on each operation, again with equal probability.

The tests allow us to vary the number of queues or hash buckets, the number of threads, and the total number of nodes we begin with. In the case of hash tables, we are also able to vary the load factor and the update fraction.

As shown in Figure 5.1, each thread performs 100 operations per quiescent state; hence, the overhead of announcing a quiescent state is amortized over 100 operations. For EBR, each *op* in Figure 5.1 is a critical section; a thread attempts to update the global epoch whenever it has entered a critical section 100 times without seeing the global epoch updated. For SMR, we chose $R = 2H + 100$.

5.1.3 Operating Environment

We performed our experiments on two machines: one with two 2.0 GHz PowerPC G5 processors, and another with two 2.8 GHz Intel Xeon processors with no hyperthreading. The PowerPC machine ran Mac OS X Server version 10.3.3 with Darwin kernel version 7.4.0, while the Xeon machine ran Red Hat Enterprise Linux version 3 with Linux kernel

Table 5.1: Characteristics of Machines

	Machine 1	Machine 2
Processor	PowerPC	IA-32
# CPUs	2	2
GHz	2.0	2.8
Kernel	Darwin	Linux
Memory Model	weakly-consistent	weakly-consistent, writes ordered
Full Fence	86 ns	73.4 ns
Write Fence	13 ns	0 ns
CAS	33.9 ns	78.6 ns
Lock + Unlock	166 ns	141 ns

version 2.4.21-20.ELsmp. The Xeon machine has a slightly stronger memory model in which writes are always performed in program order; write fences therefore do not add to the costs of algorithms on this machine. Table 5.1 summarizes the properties of these machines; the last line refers to the combined cost of locking and unlocking a spinlock. The difference in CAS costs on the two architectures is partially due to the fact that on IA-32, a CAS implies a full fence, while on PowerPC it does not. For consistency, we report the results from experiments performed on the PowerPC machine unless otherwise noted — in almost all cases, the choice of architecture made no significant performance difference.

Threads in our experiment are Unix processes. Our memory allocator provides per-thread freelists. Each thread can have two freelists of 100 nodes each at any given time. Threads which exhaust their freelists can acquire more memory from a spinlock-protected global pool. This design is similar to that of the slab allocator with magazines [9]. All nodes are pre-allocated by the parent before the test begins.

5.1.4 Limitations of Experiment

Although we believe that these experiments provide significant insight into the behavior of different memory reclamation schemes, we do not know how accurately our microbenchmark reflects real applications. Some applications may not have natural quiescent states. Furthermore, detecting quiescent states in other applications may be more expensive than it is in our program. Our QSBR implementation, for example, has less overhead than that used in the Linux kernel, which must deal with issues such as CPU hotplugging and the need to support interrupt handlers and real-time workloads.

Our experiment is also limited by the fact that the threads do nothing but invoke operations on a small number of objects repeatedly, and that our memory allocator uses locking. Performing a macrobenchmark on an existing application and testing the performance of these schemes under a lock-free memory allocator are two avenues for future work.

Each iteration of our test loop calls the `random()` function at least once. This adds a constant amount of overhead to our measurements. Since we are primarily interested in overall trends, this overhead is not a major concern in our analysis.

Finally, we were limited by the hardware we had available. Although testing on commodity dual-CPU hardware evaluates these memory reclamation strategies under realistic conditions, we were unable to evaluate how these schemes scale to large numbers of CPUs or on other platforms such as NUMA machines or clusters.

Despite these limitations, we feel that our analysis highlights trends that show when each memory reclamation scheme is and is not efficient.

5.2 Performance Analysis

Our experiments show how varying object contention, CPU contention, workload, and traversal length significantly affect the performance of each memory reclamation scheme.

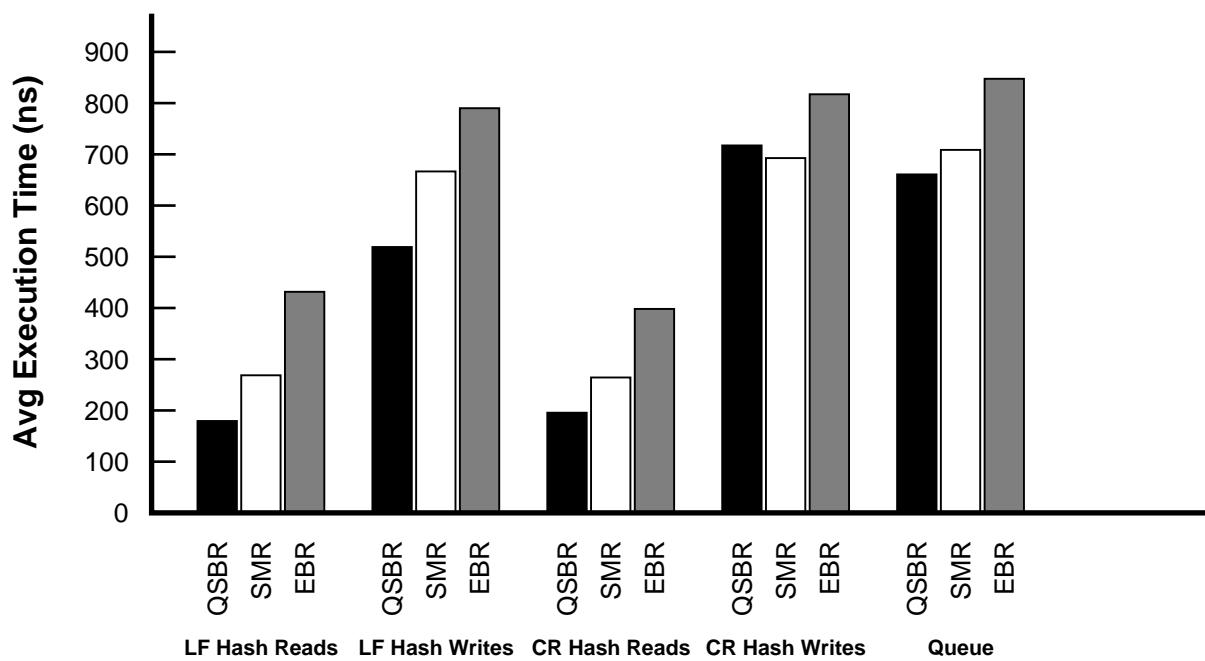


Figure 5.2: Single-threaded memory reclamation costs on PowerPC. Hash table statistics are for a 32-bucket hash table with a load factor of 1. Queue statistics are for a single non-empty queue.

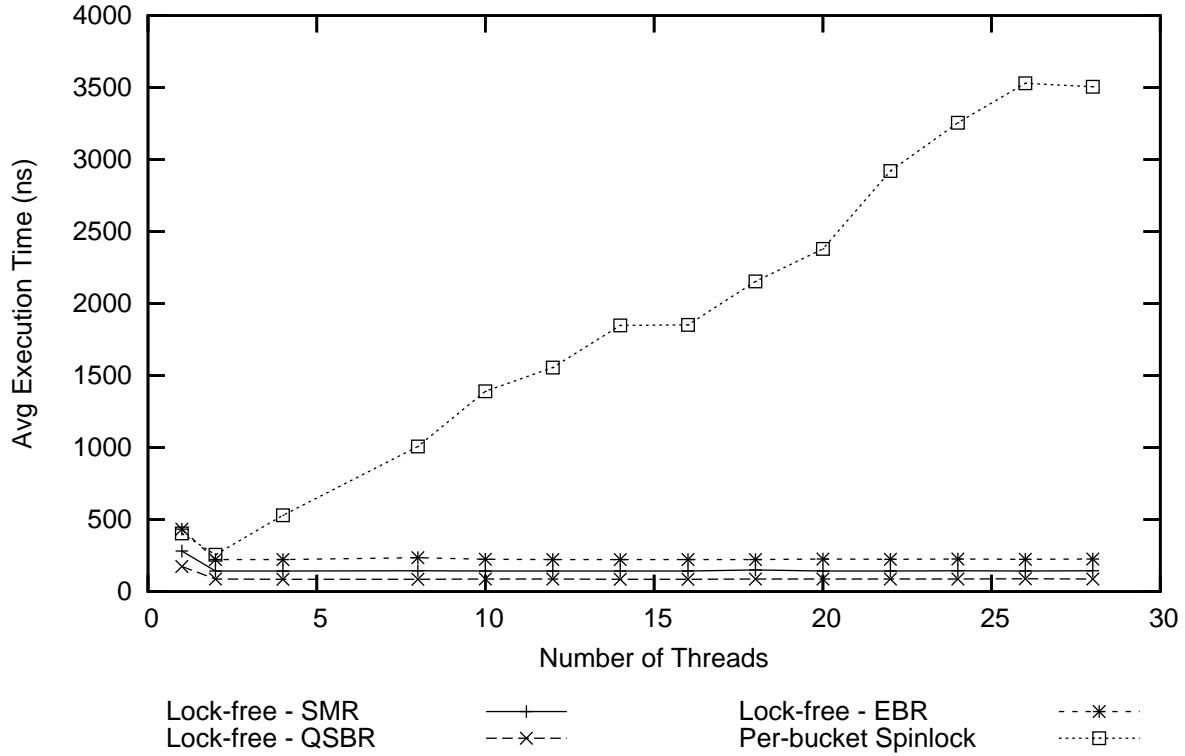


Figure 5.3: Hash table, 32 buckets, load factor 1, read-only workload. Spinlocks scale poorly as the number of threads increases.

Figure 5.2 shows the base costs of the concurrently-readable and lock-free algorithms under consideration when all these costs are low. The results are those for one thread, so that there is no contention of either type. The load factor of the hash tables is one. As predicted in Section 4.3, the fence instructions required make SMR and EBR more expensive than QSBR. The one exception is the write-only workload for the concurrently-readable hash table with SMR; this is because, in the write-only case, no hazard pointers need be used, so the overhead of using SMR is negligible.

We note that, in the base case, which memory reclamation scheme is most efficient seems to be determined solely by the per-operation fence instructions required.

We next show what is already well-established: contention for locks and the CPU seriously degrades the performance of lock-based algorithms. Figure 5.3 shows the per-

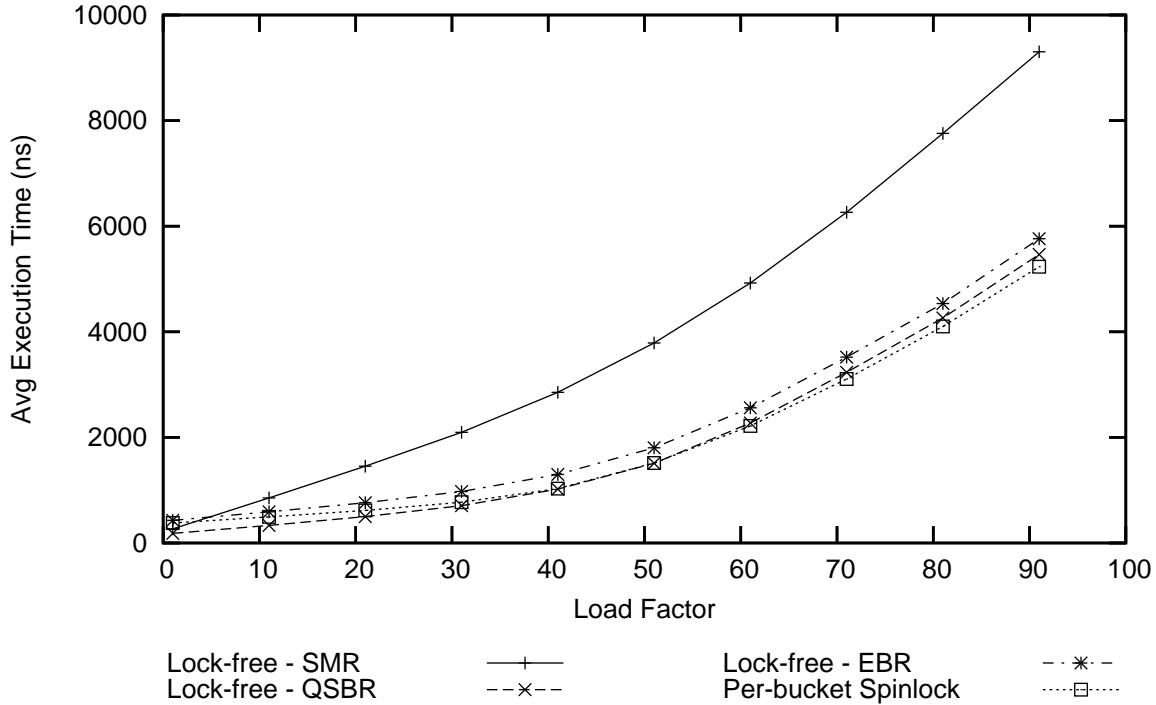


Figure 5.4: Hash table, 32 buckets, one thread, read-only workload, varying load factor.

formance degradation of spinlocks on the hash table as the number of concurrent threads increases, using a read-only workload and a load factor of 1. Since our tests were performed on a two-CPU machine, using more than two threads cannot increase our aggregate throughput; hence, horizontal lines on the graph indicate perfect scalability. Increasing the number of threads increases both the object contention, which makes threads more likely to spin while attempting to acquire a lock, and the CPU contention, which increases lock holder times. All memory reclamation schemes for the lock-free hash table scale equally well with these settings.

5.2.1 Effects of Traversal Length

In Figures 5.4 and 5.5, we show the effect of increasing the load factor of the hash table, when a single thread executes either a read-only or write-only workload, respectively.

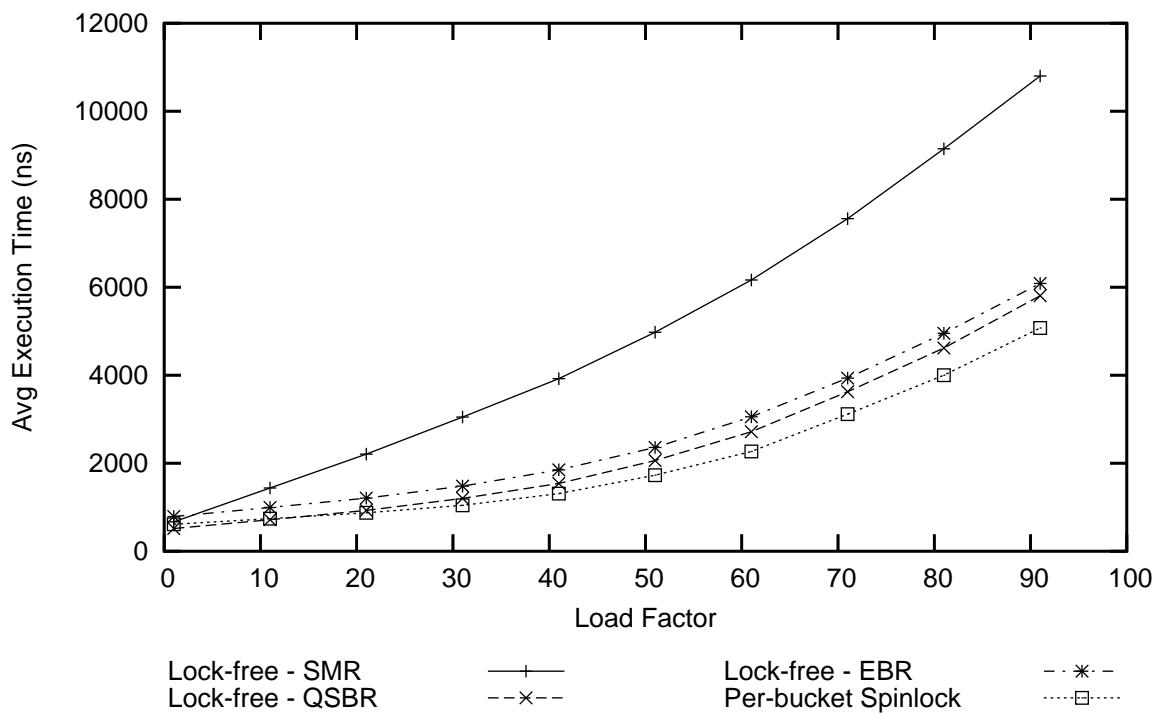


Figure 5.5: Hash table, 32 buckets, one thread, write-only workload, varying load factor.

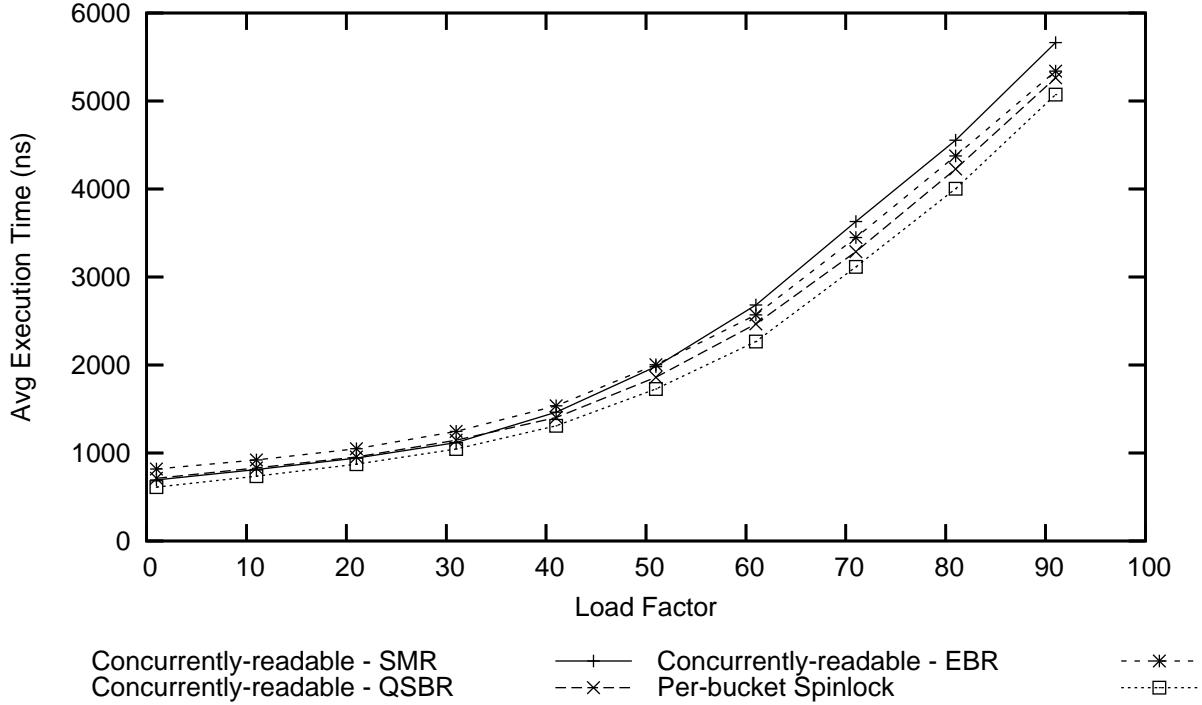


Figure 5.6: Hash table, 32 buckets, one thread, write-only workload, varying load factor.

Increasing the load factor effectively increases the traversal length for all operations, as even insertions first search for the key to be inserted in order to prevent duplicate entries. As predicted in Section 4.3, the per-node fence instructions ruin the performance of the lock-free hash table when using SMR: both the lock-based hash table and the lock-free hash table using QSBR or EBR severely out-perform it. Using SMR for memory reclamation for any structure requiring traversals of long chains of nodes, such as dynamic trees, is thus likely to be extremely expensive.

In read-mostly situations, the effect of increasing the load factor on the performance of the concurrently-readable algorithm using SMR is similar to its effect in the lock-free case. There is a difference, however, in write-mostly situations, as shown in Figure 5.6. The cause is the same as that of the anomaly in Figure 5.2: the concurrently-readable algorithm holds a lock for updates; hence, traversals done by updates do not suffer the

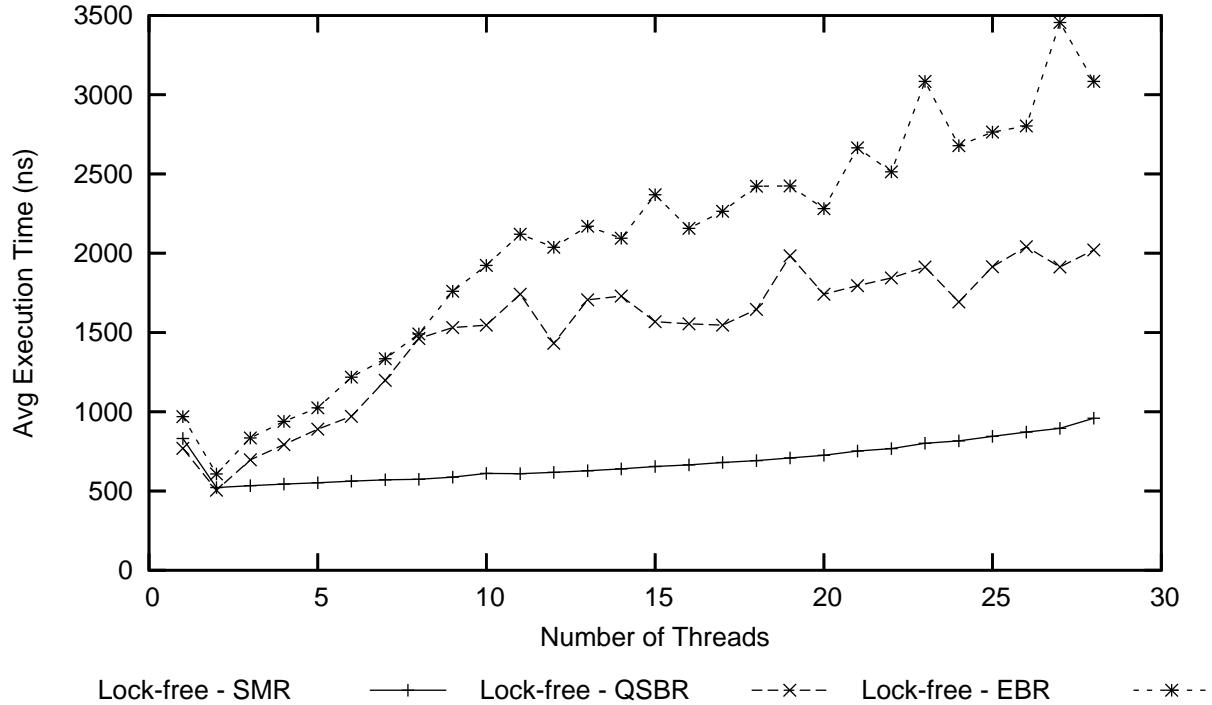


Figure 5.7: 100 queues, variable number of threads, Darwin/PPC.

per-node overhead of fence instructions, so the concurrently-readable algorithm’s updates perform similarly regardless of reclamation method. This case is somewhat degenerate, however, since with a write-only workload, the concurrently-readable algorithm is equivalent to a naive per-bucket spinlock approach.

5.2.2 Effects of CPU Contention

QSBR and EBR scale well with load factor, and, as shown in Figure 5.3, they scale well with CPU contention under read-only workloads. However, Figure 5.7 demonstrates that they scale poorly with CPU contention when the workload involves a significant number of operations that must allocate memory. For this experiment, we focus on queues where every operation either allocates or deallocates memory. Although QSBR and EBR are hurt by CPU contention and allocation-heavy workloads on both systems, the magnitude of the effect depends on the execution environment. A similar, though less pronounced,

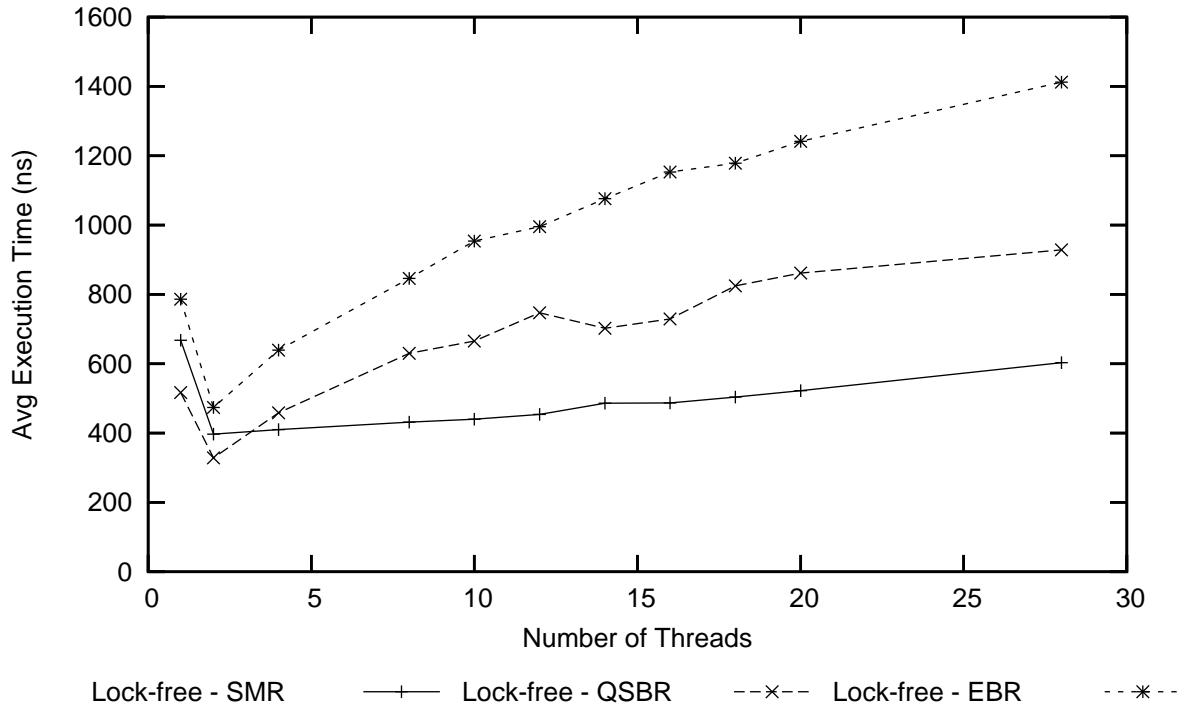


Figure 5.8: Hash table, 32 buckets, load factor 1, write-only workload, variable number of threads, Darwin/PPC.

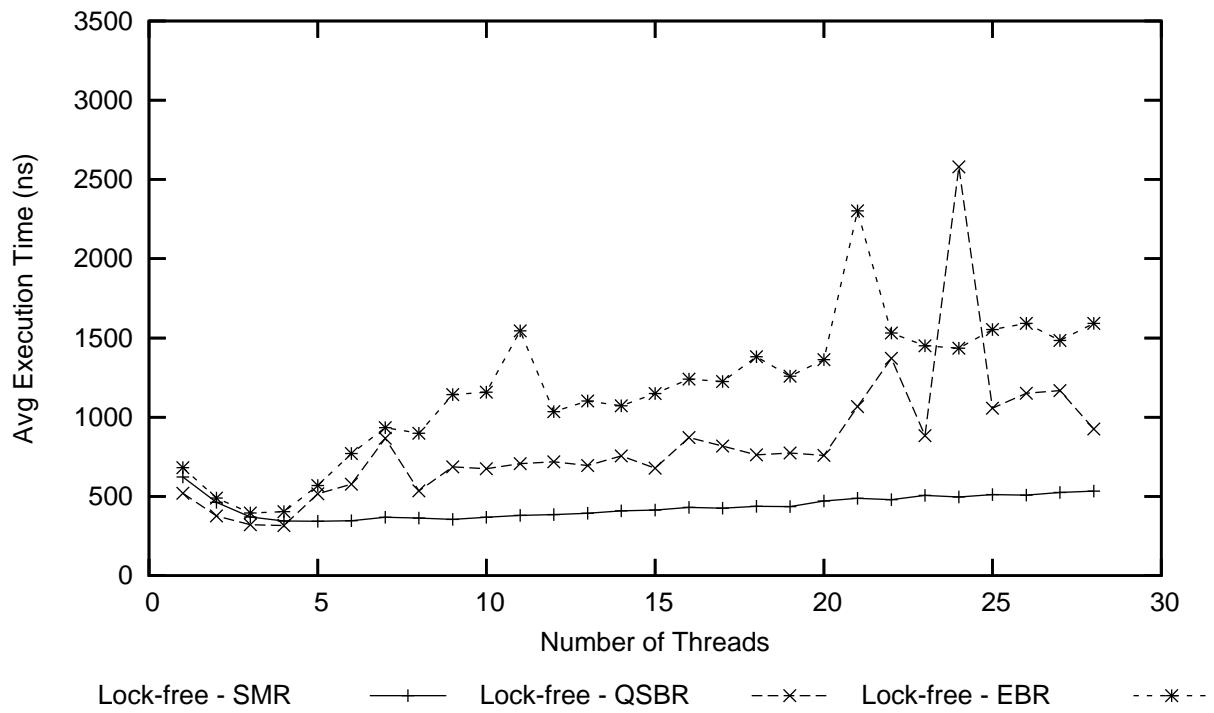


Figure 5.9: 100 queues, variable number of threads, Linux/IA-32.

effect can be observed for hash tables, as illustrated in Figure 5.8.

We predicted this scalability problem in Section 4.3. The extent to which this overhead seems to be scheduler-dependent and unpredictable is surprising, however. Figure 5.9 shows data from the same experiment as shown in Figure 5.7, but running on our Linux/IA-32 machine. Here, the shapes of the curves are quite different. Since this experiment involves significant multithreading, we believe that the differences are due to the different schedulers in the two kernels.

Although the increased contention for locks in our memory allocator and exhaustion of the memory pool play a part in QSBR and EBR’s poor performance, we found while trying to mitigate these factors that we could not make these methods perform well when both CPU contention and memory allocation rates are high. Among other factors, we have found evidence that QSBR and EBR’s inefficient use of memory interacts poorly with the OS’s memory management strategies on the Darwin/PowerPC system. We are presently unable to analyze completely all the ways that QSBR and EBR interact with the external execution environment; however, it is clear from our results that delaying threads can have an profound impact on the performance of these two schemes since threads are prevented from physically deleting nodes. Further, as external factors come into play, it is extremely difficult for an application programmer to defend against these costs. SMR, in contrast, is largely immune to contention and bounds the memory usage, shielding the programmer from external concerns.

We note that, in our experiments, QSBR scales better than EBR with increasing numbers of threads. We view this as an artifact of our implementations. Our QSBR mechanism was designed specifically to minimize the per-grace period overhead of each thread, while Fraser’s EBR scheme, which we adopted, was not.

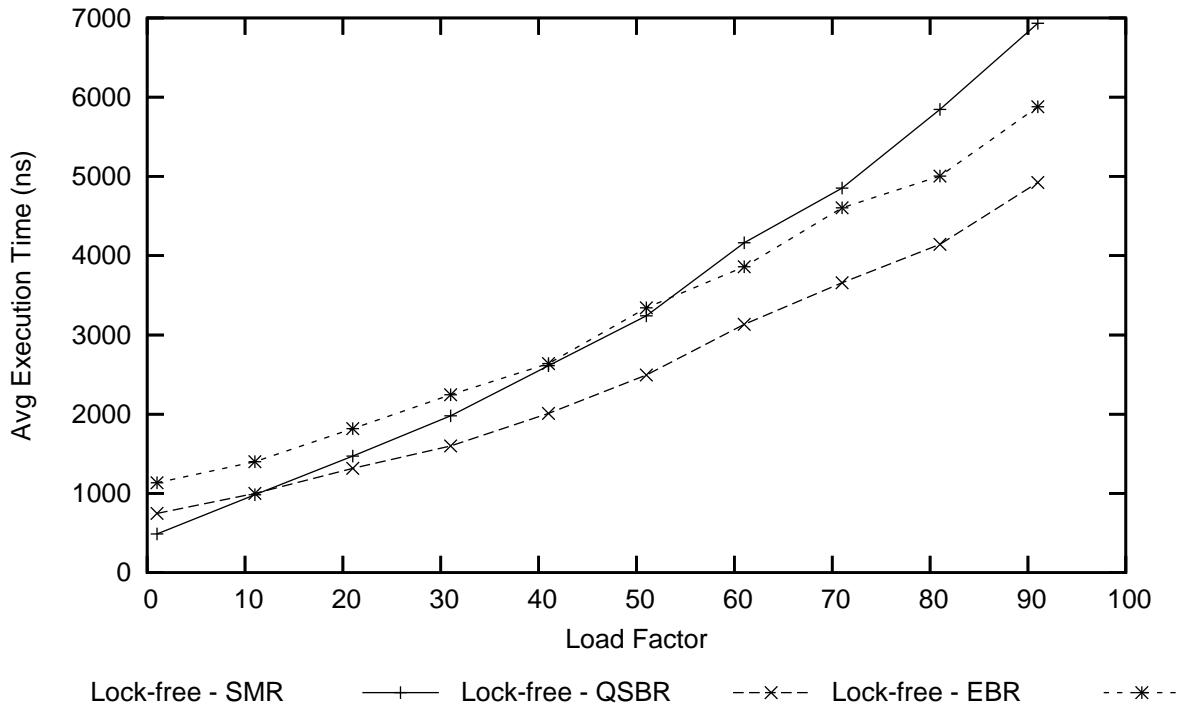


Figure 5.10: Hash table, 32 buckets, 16 threads, write-only workload, varying load factor.

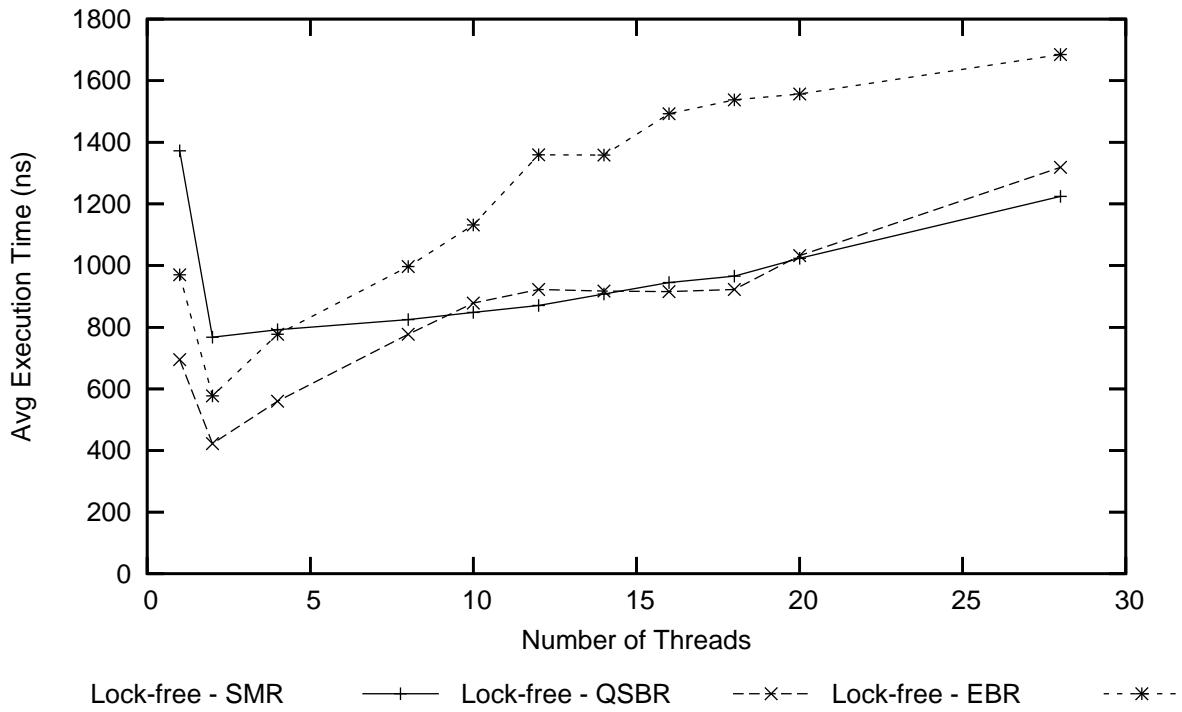


Figure 5.11: Hash table, 32 buckets, load factor 10, write-only workload, varying number of threads.

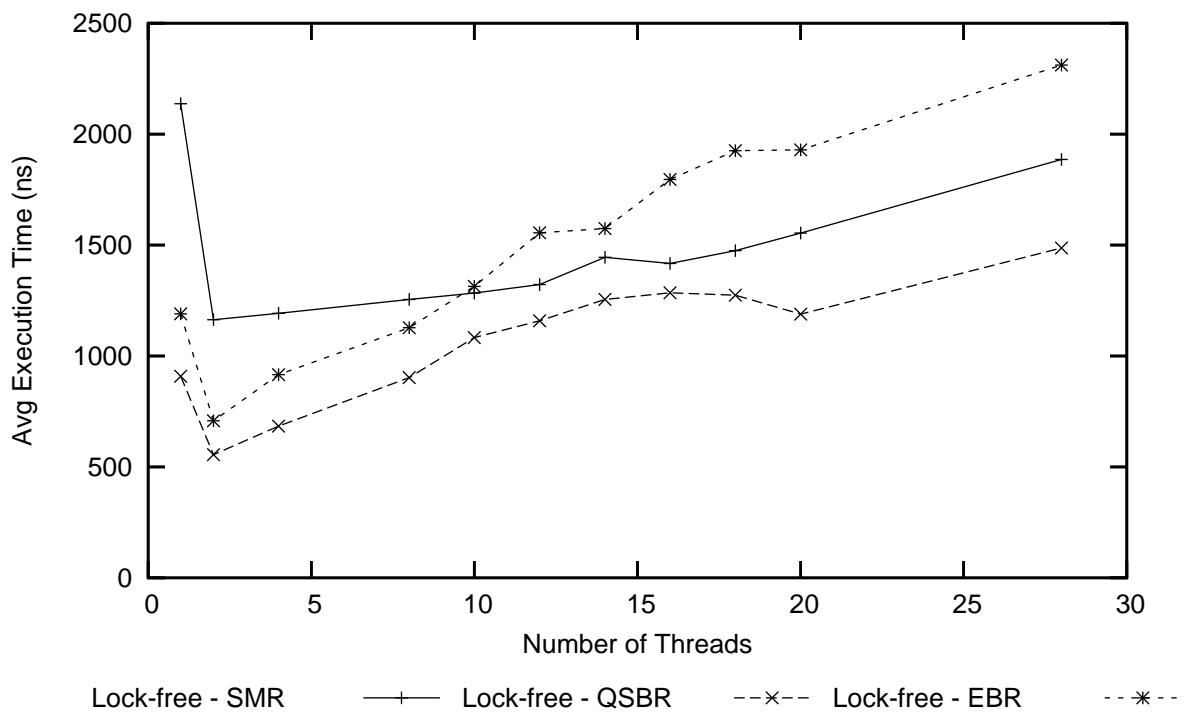


Figure 5.12: Hash table, 32 buckets, load factor 20, write-only workload, varying number of threads.

5.2.3 Relative Severity

We have seen that SMR’s performance scales poorly with traversal length, while EBR and QSBR scale poorly for update-intensive workloads in the presence of CPU contention. In cases in which traversal length is high, the workload is update-heavy, and there is CPU contention, a programmer may wish to know which factor will influence the performance of memory reclamation schemes most severely.

Figures 5.10, 5.11, and 5.12 address this question. All three graphs show runs with a high load factor, CPU contention, and a write-only workload.

Figure 5.10 shows the effect of increasing load factor when we have 16 threads — eight threads per CPU. Even at this high level of CPU contention, QSBR begins to outperform SMR when the load factor exceeds 10, and EBR starts to beat SMR when the load factor exceeds 50.

Figure 5.11 shows the effect of increasing the number of threads when the load factor is 10. SMR begins to outperform EBR when there are more than four threads. SMR and QSBR are competitive when there are between 10 and 20 threads, and SMR begins to become superior when there are more than 20 threads. Figure 5.12 shows a similar plot, but with a load factor of 20. Here, SMR only begins to outperform EBR when we have more than 10 threads, and it does not outperform QSBR for any number of threads we tested.

Judging from Figures 5.10, 5.11, and 5.12, it appears that we need only a moderate load factor in order to make SMR perform poorly, but a very large amount of CPU contention with an update-intensive workload in order to make QSBR and EBR perform poorly. However, we urge caution in interpreting these results. First, the performance difference between our EBR implementation and our QSBR implementation show that the relative performance of memory reclamation schemes is very application-dependent. Second, in the case of QSBR, our experiments all had 100 operations per quiescent state; in real code, there may be many more operations per quiescent state, and therefore more

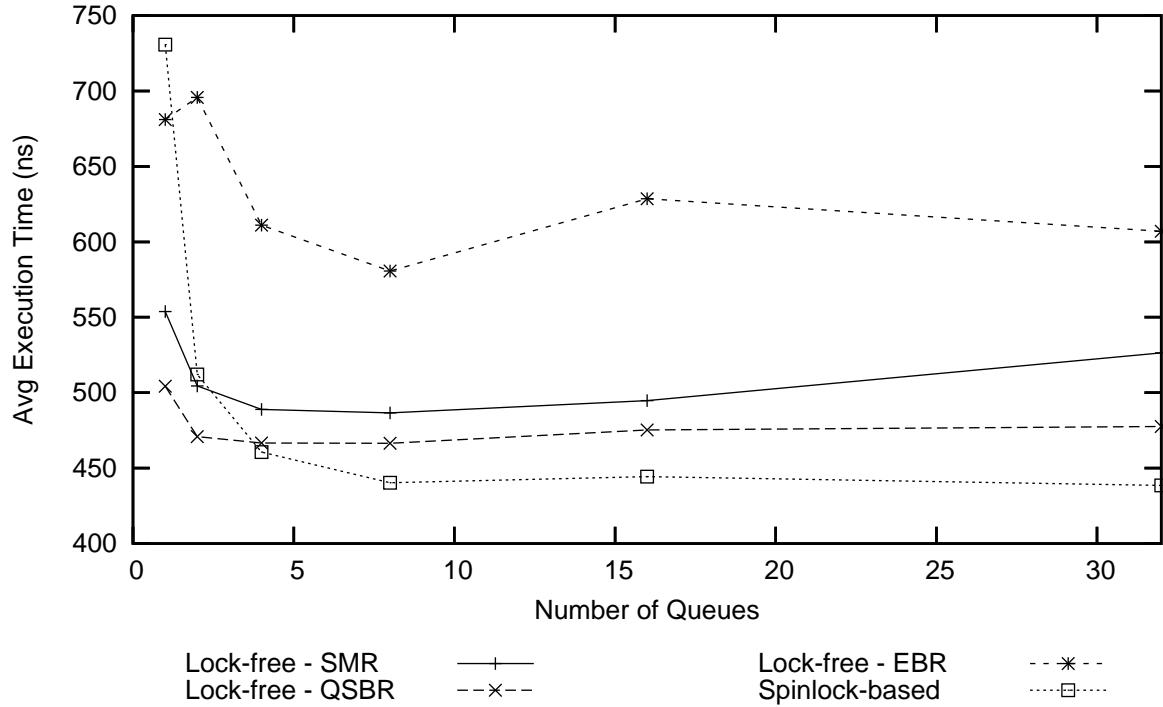


Figure 5.13: Queues, two threads, varying number of queues.

callbacks per grace period. Nevertheless, we hope that these results can give programmers some intuition as to the relative severity of factors affecting the performance of memory reclamation.

5.2.4 Low Overhead of QSBR

Based on our results so far, we find that QSBR is consistently the best-performing memory reclamation scheme, except when CPU contention combines with allocation-heavy workloads. This is further demonstrated in Figures 5.13 and 5.14, in which QSBR outperforms the other two memory reclamation strategies in all cases, often by a considerable margin. The left graph shows the effect of varying the number of queues, while the right graph shows the effect of varying the update fraction on the hash table.

The difference in performance is most pronounced when we consider hash tables. With two CPUs, one thread per CPU, and a load factor of five, the combination of the

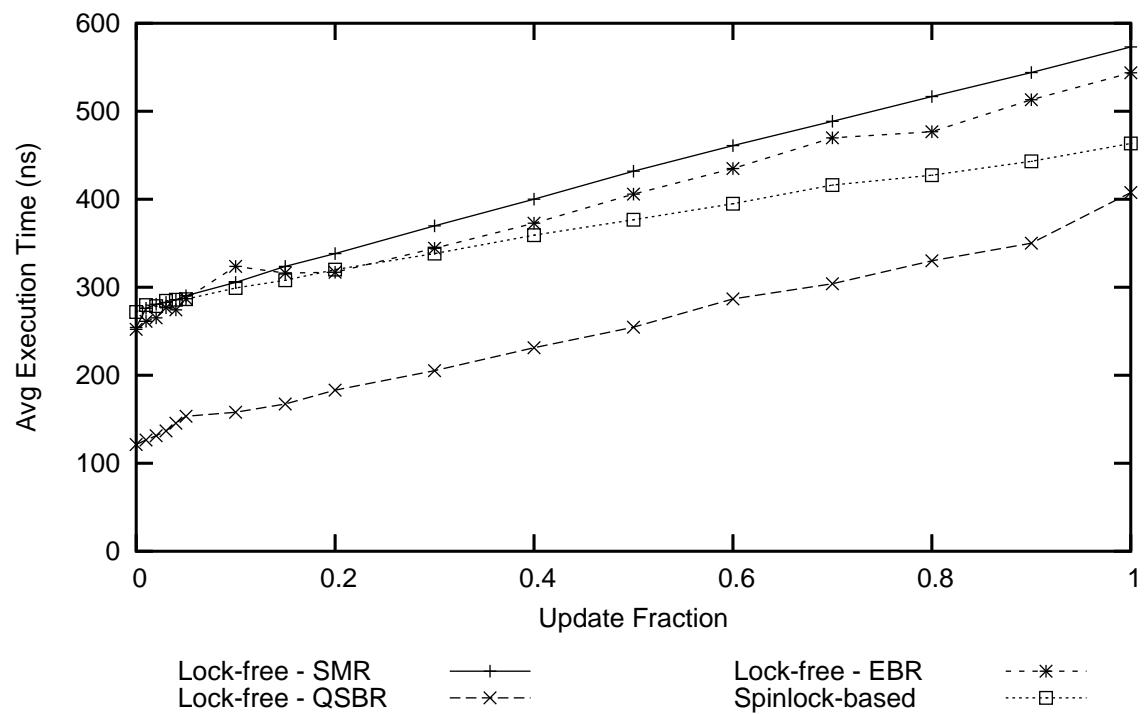


Figure 5.14: Hash table, 32 buckets, two threads, load factor 5, varying update fraction.

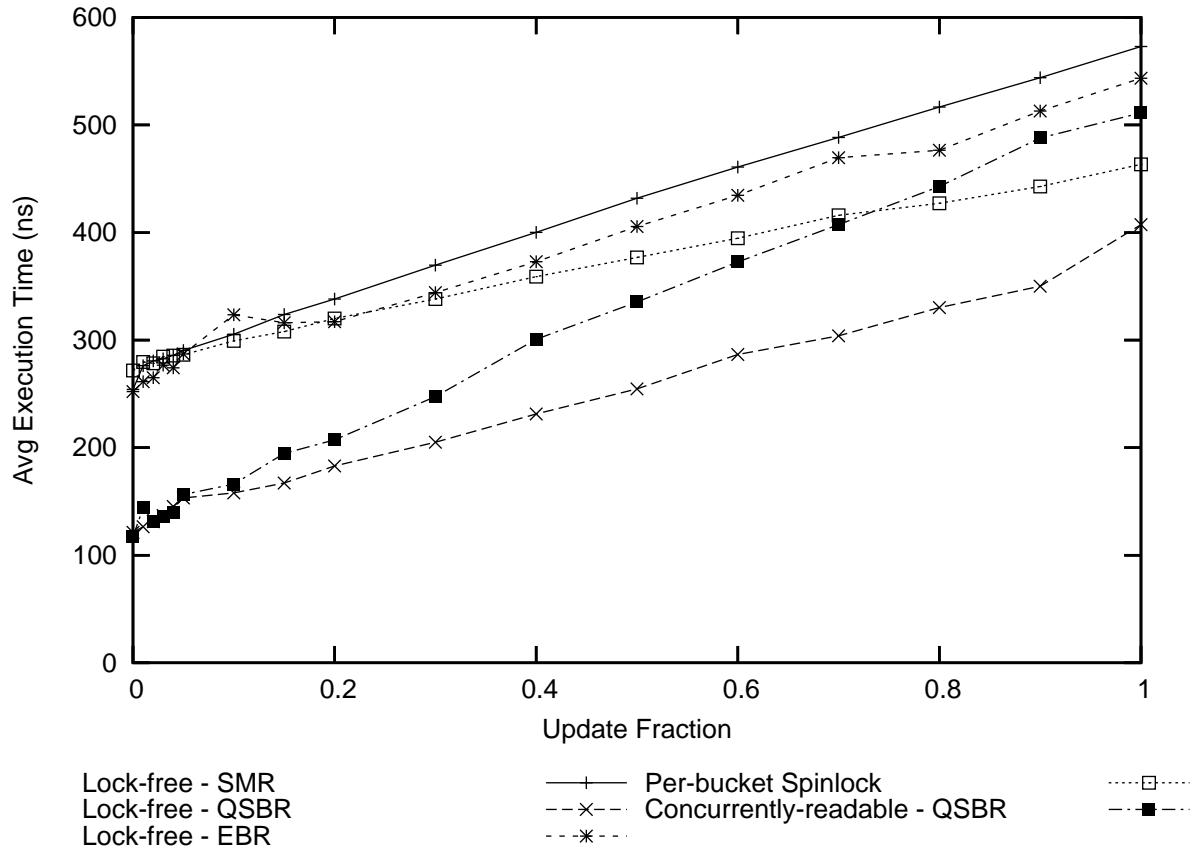


Figure 5.15: Hash table, 32 buckets, two threads, load factor 5, varying update fraction. QSBR allows the lock-free algorithm to out-perform RCU for almost any workload; neither SMR nor EBR achieve this.

lock-free algorithm with QSBR out-performs the lock-based alternative for any update fraction, while the lock-free algorithm with SMR or EBR fails to out-perform the lock-based version for almost all update fractions. Here, the choice of memory reclamation scheme clearly determines whether or not a lock-free algorithm can out-perform a lock-based one.

5.2.5 Lock-free Versus Concurrently-readable Linked List Algorithms

Using QSBR for both the concurrently-readable and lock-free linked list algorithms allows us to fairly compare their performance, and to see where each may be appropriate.

```

struct list_head *cur;

#define clean_pointer(p) ((unsigned long)((p)) & (-2))

int search (struct list_head **head, long key)
{
    cur = *head;
    while (cur != NULL) {
        long ckey = (list_entry(cur, struct el, list))->key;
        if (ckey >= key) {
            return (ckey == key);
        }
        cur = clean_pointer(cur->next);
    }
    return (0);
}

```

Figure 5.16: Code for fast searches of lock-free list; compare to pseudocode of Figure 3.4.

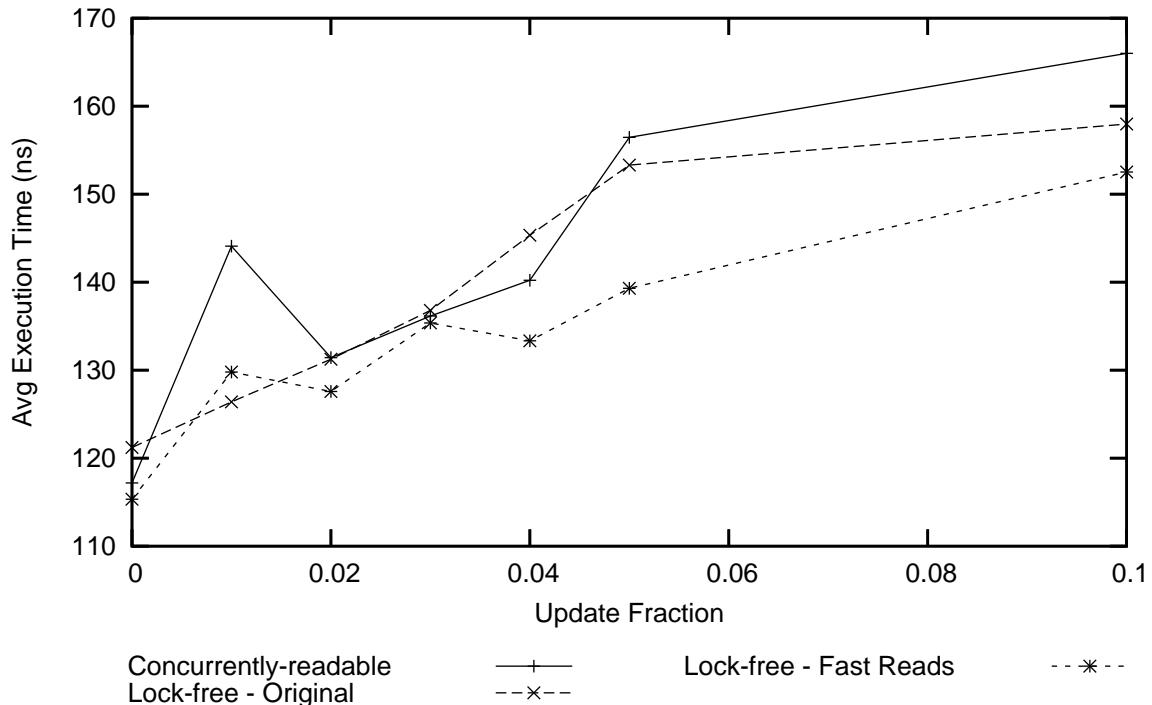


Figure 5.17: Hash table, 32 buckets, two threads, load factor 5, varying update fraction between 0 and 0.1.

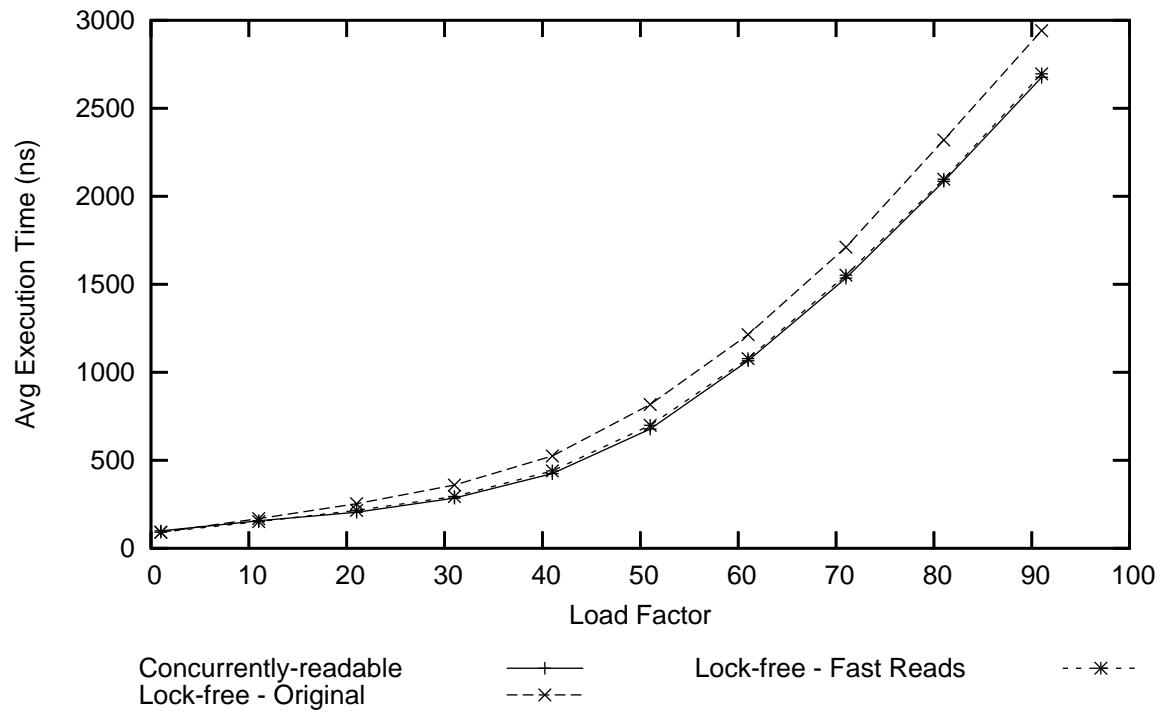


Figure 5.18: Hash table, 32 buckets, two threads, read-only workload, varying load factor.

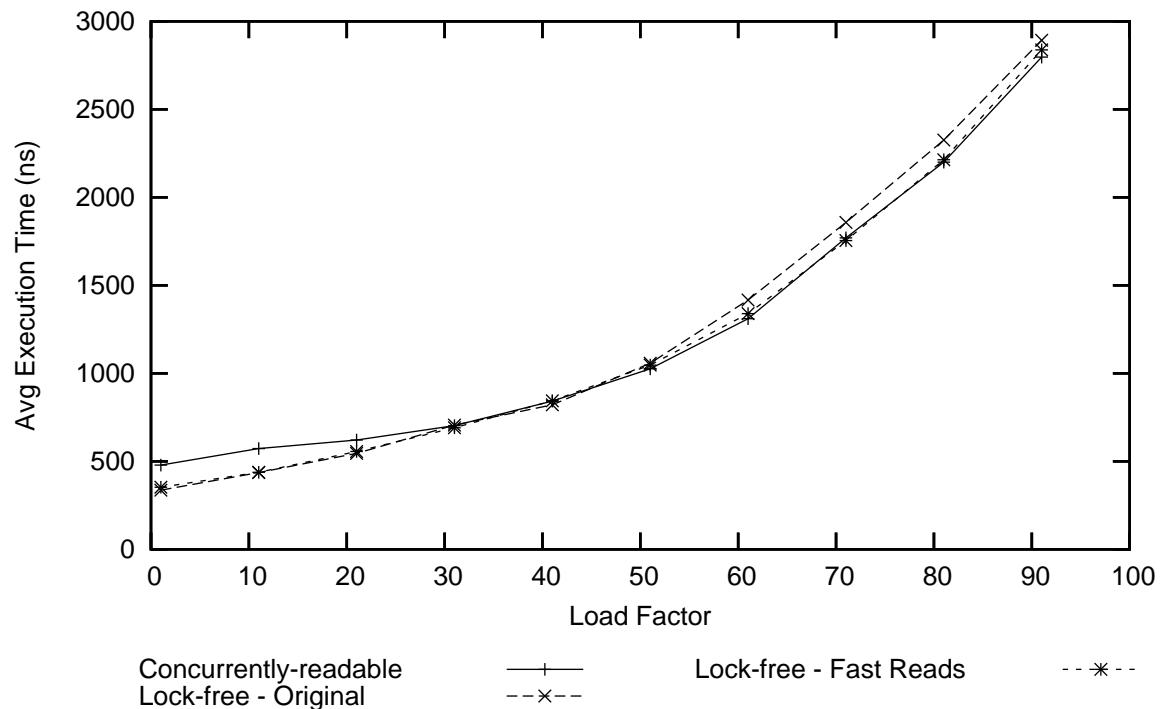


Figure 5.19: Hash table, 32 buckets, two threads, write-only workload, varying load factor.

The combination of the concurrently-readable linked list algorithm with QSBR is called read-copy update (RCU), which was mentioned in section 3.3 and is currently used in several OS kernels including Linux. RCU has extremely high performance for read-mostly workloads. We wanted to determine if, given efficient memory reclamation, our lock-free hash table could be suitable for use in OS kernels.

Figure 5.15 shows that the combination of the lock-free hash table with QSBR outperforms the concurrently-readable version for almost all update fractions. When the workload is nearly read-only, the concurrently-readable version performs slightly better than the lock-free one, since its concurrently-readable searches do not have the extra checks required of the lock-free searches. The lock-free algorithm has considerably cheaper updates, however, which allow it to scale much better as the update fraction increases. The lock-free algorithm’s ability to perform competitively against the concurrently-readable one depends on our use of QSBR. If we use either SMR or EBR, the per-operation overhead of memory reclamation makes the lock-free algorithm inefficient.

Figure 5.17 zooms in on the range of update fractions between 0 and 0.1 of Figure 5.17. Here, we consider only the algorithms using QSBR; however, we add a new version of the lock-free algorithm which weakens the semantics of its reads (see Figure 5.16) to make it more competitive with the concurrently-readable algorithm. The design of these reads takes to heart the tenet of the RCU paradigm which seeks to minimize read-side synchronization [38]. These fast reads simply ignore nodes marked for deletion instead of helping to unlink them; hence, these reads may return nodes which have already been marked for deletion. These semantics are only slightly weaker than the original one, since in a concurrent programming environment, a read could find an undeleted node, and another thread could mark that node for deletion as it is returned to the application program.

These reads are used when the application program searches the hash table. The

deletion method uses this fast search on the first attempt to find the node to delete; if the deletion method must retry, it uses the original search code in order to ensure forward progress.

These faster reads have little impact on overall performance in Figure 5.17 — the performance of all three algorithms is very similar at low update fractions. This is because the load factor is low, so the per-node overhead of lock-free searches is very small, and is in the noise for the experiment.

The fast reads are more important at higher load factors. Figures 5.18 and 5.19 show the performance of these three algorithms on a read-only and write-only workload, respectively, as the load factor increases. The per-node overhead of the extra checks becomes significant for the original version of the lock-free algorithm. The modified version, however, remains competitive with the concurrently-readable version, even at very high load factors.

The result that, when using QSBR, the lock-free algorithm can outperform the concurrently-readable one when the load factor is low and the update fraction is above 0.1 may have practical implications. The lock-free algorithm also outperforms per-bucket spinlocking for any update fraction, while the concurrently-readable algorithm does not. This lock-free algorithm may find a niche in kernels which use RCU, for use with update-heavy structures. Investigating this further is a topic for future work.

5.2.6 Summary of Recommendations

Given the results examined in the previous sections, we are able to provide some rules of thumb for choosing between competing algorithms and memory reclamation strategies.

We have seen that SMR performs poorly when long chains of nodes must be traversed, and that EBR and QSBR perform poorly when there is CPU contention and an allocation-heavy workload. Of the two factors, traversal length seems to have the greater effect. QSBR has the lowest base overhead, while EBR has the most. EBR's only advan-

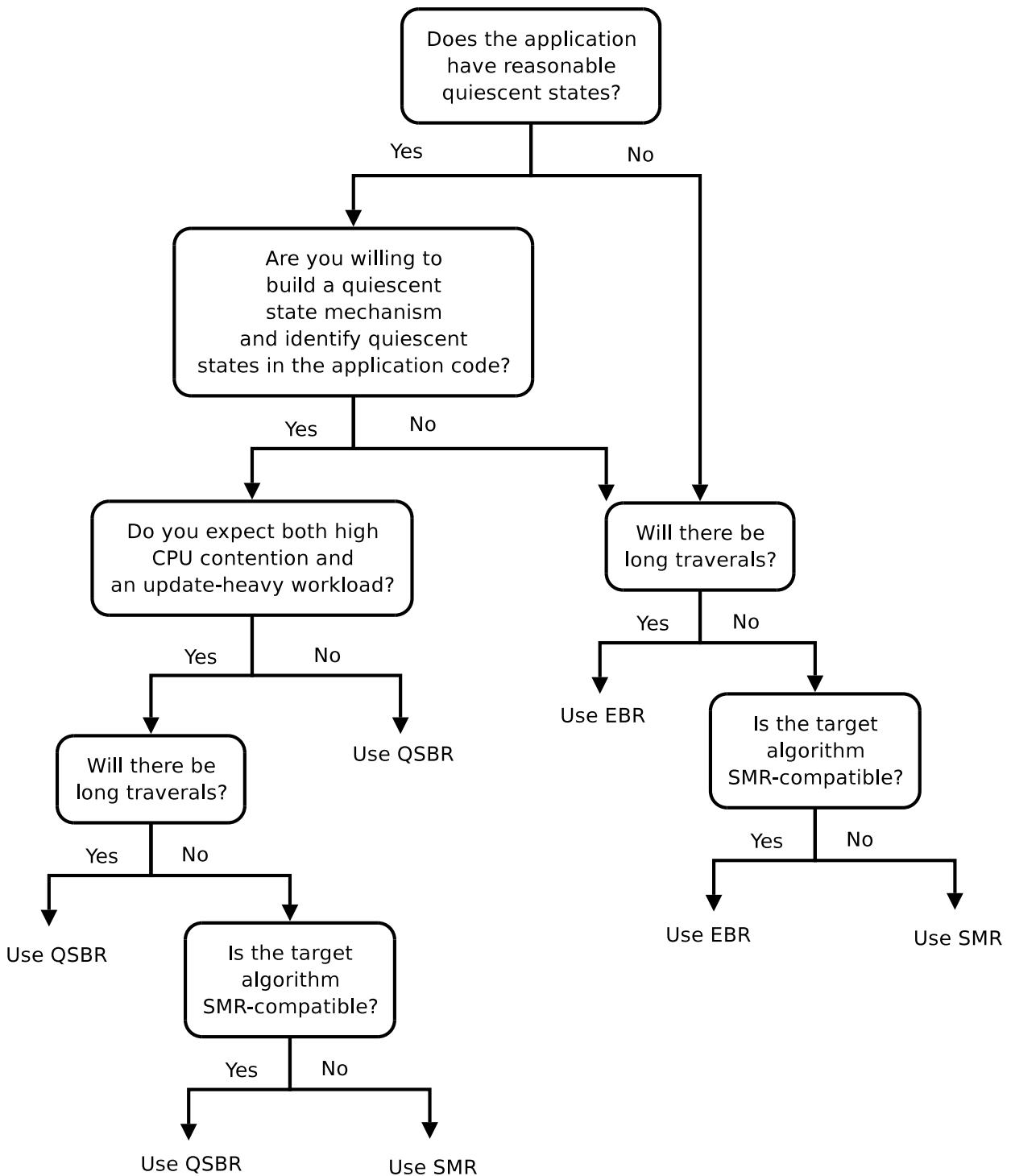


Figure 5.20: Decision tree for choosing a memory reclamation scheme.

tage over QSBR is that it is application-independent; in some cases, this advantage may make EBR preferable. Figure 5.20 presents a decision tree to help programmers choose a memory reclamation scheme.

Choosing between the lock-free and concurrently-readable algorithms is simpler. If there is CPU contention, we can expect the lock-free algorithm to perform much better. Otherwise, the lock-free algorithm will perform better if the update fraction is significant, and the traversal length is not so long that the per-node traversal overhead becomes significant. However, if operations other than insertions, deletions, and searches are required, it is likely to be easier to add them to the concurrently-readable list than the lock-free one. Figure 5.21 shows a decision tree for choosing between the two algorithms.

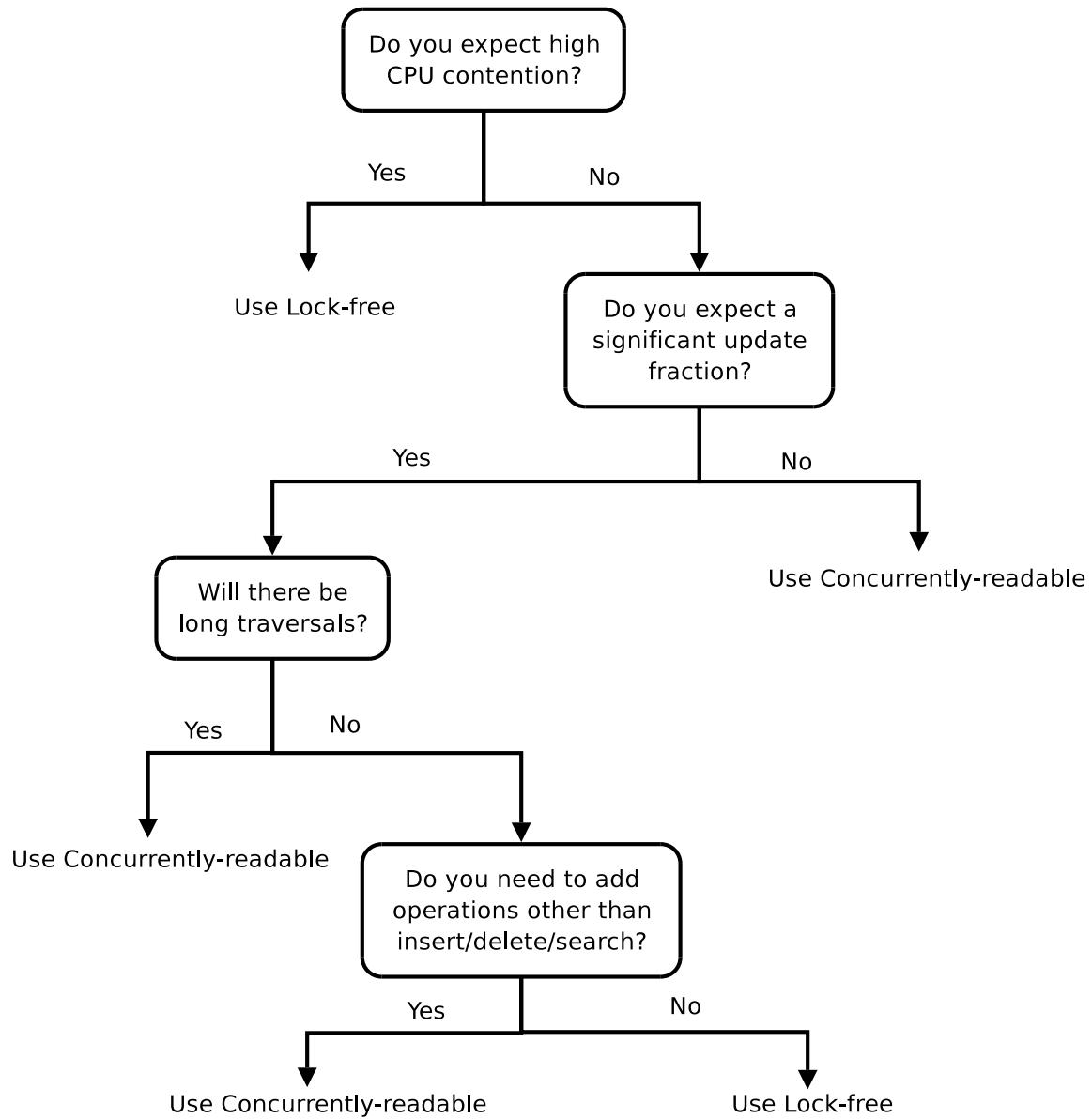


Figure 5.21: Decision tree for choosing whether to use the lock-free or concurrently-readable linked list algorithm.

Chapter 6

Related Work

Our contribution lies not in introducing new memory reclamation strategies, but in providing a comprehensive analysis of their relationship and their relative merits, and in using QSBR with lock-free queues. This work is, to our knowledge, the first to pair QSBR with an object that does not have read-only operations.

6.1 Blocking Memory Reclamation for Non-blocking Algorithms

Others have proposed that memory reclamation strategies that are not lock-free should sometimes be used in combination with lock-free algorithms. Fraser [17] noted, but did not thoroughly evaluate, the performance overhead of SMR due to the fence instructions it requires, and used EBR instead. Our work continues Fraser’s in showing that EBR itself has high overhead — often higher than that of SMR — and that more efficient memory reclamation is often possible. We view our work and Fraser’s as being part of a trend towards weakenings of lock-freedom, such as obstruction-freedom [29] and almost non-blocking data structures [8], designed to preserve the advantageous properties of lock-freedom while improving performance.

We are not the first to use QSBR with lock-free algorithms — Auslander implemented a lock-free hash table with QSBR [38] in the K42 operating system [5, 18]. However, pairing QSBR with hash tables does not fully separate it from the RCU paradigm, since hash tables have read-only operations. Thus, it does not evaluate QSBR for use with more general lock-free objects as we have done by pairing it with a lock-free queue. Furthermore, no performance evaluation, either between different memory reclamation methods or between concurrently-readable and lock-free hash tables, was provided in [38].

In response to Auslander’s work, McKenney [38] posed several questions concerning the use of RCU with lock-free synchronization; among them were:

- Whether using QSBR will make non-blocking synchronization more broadly applicable.
- With which non-blocking algorithms it makes sense to use QSBR.
- What merits using QSBR with non-blocking synchronization has, relative to other techniques.
- How various combinations of QSBR and non-blocking synchronization compare to one another, both empirically and analytically.

Although it is difficult for experimental research to answer all aspects of McKenney’s questions, our work addresses them significantly. First, our results show that while in many situations using QSBR can improve the performance of a lock-free algorithm, it is not a silver bullet. QSBR therefore makes non-blocking synchronization more feasible in many environments, but locking may still be preferable in many situations (see Figure 5.13).

Second, our good performance results from pairing QSBR with queues indicate that it makes sense to use QSBR with many lock-free algorithms. In particular, pairing QSBR

with queues shows that using QSBR makes sense even when the target data structure has no read-only operations.

Third, the results of our performance analysis show quite clearly the advantages and disadvantages of using QSBR relative to other memory reclamation schemes. QSBR has the lowest base time overhead of any memory reclamation scheme we are aware of, and its overhead does not grow when the traversal length increases. SMR, by contrast, has significant per-node overhead, so its cost grows linearly as the traversal length increases. The only disadvantage of QSBR with regard to performance is that it performs poorly for update-intensive workloads when CPU contention is high. In all other situations, QSBR is the clear winner in terms of performance.

Last, our results also provide a comprehensive comparison of two combinations of QSBR and non-blocking synchronization. Our analysis and experiments both show that, in the base case, the performance differences between different reclamation schemes come from the per-operation overhead due to expensive operations such as fences, and not due to the complexity of periodic reclamation routines. Also, as noted above, Auslander’s implementation of a lock-free hash table using QSBR [38] left a need for an analysis of such combinations, which our work has addressed.

6.2 Vulnerabilities of Blocking Memory Reclamation Schemes

Michael [47] criticized QSBR for its unbounded memory use. No evaluation was given of the impact of this limitation on the performance of lock-free algorithms using QSBR. Sarma and McKenney [56], however, have shown that this leads to the possibility of denial-of-service attacks, and that preventing these attacks becomes an engineering problem.

The denial-of-service attacks noted by Sarma and McKenney occur in the IPv4 route

cache of the Linux 2.5.53 kernel when a large number of softirqs create a correspondingly large number of pending deletion callbacks, which cause the route cache to overflow. This vulnerability and the poor results we saw for QSBR when we combined an update-heavy workload with high CPU contention are instances of a more general problem: when we get too many callbacks, they then stress parts of the systems beyond the limits for which they were designed. Our results show that EBR is similarly vulnerable.

6.3 Performance Comparisons

Several comparisons of different QSBR implementations have been made [40, 41]. These comparisons, however, have not compared QSBR to other memory reclamation schemes. In addition, these implementations have all been made in the context of operating system kernels, and have not tested QSBR under conditions of CPU contention.

Michael [47] compared the performance of SMR to that of reference counting, and found that SMR’s performance is much better; however, he did not compare SMR to any other memory reclamation schemes. Furthermore, Michael’s experiments did not show the effect of increasing traversal length, which we show is an important weakness of SMR. Finally, Michael neither discussed nor evaluated the performance tradeoffs involved between blocking and lock-free memory reclamation schemes.

Fraser [17] criticized SMR for its overhead due to fence instructions, and cited this overhead as a reason for using his EBR scheme instead. We have shown that EBR itself has high overhead due to fences (Figure 5.2); in fact, in the base case for all algorithms, EBR has more overhead than SMR. Further, all Fraser’s experiments were run with fewer threads than CPUs. This setup does not evaluate EBR’s performance in the presence of CPU contention and an allocation-heavy workload, which we have shown is one of EBR’s major weaknesses.

Although each of these three memory reclamation schemes has been evaluated by its

respective creator, in all cases, these evaluations either ignore these competing methods or criticize them with little or no experimental evaluation. Furthermore, these evaluations have been set up in such a way that they do not expose the weaknesses of the schemes. Our contribution lies not in introducing new memory reclamation schemes, but in providing, to our knowledge, the first fair comparison of these proposals.

Similarly, many publications in the area of lock-free algorithms compare the performance of these algorithms to lock-based alternatives [23, 47, 50, 17, 43]. Similarly, comparisons have been made between concurrently-readable algorithms and more traditional locking approaches [39, 42, 38]. Ours, however is, to our knowledge, the first attempt to compare the performance of a lock-free algorithm to a concurrently-readable alternative.

Chapter 7

Conclusions and Future Work

This thesis has made three main contributions:

- We have shown that memory reclamation schemes can be chosen mostly independently of the target algorithms.
- We have analyzed the strengths and weaknesses of three memory reclamation schemes.
- Using a common memory reclamation scheme, we have made a fair comparison of lock-free and concurrently-readable chaining hash tables.

Establishing that memory reclamation schemes are mostly independent of the algorithms which use them helps to clarify much of the current literature. In particular, much of the RCU literature presents QSBR and the concurrently-readable algorithms which use it in a tightly-coupled manner. Furthermore, QSBR is typically explained in terms of its implementation in operating system kernels. Our work establishes the independence of the QSBR from the algorithms it services and its kernel-level implementations, and shows that it is useful for a wider variety of algorithms than those presented in the RCU literature. Hopefully, this realization will help others to find new uses for QSBR.

Table 7.1: Properties of Memory Reclamation Schemes

	QSBR	SMR	EBR
Base Time Overhead	Negligible	Unbounded	Constant
Memory Use	Unbounded	Bounded	Unbounded
CPU Contention	Poor When Update-Heavy	Good	Poor When Update-Heavy
Scalability			
Traversal Length Scalability	Good	Poor	Good
Application-dependent?	Yes	No	No

We have demonstrated, by comparing the performance of EBR, SMR, and QSBR, that the choice of memory reclamation scheme has a huge effect on the performance of lock-free and concurrently-readable algorithms. Choosing the right scheme for the environment in which an implementation is expected to run is essential. No memory reclamation scheme provides a silver bullet — the trade-offs between the schemes are shown in Table 7.1.

SMR and EBR have a higher base cost than QSBR because of the fence instructions they require. For EBR, the overhead due to fences is constant, while for SMR it is unbounded.

Our results show that when there is no CPU contention, or the workload involves few updates, QSBR is the best-performing memory reclamation scheme available. QSBR and EBR scale poorly when there is CPU contention and many updates, since they wait for other threads to complete their operations, thus allowing descheduled threads to delay memory reclamation.

Our comparison of the lock-free and concurrently-readable linked lists shows that the lock-free list has substantially cheaper updates, but pays a small amount of per-node overhead. This per-node overhead becomes significant at high load factors. A modified version of the lock-free list reduces this overhead considerably. When the load factor is suitably low, the lock-free hash table seems to be the best performer for most update

fractions.

Given the extremely low overhead of QSBR, experimenting with user-level QSBR implementations in a realistic application would be an interesting topic for future work. In particular, it would be interesting to build a QSBR implementation and interface which works with Pthreads or a lock-free library such as NOBLE [1]. Another topic would be to experiment with applying QSBR to a wider range of data structures and with transactional-memory-based implementations such as those of Fraser [17], whose experimental setup is available under the GPL [16].

The good performance of the lock-free hash table with QSBR is encouraging. Presently, the Linux kernel provides kernel programmers with a concurrently-readable hash table with QSBR. Given that hash tables typically have very low load factors, the per-node overhead of the lock-free hash table is unlikely to be problematic. Furthermore, a similar lock-free hash table is already part of K42 [38]. Building a lock-free hash table in Linux, and, more importantly, determining which parts of the kernel would benefit from its cheaper writes or the advantages of lock-freedom, would be another avenue for future work.

Finally, we note that SMR, Pass the Buck, and reference counting — the three lock-free memory reclamation methods — all have overhead that grows with traversal length. In the case of SMR and Pass the Buck, this overhead comes from fence instructions, while reference counting’s overhead also comes from expensive operations such as CAS. This opens the question of whether high per-node overhead is an inherent downside of lock-free memory reclamation in weakly consistent memory models; this question could be answered by designing a general-purpose lock-free memory reclamation scheme with significantly less per-node overhead which scales nicely with traversal length, or proving that no such method can exist.

Bibliography

- [1] Noble - a library of non-blocking synchronization protocols.
<http://www.noble-library.org>.
- [2] Juan Alemany and Edward W. Felten. Performance issues in non-blocking synchronization on shared-memory multiprocessors. In *Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing*, pages 125–134. ACM Press, 1992.
- [3] James H. Anderson and Mark Moir. Universal constructions for large objects. In *Proceedings of the 9th International Workshop on Distributed Algorithms*, pages 168–182. Springer-Verlag, 1995.
- [4] James H. Anderson and Mark Moir. Universal constructions for multi-object operations. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 184–193. ACM Press, 1995.
- [5] Jonathan Appavoo, Marc Auslander, Maria Burtico, Dilma Da Silva, Orran Krieger, Mark Mergen, Michal Ostrowski, Bryan Rosenburg, Robert W. Wisniewski, and Jimi Xenidis. Experience with K42, an open-source, linux-compatible, scalable operating-system kernel. *IBM Systems Journal*, 44(2):427–440, 2005.
- [6] Andrea Arcangeli, Mingming Cao, Paul E. McKenney, and Dipankar Sarma. Using read-copy update techniques for System V IPC in the Linux 2.5 kernel. In

- Proceedings of the USENIX Annual Technical Conference (FREENIX Track)*, pages 297–310. USENIX Association, June 2003.
- [7] Greg Barnes. A method for implementing lock-free shared-data structures. In *Proceedings of the 5th Annual ACM symposium on Parallel Algorithms and Architectures*, pages 261–270. ACM Press, 1993.
- [8] Hans-J. Boehm. An almost non-blocking stack. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing*, pages 40–49, 2004.
- [9] Jeff Bonwick and Jonathan Adams. Magazines and vmem: Extending the slab allocator to many CPUs and arbitrary resources. In *USENIX Annual Technical Conference, General Track 2001*, pages 15–33, 2001.
- [10] David L. Detlefs, Christine H. Flood, Alex Garthwaite, Paul Martin, Nir Shavit, and Guy L. Steele Jr. Even better DCAS-based concurrent deques. In *Proceedings of the 14th International Conference on Distributed Computing*, pages 59–73. Springer-Verlag, 2000.
- [11] David L. Detlefs, Paul A. Martin, Mark Moir, and Guy L. Steele Jr. Lock-free reference counting. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing*, 2001.
- [12] David L. Detlefs, Paul A. Martin, Mark Moir, and Guy L. Steele Jr. Lock-free reference counting. *Distributed Computing*, 15(4):255–271, 2002.
- [13] Simon Doherty, David L. Detlefs, Lindsay Grove, Christine H. Flood, Victor Luchangco, Paul A. Martin, Mark Moir, Nir Shavit, and Guy L. Steele Jr. DCAS is not a silver bullet for nonblocking algorithm design. In *Proceedings of the 16th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 216–224. ACM Press, 2004.

- [14] Simon Doherty, Maurice Herlihy, Victor Luchangco, and Mark Moir. Bringing practical lock-free synchronization to 64-bit applications. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing*, pages 31–39. ACM Press, 2004.
- [15] Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing*, pages 50–59. ACM Press, 2004.
- [16] Keir Fraser. Lock-free library. Source code release. Available: <http://www.cl.cam.ac.uk/Research/SRG/netos/lock-free/src/lockfree-lib.tar.gz>.
- [17] Keir Fraser. *Practical Lock-Freedom*. PhD thesis, University of Cambridge Computer Laboratory, 2004.
- [18] Benjamin Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, pages 87–100, 1999.
- [19] Michael Greenwald. *Non-blocking Synchronization and System Design*. PhD thesis, Stanford University, 1999.
- [20] Michael Greenwald and David Cheriton. The synergy between non-blocking synchronization and operating system structure. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 123–136. ACM Press, 1996.
- [21] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle

- Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, Washington, DC, USA, 2004. IEEE Computer Society.
- [22] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing*, pages 300–314. Springer-Verlag, 2001.
- [23] Timothy L. Harris, Keir Fraser, and Ian Pratt. A practical multi-word compare-and-swap operation. In *Proceedings of the IEEE Symposium on Distributed Computing*, October 2002.
- [24] Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. In *Proceedings of the 16th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 206–215. ACM Press, 2004.
- [25] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.
- [26] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.
- [27] Maurice Herlihy, Victor Luchangco, and Mark Moir. Brief announcement: Dynamic-sized lock-free data structures. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing*, 2002.
- [28] Maurice Herlihy, Victor Luchangco, and Mark Moir. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures. In *Proceedings of the 16th International Symposium on Distributed Computing*, October 2002.

- [29] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, page 522. IEEE Computer Society, 2003.
- [30] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing*, July 2003.
- [31] H. T. Kung and Philip L. Lehman. Concurrent manipulation of binary search trees. *ACM Transactions on Database Systems*, 5(3):354–382, 1980.
- [32] Leslie Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, 1977.
- [33] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.
- [34] Doug Lea. Concurrency: where to draw the lines. Invitational Workshop on the Future of Virtual Execution Environments, September 2004. Available: <http://www.research.ibm.com/vee04/Lea.pdf>.
- [35] Robert Love. *Linux Kernel Development*. Sam’s Publishing, 2004.
- [36] Udi Manber and Richard E. Ladner. Concurrency control in a dynamic search structure. In *Proceedings of the 1st ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 268–282. ACM Press, 1982.
- [37] Henry Massalin and Calton Pu. A lock-free multiprocessor OS kernel. Technical Report CUCS-005-91, Computer Science Department, Columbia University, June 1991.

- [38] Paul E. McKenney. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels*. PhD thesis, OGI School of Science and Engineering at Oregon Health and Sciences University, 2004.
- [39] Paul E. McKenney. RCU vs. locking performance on different CPUs. In *linux.conf.au*, Adelaide, Australia, January 2004.
- [40] Paul E. McKenney, Jonathan Appavoo, Andi Kleen, Orran Krieger, Rusty Russell, Dipankar Sarma, and Maneesh Soni. Read-copy update. In *Ottawa Linux Symposium*, July 2001.
- [41] Paul E. McKenney, Dipankar Sarma, Andrea Arcangeli, Andi Kleen, Orran Krieger, and Rusty Russell. Read-copy update. In *Ottawa Linux Symposium*, pages 338–367, June 2002.
- [42] Paul E. McKenney and John D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, Las Vegas, NV, October 1998.
- [43] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the 14th Annual ACM Symposium Parallel Algorithms and Architectures*, pages 73–82, August 2002.
- [44] Maged M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing*, pages 21–30, July 2002.
- [45] Maged M. Michael. CAS-based lock-free algorithm for shared deques. In *The 9th Euro-Par Conference on Parallel Processing*, volume 2790, pages 651–660, August 2003.

- [46] Maged M. Michael. ABA prevention using single-word instructions. Technical Report RC 23089, IBM T. J. Watson Research Center, January 2004.
- [47] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, June 2004.
- [48] Maged M. Michael. Practical lock-free and wait-free ll/sc/vl implementations using 64-bit cas. In *The 18th International Conference on Distributed Computing*, October 2004.
- [49] Maged M. Michael. Scalable lock-free dynamic memory allocation. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 35–46, June 2004.
- [50] Maged M. Michael and Michael L. Scott. Correction of a memory management method for lock-free data structures. Technical Report TR599, Computer Science Department, University of Rochester, December 1995.
- [51] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th annual ACM Symposium on Principles of Distributed Computing*, pages 267–275. ACM Press, 1996.
- [52] Mark Moir. Transparent support for wait-free transactions. In *Proceedings of the 11th International Workshop on Distributed Algorithms*, pages 305–319. Springer-Verlag, 1997.
- [53] William Pugh. Concurrent maintenance of skip lists. Technical Report CS-TR-2222, Department of Computer Science, University of Maryland, June 1990.

- [54] Ravi Rajwar and James R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th International Symposium on Microarchitecture*, December 2001.
- [55] Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.
- [56] Dipankar Sarma and Paul E. McKenney. Issues with selected scalability features of the 2.6 kernel. In *Ottawa Linux Symposium*, July 2004.
- [57] Håkan Sundell. Fast and lock-free concurrent priority queues for multi-thread systems. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium*, April 2003.
- [58] Håkan Sundell. *Efficient and Practical Non-Blocking Data Structures*. PhD thesis, Chalmers University of Technology, 2004.
- [59] Håkan Sundell. Wait-free reference counting and memory management. In *Proceedings of the 19th International Parallel and Distributed Processing Symposium*, April 2005.
- [60] R. Kent Treiber. Systems programming: Coping with parallelism. Research Report RJ 5118, IBM Almaden Research Center, April 1986.
- [61] John D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 214–222, August 1995.

Performance of memory reclamation for lockless synchronization[☆]

Thomas E. Hart^{a,*},¹ Paul E. McKenney^b, Angela Demke Brown^a, Jonathan Walpole^c

^aDepartment of Computer Science, University of Toronto, Toronto, Ont., Canada M5S 2E4

^bIBM Linux Technology Center, IBM Beaverton, Beaverton, OR 97006, USA

^cDepartment of Computer Science, Portland State University, Portland, OR 97207-0751, USA

Received 1 July 2006; received in revised form 11 April 2007; accepted 25 April 2007

Available online 3 May 2007

Abstract

Achieving high performance for concurrent applications on modern multiprocessors remains challenging. Many programmers avoid locking to improve performance, while others replace locks with non-blocking synchronization to protect against deadlock, priority inversion, and convoying. In both cases, dynamic data structures that avoid locking require a *memory reclamation scheme* that reclaims elements once they are no longer in use.

The performance of existing memory reclamation schemes has not been thoroughly evaluated. We conduct the first fair and comprehensive comparison of three recent schemes—*quiescent-state-based reclamation*, *epoch-based reclamation*, and *hazard-pointer-based reclamation*—using a flexible microbenchmark. Our results show that there is no globally optimal scheme. When evaluating lockless synchronization, programmers and algorithm designers should thus carefully consider the data structure, the workload, and the execution environment, each of which can dramatically affect the memory reclamation performance.

We discuss the consequences of our results for programmers and algorithm designers. Finally, we describe the use of one scheme, quiescent-state-based reclamation, in the context of an OS kernel—an execution environment which is well suited to this scheme.

© 2007 Elsevier Inc. All rights reserved.

Keywords: Lockless; Non-blocking; Memory reclamation; Hazard pointers; Read-copy update; Synchronization; Concurrency; Performance

1. Introduction

As multiprocessors become mainstream, multithreaded applications will become more common, increasing the need for efficient coordination of concurrent accesses to shared data structures. Traditional locking requires expensive atomic operations, such as compare-and-swap (CAS), even when locks are uncontended. For example, acquiring and releasing an uncontended spinlock requires over 400 cycles on an IBM® POWER™ CPU. Therefore, many researchers recommend avoiding locking [3,10,28]. Some systems, such as the Linux™

kernel, use *concurrently readable* synchronization, which uses locks for updates but not for reads. Locking is also susceptible to priority inversion, convoying, deadlock, and blocking due to thread failure [5,13], leading researchers to pursue *non-blocking* (or *lock-free*) synchronization [9,15–17,19,39]. In some cases, lock-free approaches can bring performance benefits [31]. For clarity, we describe as *lockless* all synchronization strategies which permit access to shared data without using locks.

A major challenge for lockless synchronization is handling the *read/reclaim races* that arise in dynamic data structures. Fig. 1 illustrates this problem. Threads *T*1 and *T*2 both hold references to element *a* of a linked list, and *T*1's removal of element *a* is concurrent with *T*2's read of *a*'s *next* field. The memory occupied by removed elements must be reclaimed to allow reuse, or memory exhaustion will eventually block all threads; however, reclaiming *a* is unsafe while *T*2 continues referencing it, since after reclamation, *a*'s contents would no longer be defined, and *a*'s *next* field might not be a valid pointer. Thread *T*2 could therefore crash or corrupt the contents of

[☆] Portions of this paper appeared in the Proceedings of the 2006 International Parallel and Distributed Processing Symposium (IPDPS 2006).

* Corresponding author.

E-mail addresses: tomhart@cs.toronto.edu (T.E. Hart), paulmck@us.ibm.com (P.E. McKenney), demke@cs.toronto.edu (A.D. Brown), walpole@cs.pdx.edu (J. Walpole).

¹ Supported by an NSERC Canada Graduate Scholarship.

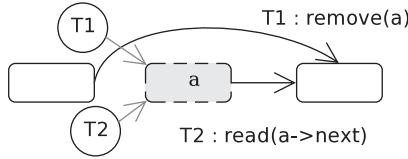


Fig. 1. Concurrent reading and writing causes read/reclaim races.

memory. The program or the system must somehow determine when a can safely be reclaimed.

Reclamation is subsumed into automatic garbage collectors in environments that provide them, such as Java™. Provided the garbage collector is thread-safe, programmers using garbage-collected languages therefore need not worry about read/reclaim races. However, for languages like C, where memory must be explicitly reclaimed (e.g. via `free()`), programmers must combine a *memory reclamation scheme* with their lockless data structures to resolve these read/reclaim races. Several such reclamation schemes have been proposed.

Programmers need to understand the semantics and the performance implications of each scheme, since the overhead of inefficient reclamation can be worse than that of locking. For example, *reference counting* [8,39] has high overhead in the base case and scales poorly with data-structure size. This is unacceptable when performance is the motivation for lockless synchronization. Unfortunately, there is no single optimal scheme, and existing work is relatively silent on factors affecting reclamation performance.

We address this deficit by comparing three recent reclamation schemes, showing the respective strengths and weaknesses of each. In Sections 2 and 3, we review these schemes and describe factors affecting their performance. Section 4 explains our experimental setup. Our analysis, in Section 5, reveals substantial performance differences between these schemes, the greatest source of which is per-operation atomic instructions. In Section 6, we discuss the relevance of our work to designers and implementers. We show that lockless algorithms and reclamation schemes are mostly independent, by combining a blocking reclamation scheme and a non-blocking algorithm, then comparing this combination to a fully non-blocking equivalent. We also present a new reclamation scheme that combines aspects of two other schemes to give good performance and ease of use. Section 7 describes the use of one of these memory reclamation schemes (QSBR) in the Linux kernel. We close with a discussion of related work in Section 8 and summarize our conclusions in Section 9.

2. Memory reclamation schemes

We examine four memory reclamation schemes: quiescent-state-based reclamation (QSBR) [3,28], epoch-based reclamation (EBR) [9], hazard-pointer-based reclamation (HPBR) [29,30], and lock-free reference counting (LFRC) [39,32]. In this section, we provide an overview of each scheme to help the reader understand our work.

Most of the functionality of each scheme is provided by a library into which clients call. However, each scheme places slightly different constraints on the calling code, leading to different interfaces to the respective libraries; hence, a developer cannot simply recompile her application with a new reclamation scheme, but must customize her code for each library's interface. These interfaces have different levels of complexity, leading to trade-offs between reclamation performance and coding difficulty. We elaborate on this point while discussing each scheme.

2.1. Quiescent-state-based reclamation

QSBR and EBR reclaim memory once a *grace period* has passed. A *grace period* is a time interval $[a, b]$ such that, after time b , all elements removed before time a can safely be reclaimed.

QSBR uses quiescent states to detect grace periods. A *quiescent state* for thread T is a state in which T holds no references to shared elements—in particular, T holds no references to any shared elements which have been removed from a lockless data structure. Any interval of time in which each thread passes through at least one quiescent state is thus a grace period for QSBR. Fig. 2 illustrates this relationship. Thread T_1 goes through quiescent states at times t_1 and t_5 , T_2 at times t_2 and t_4 , and T_3 at time t_3 . Hence, a grace period is any time interval containing either $[t_1, t_3]$ or $[t_3, t_5]$.

Note that there is no requirement that QSBR implementations find the shortest grace periods possible. In Fig. 2, for example, any interval containing $[t_1, t_3]$ or $[t_3, t_5]$ is a quiescent state; implementations which check for grace periods only when threads enter quiescent states would detect $[t_1, t_5]$, since T_1 's two quiescent states form the only pair of quiescent states from a single thread which enclose a grace period.

One convenient way to implement QSBR is with a *fuzzy barrier* [14]. A barrier protects access to some code which no thread should execute before all other threads finish some prior stage of computation. In standard (non-fuzzy) barrier synchronization, threads announce their entry into a barrier, and then block until all threads have entered the barrier. In a fuzzy barrier, instead of blocking, a thread which enters the barrier skips the protected code and continues executing if some other thread has not yet entered the barrier. The thread will again attempt to execute the protected code upon subsequent fuzzy barrier entries. For implementing QSBR, a thread can enter the barrier when passing through a quiescent state; the protected code performs the memory reclamation.

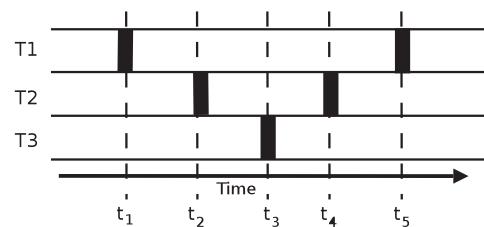


Fig. 2. Illustration of QSBR. Black boxes represent quiescent states.

Applications must somehow indicate to the QSBR library when quiescent states occur; however, the choice of quiescent states is application-dependent. In general, a thread may declare a quiescent state at any time when it has no references to any shared data. In the trivial case, a thread could declare a quiescent state after every lockless operation, as shown in Listing 1; however, it is often advantageous to declare quiescent states less frequently. In our experiments, we declare quiescent states by calling `quiescent_state()` at the end of our main test loop, as shown in Listing 4 in Section 4. A thread which calls `quiescent_state()` enters a fuzzy barrier. Calling `quiescent_state()` at the end of the loop allows us to amortize the cost of entering the fuzzy barrier across several operations.

Listing 1 : Trivial example flagging of quiescent states with QSBR.

```

1 void foo (struct list *l, long key)
2 {
3     remove (l, key);
4     quiescent_state ();
5 }
```

Many operating system kernels contain natural quiescent states. Linux uses QSBR to implement the *read-copy update* (RCU) API [10,26,28]. Several choices of quiescent state have been proposed in different QSBR implementations [24, Section 4.3] in Linux. A classical example is voluntary context switch. As another example, in parts of the Linux kernel, writers flag quiescent states immediately after writes, using the pattern shown in Listing 1, and then intentionally block until a grace period has elapsed, in order to increase maintainability and to reduce memory usage.

The standard model for reasoning about non-blocking synchronization is that any thread may experience a *fail-stop* failure at any time, and other threads cannot distinguish these failures from long stalls. In this model, QSBR is blocking, since failed threads will not go through quiescent states. Fig. 3 illustrates this problem. Thread *T₂* is a failed thread; since it has failed, it never goes through a quiescent state. By the definition of *grace period*, there are thus no grace periods in this system. Since there are no grace periods, threads *T₁* and *T₃* will never be able to reclaim memory. As a result, the system will eventually run out of memory, forcing *T₁* and *T₃* to block forever on memory allocation. In principle, systems that can detect thread failure might designate thread failure as an extended quiescent state; however, in practice, we are not aware of any such implementation, and in theory, failure detectors [7] are not a part of standard shared memory models.

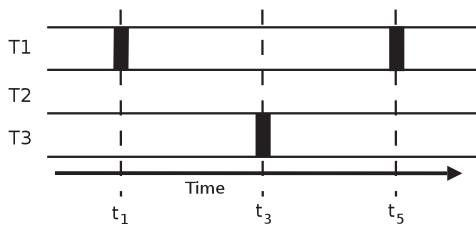


Fig. 3. QSBR is inherently blocking.

2.2. Epoch-based reclamation

Fraser's EBR [9], like QSBR, uses grace periods. EBR differs from QSBR in that QSBR relies on the programmer to annotate the program with quiescent states, but EBR hides this bookkeeping within the implementation of lockless operations. The body of a lockless operation is termed a *critical region*. Each thread sets a per-thread flag upon entry into a critical region, indicating that the thread intends to locklessly access shared data. The thread clears this flag at the end of the lockless operation. No thread is allowed to access an EBR-protected object outside of a critical region. After some pre-determined number of critical region entries, a thread attempts to enter a fuzzy barrier and reclaim memory.

Listing 2 shows an example of the use of EBR in a search of a linked list which allows lockless reads but uses locks for updates. QSBR omits lines 5, 10, and 14, which handle EBR's epoch bookkeeping, but is otherwise identical; QSBR's quiescent states are flagged explicitly by clients of the QSBR library, as shown by the pseudocode in Listings 1 and 4.

Listing 2: EBR concurrently readable search.

```

1 int search (struct list *l, long key)
2 {
3     element_t *cur;
4     int cur_key;
5     critical_enter();
6     for (cur = l->list_head;
7          cur != NULL; cur = cur->next) {
8         cur_key = cur->key;
9         if (cur->_key >= key) {
10            critical_exit();
11            return (cur_key == key);
12        }
13    }
14    critical_exit();
15    return (0);
16 }
```

EBR gets its name from the epochs it uses to implement a fuzzy barrier. Each thread executes in one of the three logical epochs and may lag at most one epoch behind the *global epoch*. Each epoch has an associated limbo list for elements awaiting reclamation. Whenever a thread enters a new epoch, it accesses the code protected by the fuzzy barrier and can safely reclaim memory. Three epochs are needed because, as Fig. 4 illustrates, a thread can execute in two epochs during a single global epoch, hence populating two limbo lists.

Fig. 4 shows how EBR tracks epochs, allowing memory to be reclaimed safely. When a thread enters a critical region, it updates its local epoch to match the global epoch. After some pre-determined number of critical region entries since changing its local epoch, a thread will attempt to increment the global epoch. This attempt will succeed only if the local epoch of each thread in a critical region is equal to the global epoch; hence, since threads update their local epochs only at the beginning of a critical region, if the global epoch is *e*, threads in critical

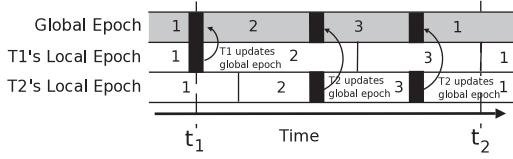


Fig. 4. Illustration of EBR. Thin solid lines show updates to a thread's local epoch, and thick solid lines show updates to both a local epoch and the global epoch.

regions can be in either epoch e or $e - 1$, but not $e + 1$ (all mod 3). Hence, when a thread T 's local epoch changes to e , all lockless operations of other threads which were in progress the last time T was in epoch e have completed—a grace period has elapsed. The time period $[t_1, t_2]$ in Fig. 4 is thus a grace period.

As with QSBR, reclamation can be stalled by failed threads; however, unlike with QSBR, only threads that fail *within* critical regions can stall EBR. EBR's bookkeeping is invisible to the application programmer, making it simple for a programmer to use. Section 5 shows that this property imposes significant overhead on EBR.

2.3. Hazard-pointer-based reclamation

Michael's HPBR[29] scheme, sometimes called *safe memory reclamation* (SMR), provides an existence locking mechanism for dynamically allocated elements. Each thread performing lockless operations has K hazard pointers which it uses to protect elements from reclamation by other threads; hence, if there are N threads, we have $H = NK$ hazard pointers in total. K is data-structure-dependent, and often small. Queues and linked lists need $K = 2$ hazard pointers, while stacks require only $K = 1$; however, we know of no upper bound on K for general tree or graph traversal algorithms.

After removing an element, a thread places that element in a private list. When the list grows to a predefined size R , the thread reclaims each removed element lacking a corresponding hazard pointer. Increasing R amortizes reclamation overhead across more elements, but increases memory usage; if R is bigger than H by some amount proportional to H , the amortized per-element processing time is constant. Setting R to a positive integer multiple of H plus some constant suffices. Furthermore, since every removed element not protected by a hazard pointer can be reclaimed, at most H of the removed elements can be unreclaimable.

Since each thread has K hazard pointers and can hold R removed elements in its private list, a crashed thread can prevent only $K + R$ removed elements from being reclaimed. HPBR thus bounds the amount of memory which can be occupied by removed elements, even in the presence of thread failures. In this sense, HPBR is non-blocking—memory held by removed elements cannot grow arbitrarily and exhaust the system's memory, which would otherwise cause threads to stall. This guarantee, however, assumes a finite number of threads and hazard pointers.

An algorithm using HPBR must identify all *hazardous references*—references to shared elements that may have been

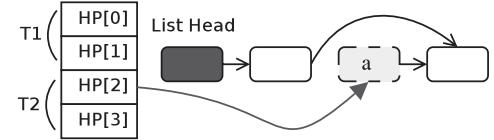


Fig. 5. Illustration of HPBR.

removed by other threads or that are vulnerable to the ABA² problem [30]. Such references require hazard pointers. The algorithm sets a hazard pointer, then checks for element removal; if the element has not been removed, then its fields may safely be accessed. As long as the hazard pointer references the element, HPBR's reclamation routine refrains from reclaiming it. Fig. 5 illustrates the use of HPBR. Element a has been removed from the linked list, but cannot be reclaimed because $T2$'s hazard pointer $HP[2]$ references it.

Listing 3, showing code adapted from Michael [30], demonstrates HPBR with a search algorithm corresponding to Listing 2. At most two elements must be protected: the current element and its predecessor ($K = 2$). The code removing elements, which are not shown here, use the low-order bit of the next pointer as a flag. This guarantees that the validation step on line 14 will fail and retry in case of concurrent removal. Full details are given by Michael [30].

Listing 3: HPBR concurrently readable search.

```

1 int search (struct list *l, long key)
2 {
3     element_t **prev, *cur, *next;
4     /* Index of our first hazard pointer. */
5     int base = getTID() * K;
6     /* Offset into our hazard pointer segment. */
7     int off = 0;
8     try_again:
9     prev = &l->list_head;
10    for (cur = *prev; cur != NULL; cur = next & ~1) {
11        /* Protect cur with a hazard pointer. */
12        HP[base+off] = cur;
13        memory_fence();
14        if (*prev != cur)
15            goto try_again;
16        next = cur->next;
17        if (cur->key >= key)
18            return (cur->key == key);
19        prev = &cur->next;
20        off = (off+1)
21    }
22    return (0);
23 }
```

² The ABA problem [20] occurs when we use CAS to update a data structure. Suppose that p is an element in a linked data structure. If some thread removes p and replaces it with p' , which uses the *same memory* previously occupied by p , a concurrent CAS operation (for example, another thread trying to remove p) will not be able to tell the difference between p' and p . This CAS operation could then succeed when it should fail (for example, unintentionally removing p').

Provided that no lockless operation returns a reference to an element, the changes needed for a program to use HPBR are localized to the implementation of lockless operations, as Listing 3 shows. If, however, a function returns a reference to an element, a hazard pointer must protect this element during the reference's lifetime. Furthermore, applying HPBR to the implementation of a lockless operation is often more complicated than applying EBR, as can be seen by contrasting Listing 2 with 3.

Herlihy et al. [18] presented a very similar scheme called Pass the Buck. Since this scheme's per-operation costs are very similar to those of HPBR, we believe that our HPBR results apply to Pass the Buck as well.

2.4. Lock-free reference counting

LFRC is a well-known garbage-collection technique. Threads track the number of references to elements, reclaiming any element whose count is zero. Valois' LFRC scheme [39] (corrected by Michael and Scott [32]) uses CAS and fetch-and-add (FAA), and requires elements to retain their type after reclamation. Sundell's scheme [37], based on Valois', is wait-free. The scheme of Detlefs et al. [8] allows elements' types to change upon reclamation, but requires double compare-and-swap (DCAS), which no current CPU supports.

Although LFRC avoids locks, it does not bound the amount of memory consumed by removed nodes like HPBR does—Michael and Scott report easily running out of memory using Valois' version of LFRC [32]. Furthermore, Michael [30] showed that LFRC introduces overhead which often makes lockless algorithms perform worse than lock-based versions. We include some experiments with Valois' scheme to reproduce Michael's findings.

As with HPBR, only the implementations of lockless operations must be changed in order to support LFRC, provided these operations do not return references to elements. Implementing these changes is slightly more complex than for HPBR, since the programmer must ensure that each reference count which is incremented is later decremented; by contrast, if hazard pointers are reused on subsequent operations, it is not necessary to explicitly unset them.

2.5. Summary

We consider QSBR, EBR, HPBR, and LFRC. For the convenience of the reader, we list these schemes, their associated acronyms, and some basic characteristics of each in Table 1;

this list also includes *new epoch-based reclamation* (NEBR), which we introduce in Section 6.2.

3. Reclamation performance factors

Several factors can affect the performance of memory reclamation schemes—memory consistency, workload, contention, thread preemption, scheduling, and memory constraints. This section explains these factors, which we vary experimentally in Section 5.

3.1. Memory consistency

Current literature on lock-free algorithms generally assumes a sequentially consistent [23] memory model, which prohibits instruction reordering and globally orders memory references. However, sequential consistency precludes many hardware and compiler performance optimizations which are possible when using a weaker memory consistency model [1]. Since most codes does not require sequential consistency, modern CPUs enforce sequential consistency only when needed by having programmers use special *fence* instructions (also called *memory barriers*). Although fences are often omitted from pseudocode, they are expensive on most modern CPUs and must be included in realistic performance analyses.

The schemes we consider require different numbers of fence instructions. HPBR, EBR, and LFRC require per-operation fences, while QSBR does not. HPBR, as shown in Listing 3, requires a fence between hazard-pointer setting and validation, thus one fence per visited element. LFRC also requires per-element fences, in addition to the atomic instructions needed to maintain reference counts. EBR requires two fences per operation: one when setting a flag when entering a critical region, and one when clearing it upon exit. QSBR has no per-operation code to manage quiescent states, so no per-operation fences are required. As we show in Section 5, this lack of per-operation fences enables QSBR to have very low per-operation overhead in many cases.

3.2. Workload, contention, and scheduling

Data structures differ in both the operations they provide and in their common workloads. Queues are update-only, but linked lists and hash tables are often read mostly [24]. Schemes which do not bound memory usage may perform poorly with update-heavy structures, since the risk of memory exhaustion is higher. Conversely, schemes which require per-element fences

Table 1
Summary of memory reclamation schemes

Acronym	Full name	Characteristics
QSBR	Quiescent-state-based reclamation	Detects grace periods using application-dependent quiescent states
EBR	Epoch-based reclamation	Detects grace periods using application-independent epochs
HPBR	Hazard-pointer-based reclamation	Uses per-thread hazard pointers for existence locking
LFRC	Lock-free reference-counting	Uses per-element reference counts for existence locking
NEBR	New epoch-based reclamation	Introduced in Section 6.2

may perform poorly with operations which must visit many elements, such as list or tree traversal.

We expect contention due to concurrent threads to be a minor source of reclamation overhead; however, for HPBR and LFRC, it could be unbounded in degenerate cases. Readers using these schemes may have to restart their traversals due to interference from concurrent writes—for example, as shown in lines 14 and 15 of Listing 3. Readers forced to repeatedly restart their traversals must *repeatedly* execute fence instructions for every element. These degenerate cases are more likely when there are many threads and the workload is update-heavy.

When the number of threads exceeds the number of processors, threads will be preempted. Preemption can adversely affect schemes which rely on grace periods—descheduled threads will not go through quiescent states or update their local epochs, and can thus delay reclamation, potentially exhausting memory. This risk of memory exhaustion is greatest with update-heavy workloads. Longer scheduling quanta may increase the risk of this exhaustion.

3.3. Memory constraints

Typically, a multi-threaded memory allocator will give each thread a local pool of memory; threads that exhaust their local pools replenish them from a global pool [4,6]. Using local pools reduces contention on the global pool. Although lock-free memory allocators exist [31], many allocators protect the global pool using locking.

If an allocator uses locking, schemes which do not bound memory usage may see greater lock contention due to having to access the lock-protected global pool more frequently. Furthermore, if a thread is preempted while holding such a lock, other threads will block on memory allocation. The size of the global pool is finite and governs the likelihood of memory exhaustion. Only HPBR [29] provides a provable bound on the amount of unreclaimed memory; it should thus be less sensitive to these constraints.

4. Experimental setup

We evaluated the memory reclamation strategies with respect to the factors outlined in Section 3 using commodity SMP systems with IBM POWER CPUs. Table 2 shows the characteristics of the two machines we used; the last line of this table gives the combined costs of locking and then unlocking a spinlock. The code for our experiments is available at <http://www.cs.toronto.edu/~tomhart/perflab/ipdps06.tgz>.

In our tests, a parent thread creates N child threads, starts a timer, and stops the threads upon timer expiry. Child threads count the number of operations they perform, and the parent then calculates the average *execution time* per operation by dividing the duration of the test by the total number of operations. The *CPU time* is the execution time multiplied by the number of threads. Provided that the threads do not outnumber the CPUs, CPU time compensates for increasing numbers of CPUs, allowing us to focus on synchronization overhead.

Table 2
Characteristics of machines

	XServer	IBM Power
CPUs	2x 2.0 GHz PowerPC G5	8x 1.45 GHz Power4+
Kernel	Linux 2.6.8-1.ydl.7g5-smp	Linux 2.6.13 (kernel.org)
Fence	78 ns (156 cycles)	76 ns (110 cycles)
CAS	52 ns (104 cycles)	59 ns (86 cycles)
Lock	231 ns (462 cycles)	243 ns (352 cycles)

In our results, we report execution time when there are more threads than CPUs, and CPU time otherwise. We average our times over five trials.

In each trial, each thread runs repeatedly through the test loop shown in Listing 4 until the timer expires. QSBR tests place a quiescent state at the end of the loop. The probabilities of inserting and removing elements are equal, keeping data-structure size roughly constant throughout a given run.

Listing 4: Per-thread test pseudocode.

```

1  while (parent's timer has not expired) {
2    for i from 1 to OPS_PER_LOOP do {
3      key = random key;
4      op = random operation;
5      d = data structure;
6      op(d, key);
7    }
8    if (using QSBR)
9      quiescent_state();
10 }
```

We tested the reclamation schemes on linked lists and queues. We used Michael's ordered lock-free linked list, which forbids duplicate keys, and Michael and Scott's lock-free queue. These data structures are known to work with HPBR and have been previously evaluated with that technique [30]; choosing these data structures makes our work more easily comparable with this prior work. We coded our concurrently readable lists similarly to the lock-free lists. Linked lists permit arbitrary lengths and read-to-update ratios, so we used them heavily in our experiments. Queues allow evaluating QSBR on an update-only data structure, which no prior studies have done.

The tests allow us to vary the number of threads and the total number of elements with which the experiment begins. When using linked lists, we are also able to specify the ratio of reads to updates.

QSBR, EBR, and HPBR all have parameters which affect the frequency of reclamation; we attempted to choose these parameters such that the schemes performed well and reclaimed memory with comparable frequency. These factors include the number of operations per quiescent state, the frequency with which threads using EBR attempt to update the global epoch, and the frequency with which threads using HPBR attempt to reclaim memory. We chose these parameters so as not to bias our experiments against any scheme. As shown in Listing 4, each thread performs OPS_PER_LOOP operations per quiescent state; hence, grace-period-related overhead is amortized across OPS_PER_LOOP operations. We set the value of OPS_PER_LOOP to 100 in our experiments. For EBR, each op

in Listing 4 is a critical region; a thread attempts to update the global epoch whenever it has entered a critical region 100 times since the last update, again amortizing grace-period-related overhead across 100 operations. For HPBR, we amortized reclamation overhead over $R=2H+100$ element removals.

Both QSBR and EBR require a fuzzy barrier algorithm; however, measuring the performance and scalability of different barrier algorithms is not our goal. We therefore chose to use EBR's fuzzy barrier algorithm for both our QSBR and EBR implementations. We experimented with other barrier algorithms and found that this one had low overhead and scaled well. In principle, we could use other fuzzy barrier algorithms for each scheme while maintaining the same programming interface.

Our memory allocator is similar to that of Bonwick [6]: each thread has two freelists of up to 100 elements each and can acquire more memory from a global non-blocking stack of freelists. This non-blocking allocator allowed us to study reclamation performance without considering pathological locking conditions discussed in Section 3.3.

The threads in our experiment were processes, created using `fork()`. We implemented CAS using POWER's LL/SC instructions (`laxx` and `stcx`), and fences using the `eieio` instruction. Our spinlocks were implemented using CAS and fences. Our locks used exponential backoff [2], implemented using busy waiting, upon encountering conflicts, as did all our lockless algorithms.

4.1. Limitations of experiment

Microbenchmarks are never perfect [22]; however, they allow us to study reclamation performance by varying each of the factors outlined in Section 3 independently. Our results show that these factors significantly affect reclamation performance. In macrobenchmark experiments, it is more difficult to gain insight into the causes of performance differences, and to test the schemes comprehensively.

Some applications may not have natural quiescent states; furthermore, detecting quiescent states in other applications may be more expensive than it is in our experiments. Our QSBR implementation, for example, is faster than that used in the Linux kernel, due to the latter's need to support dynamic insertion and removal of CPUs, interrupt handlers, and real-time workloads.

Our HPBR experiments statically allocate hazard pointers. Although this is sufficient for our experiments, some algorithms, to the best of our knowledge, require unbounded numbers of hazard pointers.

We believe that, despite the above limitations, our experiments thoroughly evaluate these memory reclamation schemes and show when each scheme is and is not efficient. In Section 7, we describe experiments with one scheme, QSBR, in the context of the Linux kernel and show that these macrobenchmark results are consistent with the predictions of our microbenchmark.

5. Performance analysis

We first investigate the base costs for the reclamation schemes: single-threaded execution on small data structures.

We then show how workload, list traversal length, number of threads, and preemption affect the performance.

5.1. Base costs

We first measure the costs of these schemes without the costs associated with contention, preemption, or traversing long lists; this represents the best-case performance of these schemes. Fig. 6 shows the single-threaded base costs of these schemes on non-blocking queues and single-element linked lists with no preemption or contention. We note that in a well-designed system, contention should usually be low—good best-case performance is thus highly desirable.

For the purposes of comparison, we show how these lockless algorithms, with the support of the different reclamation schemes, compare with simple spinlock-based alternatives in the base case. These data are presented only to show that the lockless schemes are competitive: our goal is to compare different memory reclamation schemes, not to compare lock-based algorithms to lockless ones, or to compare different types of locks. We therefore limit our experiments to lockless synchronization for the evaluation in this section.

We ran LFRC only on read-only workloads; these were sufficient for us to corroborate Michael's [30] result that LFRC performs poorly.

In these base cases, the dominant influence on the performance is per-operation atomic instructions: CAS, FAA, and fences make LFRC much more expensive than the other schemes. Since EBR requires two fences per operation (when calling `critical_enter()` and `critical_exit()`, respectively), and HPBR requires one for most operations considered here, EBR is usually the next most expensive. QSBR, needing no per-operation atomic instructions, is the cheapest scheme in the base case.

Workload affects the performance of these schemes. Under an update-intensive workload, a significant number of operations will involve removing elements; for each attempt to reclaim a removed element, HPBR must search the array of hazard pointers. This overhead can become significant for update-intensive workloads, as can be seen in Fig. 6: HPBR performs best for the read-only linked lists and worst for the

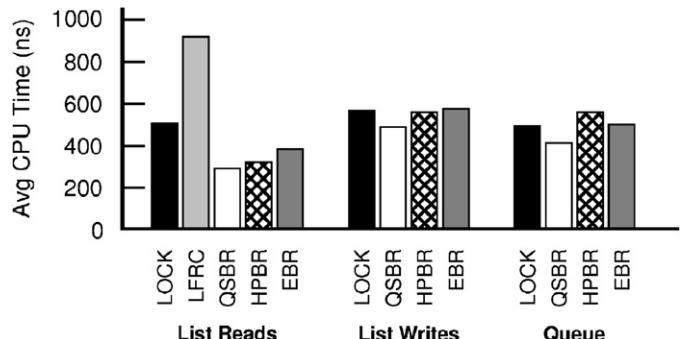


Fig. 6. Base costs—single-threaded data from 8-CPU machine. Y-axis shows *CPU time*, defined as the average per-operation execution time multiplied by the number of threads.

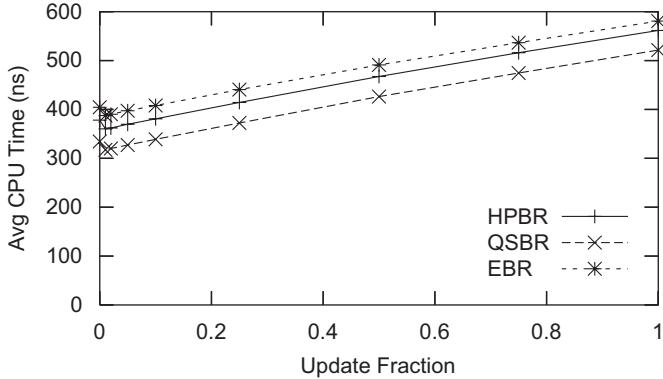


Fig. 7. Lock-free list, one thread, one element, 8-CPUs, varying workload.

update-only queues. HPBR's performance for the update-only linked lists is intermediate—list removals fail when a matching key is not found in the list, and therefore do not result in an element needing reclamation, and therefore do not result in an element needing reclamation. Reclamation overhead is thus amortized across more operations compared to the update-only queue case, in which dequeue operations fail only when the queue is empty.

5.2. Scalability with workload

Fig. 6 shows us how the reclamation schemes perform with read-only and update-only workloads. Using linked lists allows us to see how the schemes perform with intermediate read-to-update ratios. Fig. 7 shows the schemes' performance as we gradually increase the read-to-update ratio from read-only to update-only on a single-threaded workload. QSBR and EBR exhibit almost the same slope. HPBR, however, experiences a slight increase in overhead as the update fraction increases. As noted above, this is due to the fact that HPBR has to perform additional processing each time it reclaims a removed element. Since this increase is small, we conclude that workload, in isolation, is a minor factor in reclamation performance.

5.3. Scalability with traversal length

Fig. 8 shows the effect of list length on a single-threaded read-only workload. We observed similar results in update-only workloads. As expected, per-element fence instructions degrade HPBR's performance on long chains of elements; QSBR and EBR do much better.

Fig. 9 shows the same scenario, but also includes LFRC. At best, LFRC takes more than twice as long as the next slowest scheme, and the performance gap rapidly increases with the list length due to the multiple per-element atomic instructions. Because LFRC is always the worst scheme in terms of performance, we do not consider it further.

5.4. Scalability with threads

Concurrent performance is an obvious concern for memory reclamation schemes. We study the effect of threads sharing

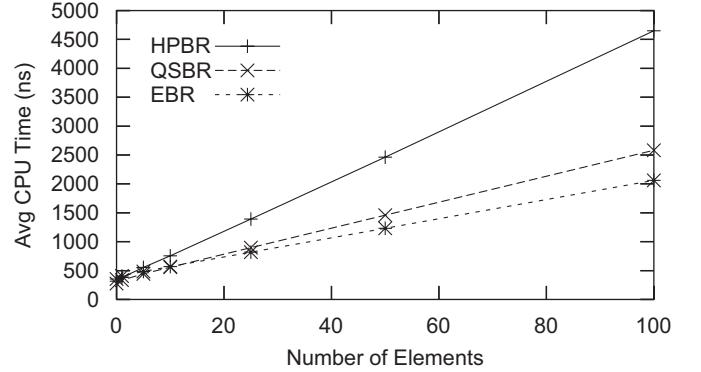


Fig. 8. Effect of traversal length—read-only lock-free list, one thread, 8 CPUs.

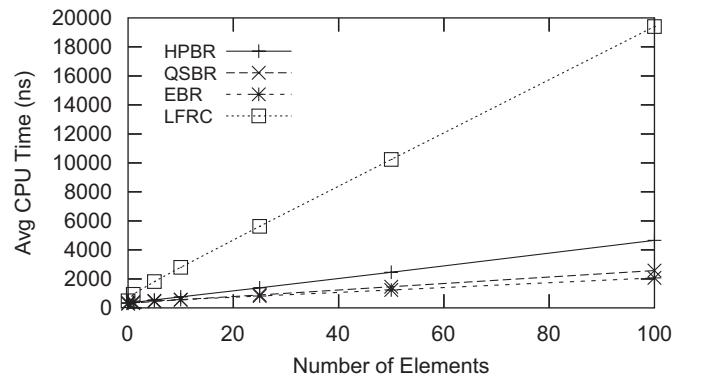


Fig. 9. Effect of traversal length, including LFRC—read-only lock-free list, one thread, 8 CPUs.

the data structure when there is no CPU contention, and when threads must also compete for the CPU.

5.4.1. No preemption

When we attempted to run eight non-real-time processes simultaneously on our 8-CPU machine, we experienced the effects of preemption, which we examine in the next subsection. To reduce the risk of preemption and the other effects of CPU contention, such as thread migration, we use a maximum of seven threads, ensuring that one CPU is available for other processes, following Fraser [9]. We could have instead prevented preemption by running our threads with real-time priority. We ran limited tests to confirm that doing so prevents preemption and its associated effects on performance; however, since the scheduler could have other effects on performance, we chose to simply keep a CPU free instead.

Figs. 10 and 11 show the performance of the reclamation schemes with a read-only workload on a linked list, and with an update-only workload on a queue, respectively. All three schemes scale almost linearly in the read-only case. In the update-only case shown in Fig. 11, we see increased overhead in all cases because multiple threads are trying to update a single queue; since hazard pointers have to be re-set when a thread restarts an operation, HPBR becomes slightly worse when contention is high. Aside from this minor increase, the schemes'

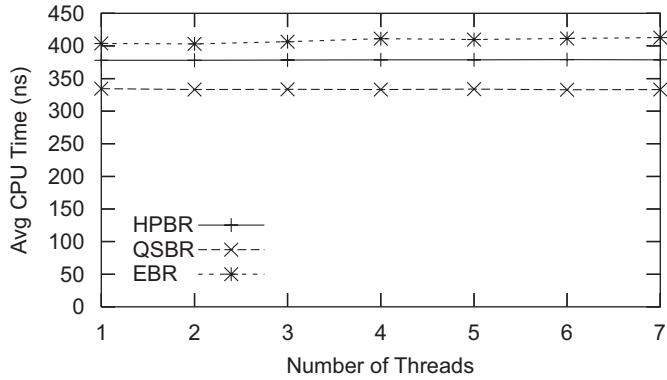


Fig. 10. Effect of adding threads—read-only lock-free list, one element, 8 CPUs.

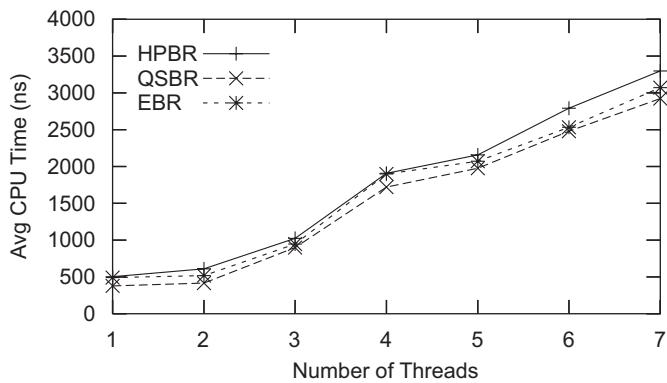


Fig. 11. Effect of adding threads—lock-free queue, 8 CPUs.

relative performance is largely unaffected by the number of threads, regardless of whether the workload is read-only or update-only.

5.4.2. With preemption

To evaluate the performance of the reclamation schemes under preemption, we ran our tests on our 2-CPU machine, varying the number of threads from 1 to 32.

Fig. 12 shows the performance of the schemes on a one-element lock-free linked list with a read-only workload. This case eliminates reclamation overhead, focusing solely on read-side and fuzzy barrier overhead. In this case, the algorithms all scale well, with QSBR remaining the most efficient.

For the update-heavy workloads, such as the update-only queue shown in Fig. 13, HPBR performs best. Preempted threads slow down threads using QSBR or EBR, since their failure to go through quiescent states in a timely manner will result in memory exhaustion, leading to allocation failures. We note that HPBR performs best even when the number of threads is equal to the number of CPUs—this is because, when the number of threads and CPUs are equal, other system threads may preempt our threads, as discussed in the previous subsection.

In the above experiments, threads using QSBR or EBR yield the processor on allocation failure using `sched_yield()`,

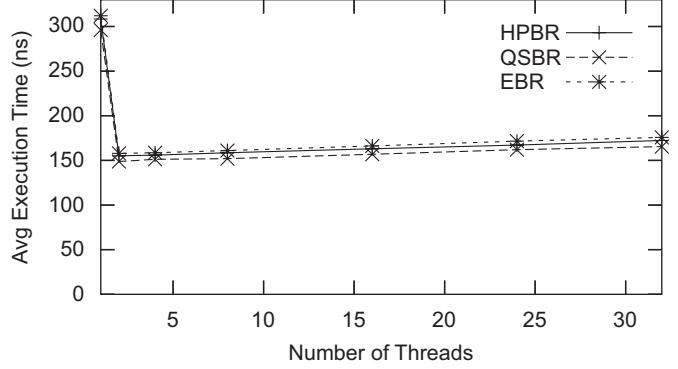


Fig. 12. Effect of preemption—read-only lock-free list, 2 CPUs.

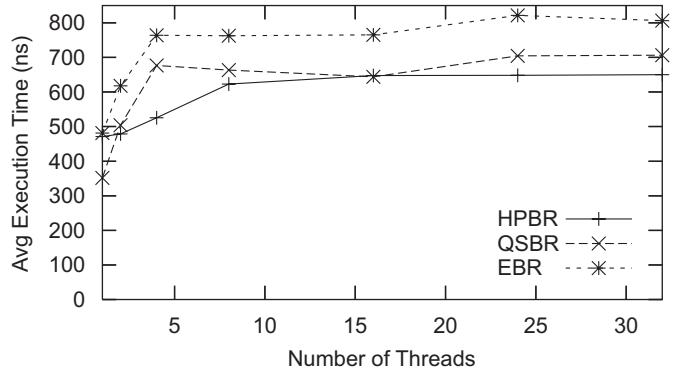


Fig. 13. Effect of preemption—lock-free queue, 2 CPUs.

since preempted threads must run in order for grace periods to occur and thus for memory to be reclaimed. Yielding the processor upon allocation failure is a necessary condition for QSBR and EBR to have acceptable performance under an update-heavy workload with preemption. Fig. 14 shows the same test as Fig. 13, but with busy-waiting upon allocation failure. Here, HPBR performs well, but EBR and QSBR quickly exhaust the pool of free memory. Each thread spins waiting for more memory to become free, thereby further preventing grace periods from completing in a timely manner and hence delaying memory reclamation.

Although busy waiting on allocation failure would be a poor design choice in an application using grace periods for memory reclamation, this test demonstrates that preemption and update-heavy workloads can cause QSBR and EBR to exhaust all memory. In situations in which grace periods are not achieved in a timely manner, HPBR's bounds on unfreed memory become valuable.

5.5. Summary

The performance of the different memory reclamation schemes is often comparable, as in Figs. 11 and 12; however, in degenerate cases, reclamation overhead can dominate execution time, as in Figs. 9 and 14. Programmers must

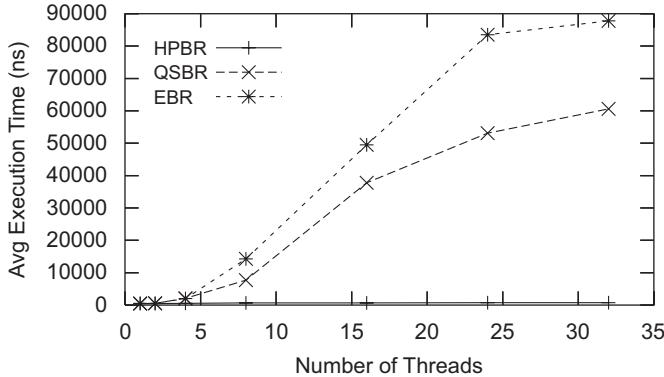


Fig. 14. Effect of busy waiting when out of memory—lock-free queue, 2 CPUs.

therefore understand the tradeoffs between the schemes in order to ensure good performance.

In the base case, atomic instructions such as fences are the dominant cost. The overhead due to factors such as entering and exiting a fuzzy barrier or scanning an array of hazard pointers is minor, and only affects the performance noticeably in the case of the queue results in Fig. 6.

HPBR and LFRC require per-element atomic instructions. Since these atomic instructions are expensive, HPBR and LFRC perform poorly when long chains of elements must be traversed. In the case of LFRC, the need for atomic increments is inherent, and in the case of HPBR, the need for fences is a consequence of the weakly consistent memory models on modern processors.

QSBR and EBR depend on grace periods occurring sufficiently often. If grace periods are stalled—for example, due to preemption—and the workload is update-heavy, these schemes may exhaust memory. In our experience, the impact of this problem can be reduced by yielding the processor on allocation failure; experience with the Linux kernel also suggests that this problem can be mitigated in practice [34].

6. Consequences and discussion

We describe the consequences of our analysis for comparing algorithms, designing new reclamation schemes, and choosing reclamation schemes for applications. We also discuss factors other than performance which affect the choice of memory reclamation scheme. Finally, we present system-level performance results obtained from the Linux kernel.

6.1. Fair evaluation of algorithms

Reclamation schemes have profound performance effects that must be accounted for when experimentally evaluating new lockless algorithms.

Fig. 15 shows one of our early, faulty experiments, performed prior to the evaluation in Section 5; it plots CPU time against update fraction for a 10-element list. The goal of this experiment was to compare the performance of a lock-free linked list with HPBR (LF-HPBR) with a concurrently readable

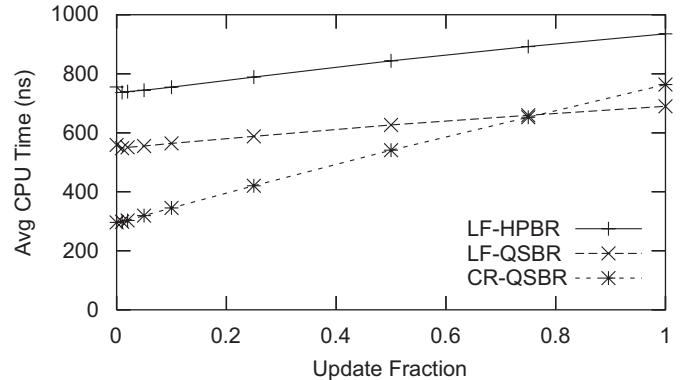


Fig. 15. Lock-free (LF) versus concurrently readable (CR) algorithms—10-element lists, one thread, 8 CPUs.

equivalent using QSBR (CR-QSBR). Concurrently readable algorithms have lockless reads but use locks for updates; such algorithms are used with QSBR in the Linux kernel. The motivation for these pairings was to compare a combination with strong fault-tolerance guarantees—a thread cannot hold a lock indefinitely or stop other threads from reclaiming memory—with a combination lacking these guarantees. Our intuition was that lock-free algorithms might pay a performance penalty for these fault-tolerance properties.

We initially performed this experiment with only the LF-HPBR and CR-QSBR traces shown in Fig. 15. Since these traces never cross, our original experiment led us to the erroneous conclusion that the concurrently readable algorithm is always faster. A better analysis also performs the experiment with LF-QSBR, noting that as the update fraction increases, lock-free performance improves relative to the concurrently readable approach, since its updates require fewer atomic instructions than does locking. For update fractions above roughly 75% LF-QSBR achieves the best performance. The large gap between the LF-HPBR and CR-QSBR traces is not due to the difference between lock-free and concurrently readable linked lists, but the fact that HPBR requires per-element fences, and this list has 10 elements. This example shows that one can accurately compare two lockless algorithms *only* when each is using the same reclamation scheme.

The lock-free linked list performs some per-element checks which the concurrently readable list does not, independent of which reclamation scheme is used. Since the list in Fig. 15 has 10 elements, these checks impose significant overhead. The lock-free linked list is thus more appealing when fewer elements must be traversed. Fig. 16 shows the performance of hash tables being concurrently accessed by four threads. The hash table consists of arrays of LF-QSBR or CR-QSBR single-element lists, using the same list algorithms as in Fig. 15. For clarity, we omit HPBR from this graph—our intent is to compare the lock-free and concurrently readable algorithms using a common reclamation scheme. Here, since there are fewer elements on which to perform checks, the lock-free algorithm that out-performs the concurrently readable alternative for update fractions above about 20%. Lock-free lists and

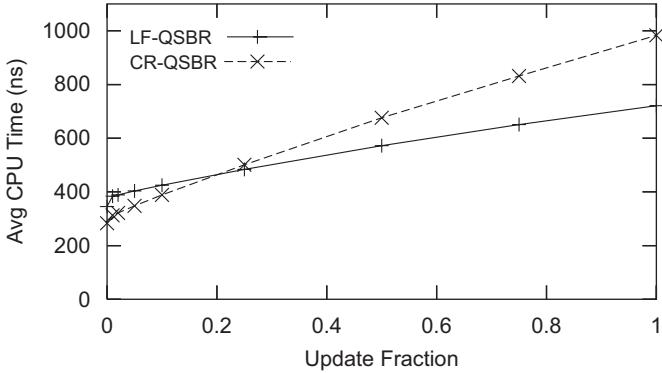


Fig. 16. Lock-free (LF) versus concurrently readable (CR) algorithms—hash tables with load factor 1, four threads, 8 CPUs.

hash tables might therefore be practical for update-heavy situations in environments providing QSBR, such as OS kernels like Linux.

New reclamation schemes should also be evaluated by varying each of the factors that can affect their performance. For example, Gidenstam et al. [11] proposed a new non-blocking reclamation scheme that combines reference counting with HPBR, and can be proven to have several attractive properties. However, like HPBR and reference counting, it requires expensive per-element atomic operations. The evaluation of this scheme consisted only of experiments on double-ended queues, thus failing to evaluate scalability with data-structure size, a weakness of HPBR. This failing shows the value of our analysis: it is necessary to vary the experimental parameters we considered to gain a full understanding of a given scheme's performance.

6.2. Improving reclamation performance

Improved reclamation schemes can be designed based on an understanding of the factors that affect the performance. For example, we observe that a key difference between QSBR and EBR is the per-operation overhead of EBR's two fences which it requires when entering and leaving a critical region. This observation allows us to make a modest improvement to EBR called NEBR.

NEBR requires compromising EBR's application-independence. Instead of setting and clearing a flag at the start and end of every lockless operation, we set it at the application level before entering any code that might contain NEBR critical regions. Since our flag is set and cleared at the application level, we can amortize the overhead of the corresponding fence instructions over a larger number of operations. We reran the experiment shown in Fig. 10, but including NEBR, and, as shown in Fig. 17, found that NEBR scaled linearly and performed slightly better than did HPBR.

Furthermore, NEBR does not need the expensive per-element atomic operations that ruin HPBR's performance for long traversals. NEBR's overhead relative to QSBR can be attributed to its need to attempt to check and update epochs when starting a lockless operation.

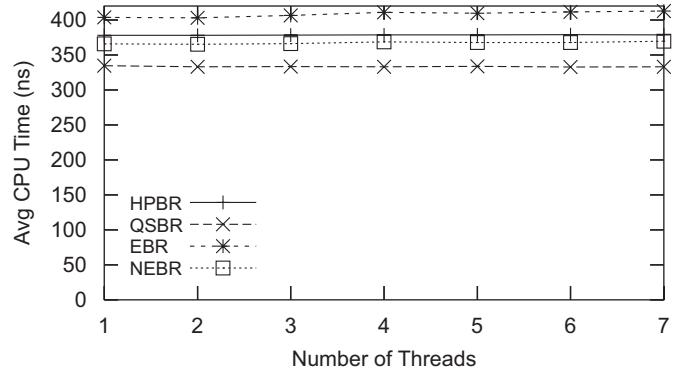


Fig. 17. Performance of NEBR—lock-free list, 8 CPUs, read-only workload, variable number of threads.

NEBR is attractive because its overhead is close to that of QSBR, but it simplifies the job of an application programmer. With QSBR, a programmer must identify an appropriate set of quiescent states in the application code, and mark them. With NEBR, the programmer's job is simplified to marking only the beginning and end of regions of the application code in which lockless calls are made; updating the global epoch is still periodically attempted after some fixed number of lockless operations. Interestingly, recent real-time variants of the Linux-kernel RCU also dispense with quiescent states [27]. Ongoing work is expected to substantially reduce real-time RCU read-side overhead.

There are many opportunities for future work in the area of memory reclamation. Our recommendation that threads using QSBR or EBR yield the CPU on allocation failure is similar in spirit to the idea of scheduler-conscious synchronization [21]. It may be possible to further apply the ideas of scheduler-conscious synchronization to create a user-level QSBR implementation with less overhead or a lower memory footprint than the one used in our experiments.

HPBR requires per-element fences on any machine with a weakly consistent memory model. A scheme which avoids per-element atomic instructions yet still makes some guarantees about memory usage would be a desirable technique. Creating such a scheme, and formalizing what progress guarantees it makes, is an interesting challenge.

We note that it is possible to combine aspects of different reclamation schemes. Seigh [35] has proposed a scheme which combines HPBR with QSBR in an attempt to get good flexibility and performance.

6.3. Blocking memory reclamation for non-blocking data structures

Schemes relying on grace periods are blocking in the sense that if grace periods do not happen, the system will run out of memory, and threads will fail to make progress. We have shown that non-blocking data structures often perform better when using these blocking schemes than they do with HPBR, which is non-blocking. One might question why one would want to use a non-blocking data structure in this case, since a halted thread would cause an infinite memory leak, thus

destroying the non-blocking data structure's fault-tolerance guarantees.

However, non-blocking data structures are often used for reasons other than fault-tolerance; for example, Qprof [5] and Cache Kernel [13] both use such structures because they can be accessed from signal handlers without risk of self-deadlock. Blocking memory reclamation does not remove this benefit. In fact, Cache Kernel uses a blocking implementation of type-stable memory to guard against read-reclamation races; its implementation [12] has similarities to QSBR. Non-blocking algorithms with blocking reclamation schemes similarly continue to benefit from resistance to preemption-induced convoying and priority inversion.

We view combining a non-blocking algorithm with a blocking reclamation scheme as part of a trend toward weakened non-blocking properties [5,19], designed to preserve selected advantages of non-blocking synchronization while improving performance. In this case, threads have all the advantages of non-blocking synchronization, *unless* the system runs out of memory.

Conversely, one may also use non-blocking reclamation with blocking algorithms to reduce the amount of memory awaiting reclamation in the face of preempted or failed threads.

6.4. Non-performance factors

Although we have analyzed memory reclamation schemes purely from a performance standpoint, performance is not the only factor to consider when choosing a memory reclamation scheme. The schemes have slightly different semantics which may make one scheme or another more attractive for a particular application.

Reference counting, for its part, has well-known difficulties in dealing with cyclic garbage [11]. However, it has the advantage that elements can be accessed after having been removed from a shared data structure; this feature was exploited by Sundell and Tsigas [36,38] to let threads accessing a doubly linked list avoid restarting their traversals when elements are removed.

Identifying appropriate quiescent states in an application may not be straightforward [30]; furthermore, choosing inappropriate quiescent states may have unforeseen consequences. Sarma and McKenney [34] discussed how the use of QSBR in the Linux kernel's IPv4 route cache introduced the possibility of denial-of-service (DoS) attacks. By sending a large number of packets to a machine, an attacker could force the victim to spend all its time in interrupt handlers, which had no quiescent states in older versions of Linux. The attacker could thus indefinitely extend grace periods, resulting in unbounded memory consumption that would eventually cause the system to hang. Enhancements to Linux's QSBR infrastructure now provide a new set of quiescent states for use in interrupt handlers that can prevent such attacks from indefinitely extending grace periods. It is an open question, however, whether Linux's QSBR infrastructure can be formally proven to be robust in the face of arbitrary DoS attacks.

As noted in Section 2.3, some algorithms may require an unbounded number of hazard pointers; this requirement stymied

an attempt to use hazard pointers in Linux to improve real-time response [27]. The potentially unbounded number of hazard pointers necessitates dynamic hazard pointer allocation [30], requiring that the HPBR reference-acquisition primitive be able to either block or return failure in low-memory situations. In certain OS kernel situations neither blocking nor failing is an acceptable response. For example, when physical memory is exhausted the OS will write pages to external swap space in order to free memory. Use of a deferred-destruction scheme based on HPBR in this case can lead to memory deadlock because memory is exhausted and hazard pointer allocation will likely fail. In general, allowing hazard pointer allocation to block would prohibit the use of this scheme in OS kernel situations where blocking is not acceptable (such as when interrupts are disabled). The alternative solution of introducing a failure return introduces new software engineering difficulties. Although these problems can be dealt with in principle, they may be prohibitively difficult to handle in practice in a large and complex code base like an OS kernel.

In addition, the not-uncommon case of aggregated data structures (for example, nested C-language structs) can pose software-engineering obstacles for the use of hazard pointers. In such cases, constraints must be imposed on memory allocation, such that a pointer to any component of the aggregate will be deemed equivalent to an alias, that is to say, a pointer to some other component of that same aggregate.

Finally, achieving the potential memory-footprint advantages of hazard pointers depends on the use of a fixed, finite number of hazard pointers per thread. In this case, hazard pointers do not need to be explicitly released, since they will be reused in a relatively short time period. If hazard pointers are dynamically allocated, however, then they must also be explicitly released to provide bounds on the amount of unreclaimed memory.

It might still be advantageous to use HPBR in operating system kernels like Linux, but only if its use is restricted to code that (1) is not used when freeing up memory, thus avoiding out-of-memory deadlocks, (2) requires a strictly bounded number of hazard pointers, and (3) avoids the pointer-aliasing problems posed by aggregated data structures. These limitations, however, make HPBR unsuitable for use as a memory reclamation scheme when implementing the current Linux RCU API. In the following section, we take a closer look at this API and several of its uses in the Linux kernel.

7. Example uses: the Linux RCU API

The Linux RCU API provides a set of operations that allow lockless concurrent reads of shared data structures and deferred destruction of elements removed from these structures. Writers may not prevent readers from accessing the shared data, but must coordinate with each other in some way. Typically, traditional spinlocks are used to prevent concurrent updates, however, other methods could be used as well; the RCU API does not dictate how updates should be coordinated. In this section, we review the Linux RCU API and its use of memory, and then consider a specific example where RCU is used in the Linux kernel.

7.1. Requirements and memory reclamation

The RCU API was designed for use in OS kernels; it is defined to neither block nor fail for readers, and cannot feasibly be altered to do so. As noted in Section 6.4, allowing blocking would prohibit the use of RCU in many situations where it has proven valuable (such as when interrupts are disabled). Introducing a failure return instead would be extremely difficult in many of the 244 uses of this primitive in the Linux 2.6.20 kernel. As a result, RCU uses QSBR for memory reclamation.

The RCU system tracks quiescent states and grace periods. In the original version, designed for non-preemptable kernels, context switches are used to identify quiescent states. To allow preemptable kernels the API requires read-side critical regions to be identified with calls to `rcu_read_lock` and `rcu_read_unlock`. These calls disable and enable preemption to guarantee that context switches (and thus quiescent states) do not occur while a thread is in the middle of accessing an RCU-protected data structure. The RCU API has been further extended with a new set of quiescent states for use in interrupt handlers, motivated by the need to prevent DoS attacks, as mentioned in Section 6.4. Finally, other extensions have targeted real-time response.

The requirements on the RCU API in an OS kernel make QSBR a natural choice for memory reclamation. Preemption can be suppressed in a kernel environment, so delayed grace periods are unlikely. In addition, uses of RCU in Linux have targeted read-mostly data structures that are rarely updated in common uses. Since updates are rare, memory reclamation is also rare and the most important performance consideration is the overhead required for reads. Our microbenchmark results show that QSBR has the lowest per-operation overhead for read-only workloads (see Fig. 6), and we would therefore expect it to have the best performance for these real uses as well. Furthermore, since QSBR does not have expensive per-element atomic instructions, it is well fitted to the concurrently readable lists which are common in Linux.

Although EBR could also be used in Linux, it has no performance advantage over QSBR; any usability advantages are less relevant given that an implementation with QSBR already exists. Attempts to use HPBR require dynamic hazard pointer allocation, leading to the difficulties of failing or blocking noted earlier. In addition, the RCU API releases references implicitly at the end of the corresponding RCU read-side critical region, rather than at the point where the reference is no longer needed. Therefore, an HPBR-based RCU implementation that traverses an arbitrary-length list in a single RCU read-side critical section could consume an arbitrarily large number of hazard pointers. On the other hand, modifying Linux's RCU API to include explicit release of RCU references would require prohibitively large and pervasive changes to Linux [27].

7.2. System V IPC

We now focus on one specific use of RCU in the Linux kernel—namely, System V IPC. Since alternate implementa-

tions using different memory reclamation schemes do not exist, we cannot compare their performance in these cases. Our performance comparisons are thus between traditional locking and RCU. These results show that lockless approaches are valuable and can out-perform lock-based approaches by a considerable margin, and that blocking reclamation schemes can be practical in large applications.

It would be desirable to make all such comparisons using a contemporary version of the Linux kernel; however, where RCU has proved valuable it is difficult and largely pointless to re-engineer the kernel to re-introduce locking again. An alternative approach would map the RCU uses in the Linux kernel to conventional locking; however, such a mapping is in general problematic. Some of the difficulties include:

- (1) RCU read-side critical sections may be entered unconditionally in any software environment within the kernel, including even the non-maskable interrupt handlers that result in deadlocks when locking is used.
- (2) RCU read-side critical sections may include update code. Attempts to map this usage into reader–writer locking again result in deadlocks.

Such difficulties are in fact common in the Linux kernel, as was learned in a failed attempt to replace the RCU API with locking in order to improve real-time response [25].

Given the impracticality of mapping RCU onto conventional locking, we instead present the performance comparisons that were made on server-class machines when proposing concurrently readable algorithms with RCU for acceptance into the Linux kernel. We refer to this use of RCU, where updates use locking, as CR-QSBR. Specifically, we analyze the use of CR-QSBR in the implementation of the System V IPC subsystem in the Linux kernel, showing the system- and application-level performance implications. Further system-level examples are discussed in detail by McKenney [24].

The System V IPC subsystem implements System V semaphores, message queues, and shared memory. Applications access these resources using an integer ID, and the Linux kernel uses an array to map from this ID to in-kernel data structures that represent the corresponding resource. The array is expandable, and prior to the conversion to use CR-QSBR, was protected by a spinlock. The array is frequently accessed read-only when System V IPC objects are used and infrequently accessed for writing when objects are created or deleted or the array is resized. Because each element of the array is a single aligned pointer, object creation and deletion events may be done in place, hence the array need only be copied for expansions. After conversion to use CR-QSBR, all writes continue to use the original spinlock, while the more common read operations simply flag entry and exit from critical regions using the `rcu_read_lock` and `rcu_read_unlock` functions.

Two experiments were used to compare the performance of the Linux 2.5.42-mm2 kernel, with and without CR-QSBR. The first experiment used a System V semaphore user-level benchmark on an 8-CPU 700 MHz Intel PIII system. In this benchmark, multiple user-level processes each repeatedly acquire and release different semaphores in parallel, with the benchmark

Table 3
Semopbench application-level results (seconds)

Kernel	Run 1	Run 2	Avg.
2.5.42-mm2	515.1	515.4	515.3
2.5.42-mm2+ipc-rcu	46.7	46.7	46.7

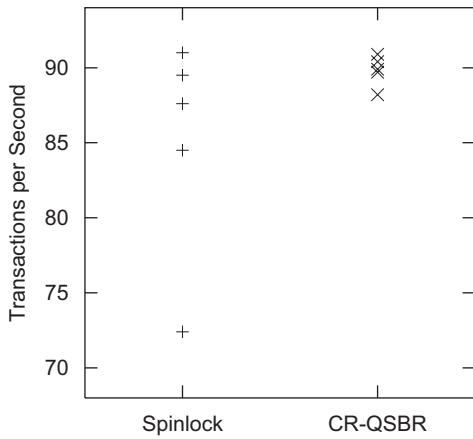


Fig. 18. DBT1 database benchmark raw results.

Table 4
DBT1 database benchmark results (TPS)

Kernel	Average	Standard deviation
2.5.42-mm2	85.0	7.5
2.5.42-mm2+ipc-rcu	89.8	1.0

metric being the length of time for each process to complete a fixed number of such operations. The second experiment used the DBT1 [33] database-webserver benchmark on an Intel dual-CPU 900 MHz PIII with 256 MB of memory. The results of the first experiment are shown in Table 3 and illustrate an order-of-magnitude increase in the performance for this user-level benchmark. The raw results for the second experiment are presented in Fig. 18, with a summary presented in Table 4. The results show that the RCU-based kernel performs over 5% better in terms of transactions per second (TPS) than does the stock kernel. More importantly, the results for the RCU-based kernel are much more stable; the erratic results for the stock kernel are not unusual for workloads with lock contention.

8. Related work

Relevant work on reclamation scheme design was discussed in Section 2. Previous work on the performance of these schemes, however, is limited. Michael [30] criticized QSBR for its unbounded memory use, but did not compare the performance of QSBR to that of HPBR, or determine when this limitation can affect a program. Fraser [9] noted, but did not thoroughly evaluate, HPBR's fence overhead and used his EBR instead. Our work extends Fraser's, showing that EBR itself has high overhead, often exceeding that of HPBR.

Auslander implemented a lock-free hash table with QSBR in K42 [24]. No performance evaluation, either between different reclamation methods or between concurrently readable and lock-free hash tables, was provided. We are unaware of any work combining QSBR with update-intensive non-blocking algorithms such as queues.

McKenney [24] details many uses of QSBR in the Linux and K42 OS kernels. We discuss one use of QSBR—namely, System V IPC—but the focus of our work is on the comparative performance of memory reclamation schemes.

9. Conclusions

We have performed the first fair comparison of blocking and non-blocking reclamation across a range of workloads, showing that reclamation has a huge effect on lockless algorithm performance. Choosing the right scheme for the environment in which a concurrent algorithm is expected to run is essential to having the algorithm perform well.

Our results, starting with Fig. 6, show that quiescent-state-based reclamation (QSBR) is usually the best-performing reclamation scheme; however, the performance of both QSBR and epoch-based reclamation (EBR) can suffer due to memory exhaustion in the face of thread preemption or failure. Hazard-pointer-based reclamation (HPBR) and EBR have higher base costs than QSBR due to their required fences; for EBR, the worst-case overhead of fences is constant, while for HPBR it is unbounded. Lock-free reference counting (LFRC) has even higher overhead due to the per-element atomic instructions it requires. HPBR and LFRC both scale poorly when many elements must be traversed.

Our analysis helped us to identify the main source of overhead in EBR and decrease it, resulting in our new epoch-based reclamation (NEBR) scheme. Furthermore, understanding the impact of reclamation schemes on algorithm performance enables fair comparison of different algorithms—in our case, lock-free and concurrently readable lists and hash tables.

The results of our analysis indicate that QSBR is, in fact, the scheme best suited to an OS kernel environment. Our performance data from the Linux kernel shows that lockless approaches using QSBR are practical and can outperform locking approaches by a large margin.

We reiterate that blocking reclamation can be useful with non-blocking algorithms: in the absence of thread failure, non-blocking algorithms still benefit from deadlock-freedom, signal handler safety, and avoidance of priority inversion. Nevertheless, one important question remains open, namely, what sort of weakened non-blocking property could be satisfied by a reclamation scheme that avoids the per-element overhead that is incurred by all currently known non-blocking reclamation schemes.

Legal statement

IBM and POWER are trademarks or registered trademarks of International Business Machines Corporation in the United States and/or other countries.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.

Acknowledgments

We owe thanks to Maged Michael and Keir Fraser for helpful comments on their respective work, to Faith Fich and Cristiana Amza for much helpful advice, and to Dan Frye for his support of this effort. We are indebted to Martin Bligh, Andy Whitcroft, and the ABAT team for access to the 8-CPU machine used in our experiments. Finally, we wish to thank our colleagues at the University of Toronto who suggested several clarifications.

References

- [1] S.V. Adve, K. Gharachorloo, Shared memory consistency models: a tutorial, *IEEE Comput.* 29 (12) (1996) 66–76.
- [2] T.E. Anderson, The performance of spin lock alternatives for shared-memory multiprocessors, *IEEE Trans. Parallel Distrib. Syst.* 1 (1) (1990) 6–16.
- [3] A. Arcangeli, M. Cao, P.E. McKenney, D. Sarma, Using read-copy update techniques for System V IPC in the Linux 2.5 kernel, in: Proceedings of the 2003 USENIX Annual Technical Conference (FREENIX Track), USENIX Association, 2003.
- [4] E. Berger, K. McKinley, R. Blumofe, P. Wilson, Hoard: a scalable memory allocator for multithreaded applications, in: Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems, Cambridge, MA, 2000.
- [5] H.-J. Boehm, An almost non-blocking stack, in: Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing, ACM Press, New York, NY, USA, 2004.
- [6] J. Bonwick, J. Adams, Magazines and Vmem: extending the slab allocator to many CPUs and arbitrary resources, in: USENIX Technical Conference, General Track, 2001.
- [7] T.D. Chandra, S. Toueg, Unreliable failure detectors for reliable distributed systems, *J. ACM* 43 (2) (1996) 225–267.
- [8] D.L. Detlefs, P.A. Martin, M. Moir, G.L. Steele Jr., Lock-free reference counting, *Distrib. Comput.* 15 (4) (2002) 255–271.
- [9] K. Fraser, Practical lock-freedom, Ph.D. Thesis, University of Cambridge Computer Laboratory, 2004.
- [10] B. Gamsa, O. Krieger, J. Appavoo, M. Stumm, Tornado: maximizing locality and concurrency in a shared memory multiprocessor operating system, in: Proceedings of the Third Symposium on Operating Systems Design and Implementation, USENIX Association, Berkeley, CA, USA, 1999.
- [11] A. Gidenstam, M. Papatriantafilou, H. Sundell, P. Tsigas, Efficient and reliable lock-free memory reclamation based on reference counting, in: Proceedings of the Eighth International Symposium on Parallel Architectures, Algorithms and Networks, IEEE Computer Society, Washington, DC, USA, 2005.
- [12] M. Greenwald, Non-blocking synchronization and system design, Ph.D. Thesis, Stanford University, 1999.
- [13] M. Greenwald, D. Cheriton, The synergy between non-blocking synchronization and operating system structure, in: Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation, ACM Press, 1996.
- [14] R. Gupta, The fuzzy barrier: a mechanism for high speed synchronization of processors, in: Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems, ACM Press, New York, NY, USA, 1989.
- [15] T.L. Harris, A pragmatic implementation of non-blocking linked-lists, in: Proceedings of the 15th International Conference on Distributed Computing, Springer, Berlin, 2001.
- [16] M. Herlihy, Wait-free synchronization, *ACM Trans. Prog. Lang. Syst.* 13 (1) (1991) 124–149.
- [17] M. Herlihy, A methodology for implementing highly concurrent data objects, *ACM Trans. Prog. Lang. Meth.* 15 (5) (1993) 745–770.
- [18] M. Herlihy, V. Luchangco, M. Moir, The repeat offender problem: a mechanism for supporting dynamic-sized, lock-free data structures, in: Proceedings of the 16th International Symposium on Distributed Computing, 2002.
- [19] M. Herlihy, V. Luchangco, M. Moir, Obstruction-free synchronization: double-ended queues as an example, in: Proceedings of the 23rd International Conference on Distributed Computing Systems, IEEE Computer Society, 2003.
- [20] IBM, IBM System/370 Extended Architecture, Principles of Operation, No. SA22-7085, 1983.
- [21] L.I. Kontothanassis, R.W. Wisniewski, M.L. Scott, Scheduler-conscious synchronization, *ACM Trans. Comput. Syst.* 15 (1) (1997) 3–40.
- [22] S. Kumar, D. Jiang, R. Chandra, J.P. Singh, Evaluating synchronization on shared address space multiprocessors: methodology and performance, in: Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, ACM Press, New York, NY, USA, 1999.
- [23] L. Lamport, How to make a multiprocessor computer that correctly executes multiprocess programs, *IEEE Trans. Comput.* 28 (9) (1979) 690–691.
- [24] P.E. McKenney, Exploiting deferred destruction: an analysis of read-copy-update techniques in operating system kernels, Ph.D. Thesis, OGI School of Science and Engineering at Oregon Health and Sciences University, 2004.
- [25] P.E. McKenney, Real-time preemption and RCU, available: <http://lkml.org/lkml/2005/3/17/199> March 2005, (Viewed September 5, 2005).
- [26] P.E. McKenney, J. Appavoo, A. Kleen, O. Krieger, R. Russell, D. Sarma, M. Soni, Read-copy update, in: Proceedings of the 2001 Ottawa Linux Symposium, 2001.
- [27] P.E. McKenney, D. Sarma, Towards hard realtime response from the Linux kernel on SMP hardware, in: linux.conf.au, Canberra, AU, 2005.
- [28] P.E. McKenney, J.D. Slingwine, Read-copy update: using execution history to solve concurrency problems, in: Proceedings of the 1998 International Conference on Parallel and Distributed Computing and Systems, Las Vegas, NV, 1998.
- [29] M.M. Michael, Safe memory reclamation for dynamic lock-free objects using atomic reads and writes, in: Proceedings of the 21st ACM Symposium on Principles of Distributed Computing, 2002.
- [30] M.M. Michael, Hazard pointers: safe memory reclamation for lock-free objects, *IEEE Trans. Parallel Distrib. Syst.* 15 (6) (2004) 491–504.
- [31] M.M. Michael, Scalable lock-free dynamic memory allocation, in: Proceedings of the ACM Conference on Programming Language Design and Implementation, 2004.
- [32] M.M. Michael, M.L. Scott, Correction of a memory management method for lock-free data structures, Technical Report TR599, Computer Science Department, University of Rochester, December 1995.
- [33] Open Source Development Labs, Inc., Database test suite, available: http://www.osdl.org/lab_activities/kernel_testing/osdl_database_test_suite/ 2003, (Viewed June 29, 2005).
- [34] D. Sarma, P.E. McKenney, Issues with selected scalability features of the 2.6 kernel, in: Proceedings of the 2004 Ottawa Linux Symposium, 2004.
- [35] J. Seigh, RCU + SMR for preemptive kernel/user threads, linux-kernel mailing list, available: <http://lkml.org/lkml/2005/5/9/129>, 2005.
- [36] H. Sundell, Efficient and practical non-blocking data structures, Ph.D. Thesis, Chalmers University of Technology, 2004.
- [37] H. Sundell, Wait-free reference counting and memory management, in: Proceedings of the 19th International Parallel and Distributed Processing Symposium, 2005.

- [38] H. Sundell, P. Tsigas, Lock-free and practical doubly linked list-based deques using single-word compare-and-swap, in: Proceedings of the Eighth International Conference on Principles of Distributed Systems, 2004.
- [39] J.D. Valois, Lock-free linked lists using compare-and-swap, in: Proceedings of the 14th ACM Symposium on Principles of Distributed Computing, 1995.



Thomas E. Hart is a Ph.D. student in the Department of Computer Science at the University of Toronto. His research interests are in computer security, automated verification, and synchronization. He holds a B.Sc. in Mathematics and Computer Science from Brandon University (2003) and an M.Sc. in Computer Science from the University of Toronto (2005). He is the holder of an NSERC Canada Graduate Scholarship.



Paul E. McKenney is a Distinguished Engineer at IBM's Linux Technology Center, where he works on parallel real-time synchronization primitives in the Linux kernel. Prior to that, he worked on a parallel UNIX database-server operating system at Sequent Computer Systems, and on Internet and packet-radio congestion-control protocols at SRI International. He holds more than 20 patents and has published more than 30 papers. He holds B.Sc. degrees in Computer Science and in Mechanical Engineering and an M.Sc. degree from Oregon State University (1981 and 1988), and a Ph.D. from Oregon Health and Sciences University (2004).



Angela Demke Brown is an Assistant Professor in the Department of Computer Science at the University of Toronto. Her research interests include operating systems, dynamic compilers, and interpreters. Her focus is on delivering the underlying performance of emerging computer architectures to user applications through the interaction of language tools and operating system services. She holds M.Sc. and Ph.D. degrees in Computer Science from the University of Toronto (1997) and Carnegie Mellon University (2005), respectively.



Jonathan Walpole is a Full Professor in the Computer Science Department at Portland State University. His research interests are in operating systems, distributed systems, and networking. His current research focuses on scalable synchronization mechanisms for shared memory multiprocessor systems. He has published over 100 research papers and has served as a program committee member and reviewer for numerous International scientific conferences and journals. He holds B.Sc. and Ph.D. degrees in Computer Science from Lancaster University, UK (1984 and 1987) and was awarded a Post-Doctoral Fellowship by the UK Science and Engineering Research Council in 1988.

Lock-free programming is a challenge, not just because of the complexity of the task itself, but because of how difficult it can be to penetrate the subject in the first place.

--preshing.com



This is a revised version of the previously published paper. It includes a contribution from Shahar Frank who raised a problem with the *fifo-pop* algorithm.
Revised version date: sept. 30 2003.

Lock-Free Techniques for Concurrent Access to Shared Objects

Dominique Fober

Yann Orlarey

Stephane Letz

Grame - Centre National de Création Musicale

9, rue du Garet BP 1185

69202 LYON CEDEX 01

Tél +33 (0)4 720 737 00 Fax +33 (0)4 720 737 01

[fober, orlarey, letz]@grame.fr

Abstract

Concurrent access to shared data in preemptive multi-tasks environment and in multi-processors architecture have been subject of many works. Proposed solutions are commonly based on semaphores which have several drawbacks. For many cases, lock-free techniques constitute an alternate solution and avoid the disadvantages of semaphore based techniques. We present the principle of these lock-free techniques with the simple example of a LIFO stack. Then, based on Michael-Scott previous work, we propose a new algorithm to implements lock-free FIFO stacks with a simple constraint on the data structure.

1. Introduction

A shared data structure is lock-free if its operations do not require mutual exclusion: if a process is interrupted in the middle of an operation, it will not prevent the other processes from operating on that object. Lock-free techniques avoid common problems associated with conventional locking techniques:

- priority inversion: occurs when a high-priority process requires a lock held by a lower-priority process,
- convoying: occurs when a process holding a lock is descheduled by exhausting its quantum, by a page fault or by some other kind of interrupt. In this case, running processes requiring the lock are unable to progress.
- deadlock: can occur if different processes attempt to lock the same set of objects in different orders.

In particular locking techniques are not suitable in a real-time context and more generally, they suffer significant performance degradation on multiprocessors systems.

A lot of works have investigated lock-free concurrent data structures implementations [1, 2, 3, 4]. Advantages and limits of these works are discussed in [5]. We propose a new lock-free FIFO queue algorithm. It has been initially designed to be part of a multi-tasks, real-time MIDI operating system [6] in order to support an efficient inter-applications communication mechanism. Its implementation is based on Michael-Scott [4] but removes the necessary node allocation when enqueueing a value, by introducing a simple constraint on the value data type structure.

The rest of this paper is organized as follow: section 2 introduces lock-free techniques with the example of a LIFO stack, section 3 presents our proposed lock-free FIFO queue algorithm, section 4 discuss the correctness of the FIFO operations and section 5 is dedicated to performances issues.

2. Lock-free LIFO stacks

A LIFO stack is made up of linked cells. A *cell* can be anything provided it starts with a pointer available to link together the cells of the stack (figure 1) and the structure of a LIFO is a simple pointer to the top of the stack (figure 2). The last cell of the LIFO always points to NULL.

```
structure cell {
    next:      a pointer to next cell
    value:     any data type
}
```

Figure 1: a cell structure

```
structure lifo {
    top:       a pointer to a cell
}
```

Figure 2: a lifo structure

Common operations on a LIFO are:

- **lifo-init:** to initialize the LIFO stack by setting the top pointer to NULL.
- **lifo-push:** to push a new cell on top of the stack
- **lifo-pop:** to pop the top cell of the stack

A naive and unsafe implementation of the push operation is presented in figure 3.

```
lifo-push (lf: pointer to lifo, cl: pointer to cell)
A1:      cl->next = lf->top          # set the cell next pointer to top of the lifo
A2:      lf->top = cl              # set the top of the lifo to cell
```

Figure 3: non-atomic lifo-push

Obviously, if a process trying to enqueue a new cell is preempted after A1 and if the top pointer has been modified when it resumes at A2, the push operation will not operate correctly.

2.1. Atomic operations implementation

To guaranty the correctness of the lifo operations, they should appear as taking instantaneously effect, as if they couldn't be interrupted. We'll further talk of “*atomic operation*” to refer to this property. A common approach is to make use of an atomic primitive such as *compare-and-swap* which takes as argument the address of a memory location, an expected value and a new value (figure 4). If the location holds the expected value, it is assigned the new value atomically. The returned boolean value indicates whether the replacement occurred.

```
compare-and-swap (addr: pointer to a memory location, old, new: expected and new values): boolean
x = read (addr)
if x == old
    write (addr, new)
    return true
else
    return false
endif
```

Figure 4: atomic compare-and-swap

The *compare-and-swap* primitive was first implemented in hardware in the IBM System 370 architecture [7]. More recently, it can be found on the Intel i486 [8] and on the Motorola 68020 [9]. A variation of the *compare-and-swap* primitive can also operate in memory on double-words. To differenciate between the two primitives in the following examples we'll refer to them with:

CAS (mem, old, new) for single word operations
 where *mem* is a pointer to a memory location
old and *new* are the expected and the new value

and

CAS2 (mem, old1, old2, new1, new2) for double word operations
 where *mem* is a pointer to a memory location
old1, *old2* and *new1*, *new2* are the expected and the new values

On PowerPC architecture, the *compare-and-swap* primitive may be implemented using the *load-and-reserve* instruction associated with a *store-conditional* instruction [10].

Using compare-and-swap, the operations on the stack are now implemented as shown in figure 5 and 6 and appear like atomic operations.

```
lifo-push (lf: pointer to lifo, cl: pointer to cell)
B1:      loop
B2:          cl->next = lf->top          # set the cell next pointer to top of the lifo
B3:          if CAS (&lf->top, cl->next, cl)  # try to set the top of the lifo to cell
B4:              break
B5:          endif
B6:      endloop
```

Figure 5: lifo-push

```
lifo-pop (lf: pointer to lifo): pointer to cell
C1:      loop
C2:          head = lf->top          # get the top cell of the lifo
C3:          if head == NULL        # LIFO is empty
C4:              return NULL
C5:          endif
C6:          next = head->next      # get the next cell of cell
C7:          if CAS (&lf->top, head, next)  # try to set the top of the lifo to the next cell
C8:              break
C9:          endif
C10:     endloop
C11:     return head
```

Figure 6: lifo-pop

2.2. The ABA problem

However, the above implementation of the LIFO pop operations doesn't catch the ABA problem. Assume that a process is preempted while dequeuing a cell after C6: several concurrent push and pop operations may result in a situation where the top cell remains unchanged but points to a different next cell as shown in figure 7.

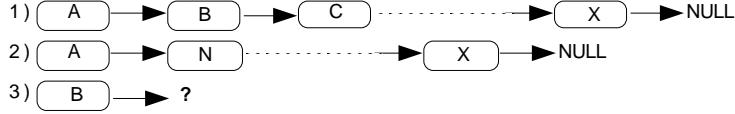


Figure 7: 1) state at the beginning of the pop operation,

2) state after preemption,

3) state after pop completion

The LIFO change won't prevent the CAS operation to operate in C7, allowing to put a wrong cell on top of the stack. The solution to the ABA problem consists in adding a count of the cells popped from the stack to the LIFO structure as shown in figure 8 and to make use of the CAS2 primitive.

```
structure lifo {
    top:      a pointer to a cell
    ocount:   total count of pop operations
}
```

Figure 8: extended lifo structure

The push operation remains unchanged and the pop operation is now implemented as shown in figure 9: it checks both for lifo top and output count changes when trying to modify the lifo top.

```
lifo-pop (lf: pointer to lifo): pointer to cell
SC1:  loop
SC2:      head = lf->top                      # get the top cell of the lifo
SC2:      oc = lf->ocount                      # get the pop operations count
SC3:      if head == NULL
SC4:          return NULL                         # LIFO is empty
SC5:      endif
SC6:      next = head->next                     # get the next cell of cell
SC7:      if CAS2 (&lf->top, head, oc, next, oc + 1)  # try to change both the top of the lifo and pop count
SC8:          break
SC9:      endif
SC10: endloop
SC11: return head
```

Figure 9: lifo-pop catching the ABA problem

3. Lock-free FIFO stacks

The FIFO queue is implemented as a linked list of cells with *head* and *tail* pointers. Each pointer have an associated counter, *ocount* and *icount*, which maintains a unique modification count of operations on *head* and *tail*. The cell structure is the same as above (figure 1) and the fifo structure is shown in figure 10.

```
structure fifo {
    head:      a pointer to head cell
    ocount:   total count of pop operations
    tail:      a pointer to tail cell
    icount:   total count of push operations
}
```

Figure 10: the fifo structure

As in Michael-Scott [4] and Valois [3], the FIFO always contains a dummy cell, only intended to maintain the consistency. An empty FIFO contains only this dummy cell which points to an *end fifo marker* unique to the system: a trivial solution consists in using the FIFO address itself as a unique marker. All along the operations, *head* always points to the dummy cell which is the first cell in the list and *tail* always points to the last or the second last cell in the list. The double-word *compare-and-swap* increments the modification counters to avoid the ABA problem.

The queue consistency is maintained by *cooperative concurrency*: when a process trying to enqueue a cell detects a pending enqueue operation (*tail* is not the last cell of the list), it first tries to complete the pending operation before enqueueing the cell. The dequeue operation also ensures that the *tail* pointer does not point to the dequeued cell and if necessary, tries to complete any pending enqueue operation. Figure 11 to 13 presents the commented pseudo-code for the fifo queue operations.

```

fifo-init (ff: pointer to fifo, dummy: pointer to dummy cell)
    dummy->next = NULL
    ff->head = ff->tail = dummy
    # makes the cell the only cell in the list
    # both head and tail point to the dummy cell

```

Figure 11: the fifo initialization operation

```

fifo-push (ff: pointer to fifo, cl: pointer to cell)
    E1:   cl->next = ENDFIFO(ff)           # set the cell next pointer to end marker
    E2:   loop                                # try until enqueue is done
    E3:     icount = ff->icount             # read the tail modification count
    E4:     tail = ff->tail                # read the tail cell
    E5:     if CAS (&tail->next, ENDFIFO(ff), cl)  # try to link the cell to the tail cell
    E6:       break;                         # enqueue is done, exit the loop
    E7:     else    # tail was not pointing to the last cell, try to set tail to the next cell
    E8:       CAS2 (&ff->tail, tail, icount, tail->next, icount+1)
    E9:     endif
    E10:  endloop
    E11:  CAS2 (&ff->tail, tail, icount, cl, icount+1)  # enqueue is done, try to set tail to the enqueued cell

```

Figure 12: the fifo push operation

```

fifo-pop (ff: pointer to fifo): pointer to cell
    D1:   loop                                # try until dequeue is done
    D2:     ocount = ff->ocount             # read the head modification count
    D3:     icount = ff->icount             # read the tail modification count
    D4:     head = ff->head                # read the head cell
    D5:     next = head->next              # read the next cell
    D6:     if ocount == ff->ocount        # ensures that next is a valid pointer
                                         # to avoid failure when reading next value
                                         # is queue empty or tail falling behind ?
    D7:       if head == ff->tail          # is queue empty ?
    D8:         if next == ENDFIFO(ff)      # queue is empty: return NULL
    D9:           return NULL
    D10:        endif
                                         # tail is pointing to head in a non empty queue, try to set tail to the next cell
    D11:        CAS2 (&ff->tail, head, icount, next, icount+1)
    D12:      else if next <> ENDFIFO(ff)  # if we are not competing on the dummy next
    D13:        value = next->value        # read the next cell value
    D14:        if CAS2 (&ff->head, head, ocount, next, ocount+1) # try to set head to the next cell
    D15:          break                  # dequeue done, exit the loop
    D16:        endif
    D17:      endif
    D18:    endloop
    D19:    head->value = value          # set the head value to previously read value
    D20:  return head                  # dequeue succeed, return head cell

```

Figure 13: the fifo pop operation

4 Correctness of the FIFO operations

Traditional sequential programs may be viewed as functions from inputs to outputs which may be specified as a pair consisting of a precondition describing the allowed inputs and postcondition describing the desired results for these inputs. However for concurrent programs, this approach is too limited and numerous work has been done for formal verification of concurrent systems. Although informal, two properties introduced by Lamport [11] are required for correctness of concurrent programs:

- *safety property*: states that “something bad never happens”,
- *liveness property*: states that “something good eventually happens”.

Formalizing this classification has been a main motivation for much of the work done on specification and verification of concurrent systems [12]. Formal methods successfully applied to sequential programs have also been extended to consider concurrent programming: Herlihy proposed a correctness condition for concurrent objects called “*Linearizability*” [13, 14]. It states that a concurrent computation is linearizable if it is equivalent to a legal sequential computation. An object (viewed as the aggregate of a type, which defines a set of possible values, and a set of primitive operations), is linearizable if each operation appears to take effect instantaneously at some point between the operation’s invocation and response. It implies that processes appear to be interleaved at the granularity of complete operations and that the order of non-overlapping operations is preserved.

Correctness of the FIFO operations formal proof is beyond the scope of this paper, however it will be examined according to the properties mentioned above.

3.1 Linearizability

The algorithm is linearizable because each operation takes effect at an atomic specific point: E5 for enqueue and D14 for dequeue. Therefore, the queue will never enter any transient unsafe state: along any concurrent implementation history, it can only swing between the two different states S0 and S1 illustrated in figure 14 and 15, which are acceptable and safe states for the queue:

Assuming a queue in state S0:

- 1) consider an *push* operation : as the queue state is S0, the atomic operation in E5 will succeed and the queue swings to S1 state. Then the atomic operation in E10 is executed: in case of success, the queue swings back to S0, in case of failure a successfull concurrent operation occurs on a S1 state and therefore by 3) and 4), the queue state should be S0.
- 2) consider a *pop* operation : if the queue is empty the operation returns in D9 and the state remains unchanged, otherwise the operation atomically executes D14: in case of success, the queue state remains in S0, in case of failure, a concurrent dequeue occurred and as it has successfully operated on a S0 queue (by hypothesis) the final state remains also in S0.

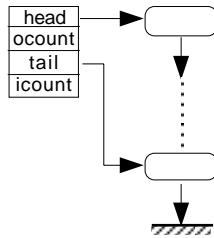


Figure 14: FIFO state S0

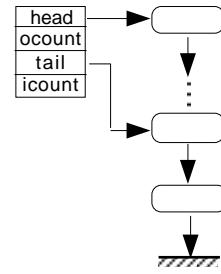


Figure 15: FIFO state S1

Assuming a queue in state S1:

- 3) consider an *enqueue* operation: as the queue state is S1, the operation atomically executes E8 and then loops. In case of success, the queue swings to S0 otherwise a concurrent dequeue or enqueue successfully occurred and the operation loop should operate on a queue back to S0.
- 4) consider a *dequeue* operation: it is concerned by S1 only if *tail* and *head* points to the same cell which is only possible with a queue containing a single cell linked to the dummy cell. In this case, the operation atomically executes D11 and then loop. In case of success, the queue swings to S0 state. A failure means that a concurrent dequeue or enqueue successfully occurred: a successfull dequeue swing the queue to S0 (but it is now empty) and a successfull enqueue too (by 3).

3.2 Safety

The main difference with the Michael-Scott algorithm [4] relies on the cells structure constraint, which allows to avoid nodes allocation and release. In fact, the cells memory management is now in charge of the FIFO clients and may be optimised to the clients requirements but it doesn't introduce any change in the algorithm functionning. Another difference is the modification counts to take account of the ABA problem: they are now associated only to the *head* and *tail* pointers to ensures atomic modifications of these pointers.

The safety properties satisfied by the Michel-Scott algorithm continue to hold ie:

- the linked list is always connected,
- cells are only inserted after the last cell in the linked list,
- cells are only deleted from the beginning of the linked list,
- head always points to the first node in the linked list,
- tail always points to a node in the linked list.

3.3 Liveness

The lock-free algorithm is non-blocking. This is asserted similarly to [4].

Assume a process attempting to operate on the queue:

- the process tries to enqueue a new cell: a failure means that the process is looping thru E8 and then another process must have succeeded in completing an enqueue operation or in dequeuing the tail cell.
- the process tries to dequeue a cell: a failure means that the process is looping thru D11 or D14. A failure in D11 means that another process must have succeeded in completing an enqueue operation or in dequeuing the tail cell. A failure in D14 means that another process must have succeeded in completing a dequeue operation.

5 Performances

Performances have been measured both for the lock-free LIFO compared to a lock-based implementation and for the lock-free FIFO algorithm compared to a lock-based implementation and to the Michael Scott algorithm. The bench has been made on a Bi-Celeron 500MHz SMP station running a 2.4.8 Linux kernel. It measures the time required for 1 to 8 concurrent threads to perform 500 000 x 6 concurrent push and pop operations on a shared LIFO or FIFO queue. The code executed by each thread is shown in Figure 16. The lock-based implementation makes use of the pthread mutex API with a statically allocated mutex.

```
long stacktest (long n) {
    cell* tmp[6]; long i; clock_t t0, t1;

    t0 = clock();
    while (n--) {
        for (i=0; i<6; i++) tmp[i] = pop(&gstack);
        for (i=0; i<6; i++) push(&gstack, tmp[i]);
    }
    t1 = clock();
    return t1-t0;
}
```

Figure 16: the bench task.

The integrity of the queue was checked after the threads had completed their operations. Results are presented by figures 17 and 18 as average time (in μ s) to perform a paired pop and push operations.

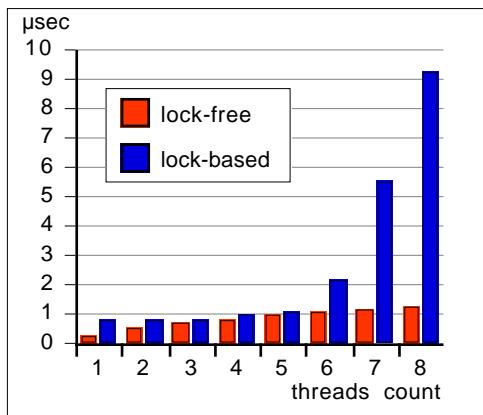


Figure 17: lock-free LIFO compared to lock-based.

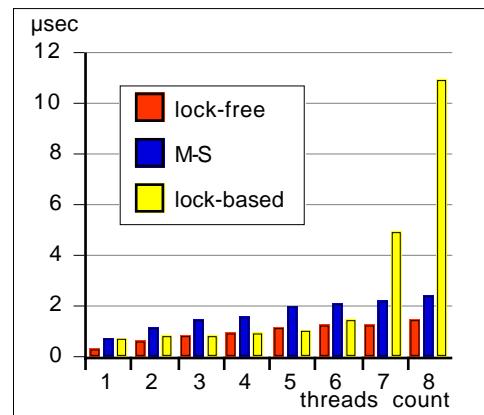


Figure 18: lock-free FIFO compared to Michael-Scott and lock-based.

In the Michael-Scott implementation, nodes allocation is performed using a statically allocated set of nodes and an index atomically incremented to access the next free node in the table (figure 19). The node table size prevents multiple node allocation. A node release is implicit and needs no additional operation.

```
node_t * new_node() {
    static long index = 0;
    long next, i;
    do {
        i = index;
        next = (i >= MAXNODES) ? 0 : i+1;
    } while (!CAS(&index, i, next));
    return &nodes[next];
}
```

Figure 19: node allocation in Michael Scott implementation

Comparison between the lock-free and the lock-based operations shows the following:

- in lack of concurrency (single thread), the lock-based operations are more than 2 times more expensive than the lock-free operations,
- performances are roughly the same for a few concurrency (2 to 5 threads),
- lock-based operations cost dramatically increases in medium-high concurrency to reach more than 7 times the lock-free cost for 8 concurrent threads.

Comparison between our lock-free FIFO algorithm and the Michael-Scott algorithm shows the following:

- for a single thread, the Michael-Scott operations cost is roughly 2 times more expensive
- when the concurrency increases, this cost is converging to 1.6 times our solution cost.

This behavior may be explained by the necessity to allocate the nodes pushed on the stack and to handle additional concurrency while performing the allocation.

7. Conclusion

Lock-free techniques are clearly more suited to real-time applications than lock-based techniques. They are more efficient and avoid priority inversion which is a major drawback in a real-time context. We have showed how to apply this technique to simple objects like LIFO and FIFO queues associated with basic operations. Finally, our proposed new algorithm for FIFO operations improves existing algorithms with a simple constraint on the value data structure which allows more efficient specialized implementations. Although limited to LIFO and FIFO queues, the presented lock-free techniques may be very useful to solve situations commonly encountered in the musical domain where events have frequently to be queued while waiting for their deadline.

8. Acknowledgements

Thanks to Shahar Frank <fesh@exanet.com> who reported the *fifo-pop* problem and for its suggested solution.

References

- [1] James H. Anderson, Srikanth Ramamurthy and Kevin Jeffay. "Real-time computing with lock-free shared objects." ACM Transactions on Computer Systems Vol. 15, No. 2, May 1997, pp. 134 - 165
- [2] M. Herlihy. "A methodology for implementing highly concurrent data objects." ACM Trans. Program. Lang. Syst. 1993, Vol. 15, No.5, pp. 745–770.
- [3] John D. Valois. "Implementing Lock-Free Queues." Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems, Las Vegas, October 1994, pp. 64-69
- [4] M. M. Michael and M. L. Scott. "Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms." 15th ACM Symp. on Principles of Distributed Computing (PODC), May 1996. pp. 267 - 275
- [5] M. M. Michael and M. L. Scott. "Nonblocking Algorithms and Preemption-Safe Locking on Multiprogrammed Shared Memory Multiprocessors." Journal of Parallel and Distributed Computing, 1998, pp. 1-26.
- [6] Y. Orlarey, H. Lequay. "MidiShare : a Real Time multi-tasks software module for Midi applications" Proceedings of the International Computer Music Conference 1989, Computer Music Association, San Francisco, pp.234-237
- [7] International Business Machines Corp. "System / 370 Principles of Operation" 1983
- [8] Intel Corporation. "i486 Processor Programmer's reference Manual" Intel, Santa Clara, CA, 1990
- [9] Motorola. "MC68020 32-Bit Microprocessor User's Manual" Prentice-Hall, 2nd edition, 1986
- [10] IBM Microelectronics, Motorola. "PowerPC 601 RISC Microprocessor User's Manual", 1993
- [11] L. Lamport. "Proving the Correctness of Multiprocessor Programs." IEEE Transactions on Software Engineering SE-3, 2 (March 1977), 125-143.
- [12] R. Cleaveland, S.A. Smolka & al. "Strategic Directions in Concurrency Research." ACM Computing Surveys, Vol. 28, No. 4, December 1996, pp. 607-625
- [13] M. P. Herlihy, J. M. Wing. "Axioms for concurrent objects." In Proceedings of the 14th ACM Symposium on Principles of Programming Languages, Jan. 1987, pp. 13-26.
- [14] M. P. Herlihy, J. M. Wing. "Linearizability: A Correctness Condition for Concurrent Objects." ACM Transactions on Programming Languages and Systems, Vol. 12, No. 3, July 1990, pp. 463-492.

Nonblocking Algorithms and Preemption-Safe Locking on Multiprogrammed Shared Memory Multiprocessors¹

Maged M. Michael² and Michael L. Scott

Department of Computer Science, University of Rochester,
Rochester, New York 14627-0226
E-mail: michael@watson.ibm.com, scott@cs.rochester.edu

Received January 14, 1997; revised December 27, 1997; accepted January 10, 1998

Most multiprocessors are multiprogrammed to achieve acceptable response time and to increase their utilization. Unfortunately, inopportune preemption may significantly degrade the performance of synchronized parallel applications. To address this problem, researchers have developed two principal strategies for a concurrent, atomic update of shared data structures: (1) *preemption-safe locking* and (2) *nonblocking* (lock-free) *algorithms*. Preemption-safe locking requires kernel support. Nonblocking algorithms generally require a universal atomic primitive such as compare-and-swap or load-linked/store-conditional and are widely regarded as inefficient.

We evaluate the performance of preemption-safe lock-based and nonblocking implementations of important data structures—queues, stacks, heaps, and counters—including nonblocking and lock-based queue algorithms of our own, in microbenchmarks and real applications on a 12-processor SGI Challenge multiprocessor. Our results indicate that our nonblocking queue consistently outperforms the best known alternatives and that data-structure-specific nonblocking algorithms, which exist for queues, stacks, and counters, can work extremely well. Not only do they outperform preemption-safe lock-based algorithms on multiprogrammed machines, they also outperform ordinary locks on dedicated machines. At the same time, since general-purpose nonblocking techniques do not yet appear to be practical, preemption-safe locks remain the preferred alternative for complex data structures: they outperform conventional locks by significant margins on multiprogrammed systems. © 1998 Academic Press

¹ This work was supported in part by NSF Grants CDA-94-01142 and CCR-93-19445 and by ONR Research Grant N00014-92-J-1801 (in conjunction with the DARPA Research in Information Science and Technology-High Performance Computing, Software Science, and Technology program, ARPA Order 8930).

² Current address: IBM Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598.

Key Words: nonblocking; lock-free; mutual exclusion; locks; multiprogramming; concurrent queues; concurrent stacks; concurrent heaps; concurrent counters; concurrent date structures; compare-and-swap; load-linked; store-conditional.

1. INTRODUCTION

Shared data structures are widely used in parallel applications and multiprocessor operating systems. To ensure the consistency of these data structures, processes perform synchronized concurrent update operations, mostly using critical sections protected by mutual exclusion locks. To achieve acceptable response time and high utilization, most multiprocessors are multiprogrammed by time-slicing processors among processes. The performance of mutual exclusion locks in parallel applications degrades significantly on time-slicing multiprogrammed systems [42] due to the preemption of processes holding locks. Any other processes busy-waiting on the lock are then unable to perform useful work until the preempted process is rescheduled and subsequently releases the lock.

Alternative multiprogramming schemes to time-slicing have been proposed to avoid the adverse effect of time-slicing on the performance of synchronization operations. However, each has limited applicability and/or reduces the utilization of the multiprocessor. Coscheduling [29] ensures that all processes of an application run together. It has the disadvantage of reducing the utilization of the multiprocessor if applications have a variable amount of parallelism or if processes cannot be evenly assigned to time-slices of multiprocessor. Another alternative is hardware partitioning, under which no two applications share a processor. However, fixed size partitions have the disadvantage of resulting in poor response time when the number of processes is larger than the number of processors, and adjustable size partitions have the disadvantage of requiring applications to be able to adjust their number of processes as new applications join the system. Otherwise, processes from the same application might have to share the same processor, allowing one to be preempted while holding a mutual exclusion lock. Traditional time-slicing remains the most widely used scheme of multiprogramming on multiprocessor systems.

For time-sliced systems, researchers have proposed two principal strategies to avoid inopportune preemption: *preemption safe locking* and *nonblocking algorithms*. Most preemption-safe locking techniques require a widening of the kernel interface to facilitate cooperation between the application and the kernel. Generally, these techniques try either to recover from the preemption of lock-holding processes (or processes waiting on queued locks) or to avoid preempting processes while holding locks.

An implementation of a data structure is *nonblocking* (also known as *lock-free*) if it guarantees that at least one process of those trying to update the data structure concurrently will succeed in completing its operation within a bounded amount of time, assuming that at least one process is active, regardless of the state of other processes. Nonblocking algorithms do not require any communication with the kernel and by definition they cannot use mutual exclusion. Rather, they generally

rely on hardware support for a universal³ atomic primitive such as compare-and swap⁴ or the pair load-linked and store-conditional,⁵ while mutual exclusion locks can be implemented using weaker atomic primitives such as test-and-set, fetch-and-increment, or fetch-and-store.

Few of the above-mentioned techniques have been evaluated experimentally and then only in comparison to ordinary (preemption-oblivious) mutual exclusion locks. We evaluate the relative performance of preemption-safe and nonblocking atomic update techniques on multiprogrammed (time-sliced) as well as dedicated multiprocessor systems. We focus on four important data structures: queues, stacks, heaps, and counters. For queues, we present fast new nonblocking and lock-based algorithms [27]. Our experimental results, employing both microbenchmarks and real applications, on a 12-processor Silicon Graphics Challenge multiprocessor, indicate that our nonblocking queue algorithm outperforms existing algorithms under almost all circumstances. In general, efficient data-structure-specific non-blocking algorithms outperform both ordinary and preemption-safe lock-based alternatives, not only on time-sliced systems, but on dedicated machines as well [28]. At the same time, preemption-safe algorithms outperform ordinary locks on time-sliced systems and should therefore be supported by multiprocessor operating systems. We do not examine general-purpose nonblocking techniques in detail; previous work indicates that they are highly inefficient, though they provide a level of fault tolerance unavailable with locks. Our contributions include:

- A simple, fast, and practical nonblocking queue algorithm that outperforms all known alternatives and should be the algorithm of choice for multiprocessors that support universal atomic primitives.
- A two-lock queue algorithm that allows one enqueue and one dequeue to proceed concurrently. This algorithm should be used for heavily contended queues or multiprocessors with nonuniversal atomic primitives such as test-and-set or fetch-and-add.
- An evaluation of the performance of nonblocking algorithms in comparison to preemption-safe and ordinary (preemption-oblivious) locking for queues, stacks,

³ Herlihy [9] presented a hierarchy of nonblocking objects that also applies to atomic primitives. A primitive is at level n of the hierarchy if it can provide a nonblocking solution to a consensus problem for up to n processors. Primitives at higher levels of the hierarchy can provide nonblocking implementations of those at lower levels, but not conversely. Compare-and-swap and the pair load-linked and store-conditional are *universal* primitives as they are at level ∞ of the hierarchy. Widely supported primitives such as test-and-set, fetch-and-add, and fetch-and-store are at level 2.

⁴ Compare-and-swap, introduced on the IBM System 370, takes as arguments the address of a shared memory location, an expected value, and a new value. If the shared location currently holds the expected value, it is assigned the new value atomically. A Boolean return value indicates whether the replacement occurred. Compare-and-swap is supported on the Intel Pentium Pro and Sparc V9 architectures.

⁵ Load-linked and store-conditional, proposed by Jensen *et al.* [15], must be used together to read, modify, and write a shared location. Load-linked returns the value stored at the shared location. Store-conditional checks if any other processor has since written to that location. If not then the location is updated and the operation returns success, otherwise it returns failure. Load-linked/store-conditional is supported by the MIPS II, PowerPC, and Alpha architectures.

heaps, and counters. The paper demonstrates the superior performance of data-structure-specific nonblocking algorithms on time-slicing as well as dedicated multiprocessor systems.

The rest of this paper is organized as follows. We discuss preemption-safe locking in Section 2 and nonblocking algorithms in Section 3. In Section 4, we discuss nonblocking queue algorithms and present two concurrent queue algorithms of our own. We describe our experimental methodology and results in Section 5. Finally, we summarize our conclusions and recommendations in Section 6.

2. PREEMPTION-SAFE LOCKING

For simple mutual exclusion locks (e.g., test-and-set), preemption-safe locking techniques allow the system either to avoid or to recover from the adverse effect of the preemption of processes holding locks. Edler *et al.*'s Symunix system [7] employs an avoidance technique: a process may set a flag requesting that the kernel not preempt it because it is holding a lock. The kernel will honor the request up to a predefined time limit, setting a second flag to indicate that it did so and deducting any extra execution time from the beginning of the process's next quantum. A process yields the processor if it finds, upon leaving a critical section, that it was granted an extension.

The *first-class threads* of Marsh *et al.*'s Psyche system [21] employ a different avoidance technique: they require the kernel to warn an application process a fixed amount of time in advance of preemption by setting a flag that is visible in user space. If a process verifies that the flag is unset before entering a critical section (and if critical sections are short), then it is guaranteed to be able to complete its operation in the current quantum. If it finds the flag is set, it can voluntarily yield the processor.

Recovery-based preemption-safe locking techniques include the *spin-then-block* locks of Ousterhout [29] which let a waiting process spin for a certain period of time and then—if unsuccessful in entering the critical section—block, thus minimizing the adverse effect of waiting for a lock held by a descheduled process. Karlin *et al.* [16] presented a set of spin-then-block alternatives that adjust the spin time based on past experience. Black's work on Mach [6] introduced another recovery technique: a process may suggest to the kernel that it be descheduled in favor of some specific other process (presumably the one that is holding a desired lock). The *scheduler activations* of Anderson *et al.* [4] also support recovery: when a processor is taken from an application process, another active process belonging to the same application is informed via software interrupt. If the preempted process was holding a lock, the interrupted process can perform a context switch to the preempted process and push it through the critical section.

Simple preemption-safe techniques rely on the fact that processes acquire a test-and-set lock in nondeterministic order. Unfortunately, test-and-set locks do not scale well to large machines. Queue-based locks scale well, but impose a deterministic order on lock acquisitions, forcing a preemption-safe technique to deal with preemption not only of the process holding a lock, but of processes

waiting in the lock's queue as well. Preempting and scheduling processes in an order inconsistent with their order in the lock's queue can degrade performance dramatically. Kontothanassis *et al.* [17] presented preemption-safe (or “scheduler-conscious”) versions of the ticket lock, the MCS lock [24], and Krieger *et al.*'s reader-writer lock [18]. These algorithms detect the descheduling of critical processes using handshaking and/or a widened kernel-user interface and use this information to avoid handing the lock to a preempted process.

The proposals of Black and of Anderson *et al.* require the application to recognize the preemption of lock-holding processes and to deal with the problem. By performing recovery on a processor other than the one on which the preempted process last ran, they also sacrifice cache footprint. The proposal of Marsh *et al.* requires the application to estimate the maximum duration of a critical section, which is not always possible. To represent the preemption-safe approach in our experiments, we employ test-and-test-and-set locks with exponential backoff, based on the kernel interface of Edler *et al.*. For machines the size of ours (12 processors), the results of Kontothanassis *et al.* indicate that these will out-perform queue-based locks.

3. NONBLOCKING ALGORITHMS

Several nonblocking implementations of widely used data structures as well as general methodologies for developing such implementations systematically have been proposed in the literature. These implementations and methodologies were motivated in large part by the performance degradation of mutual exclusion locks as a result of arbitrary process delays, particularly those due to preemption on a multiprogrammed system.

3.1. General Nonblocking Methodologies

Herlihy [10] presented a general methodology for transforming sequential implementations of data structures into concurrent nonblocking implementations using compare-and-swap or load-linked/store-conditional. The basic methodology requires copying the entire data structure on every update. Herlihy also proposed an optimization by which the programmer can avoid some fraction of the copying for certain data structures; he illustrated this optimization in a non-blocking implementation of a skew-heap-based priority queue. Alemany and Felten [1] and LaMarca [19] proposed techniques to reduce unnecessary copying and useless parallelism associated with Herlihy's methodologies using extra communication between the operating system kernel and application processes. Barnes [5] presented a general methodology in which processes record and timestamp their modifications to the shared object and cooperate whenever conflicts arise. Shavit and Touitou [32] presented *software transactional memory*, which implements a k -word compare-and-swap using load-linked/store-conditional. Also, Anderson and Moir [2] presented nonblocking methodologies for large objects that rely on techniques for implementing multiple-word compare-and-swap using load-linked/store-conditional and vice versa. Turek *et al.* [39] and

Prakash *et al.* [30] presented methodologies for transforming multiple lock concurrent objects into lock-free concurrent objects. Unfortunately, the performance of nonblocking algorithms resulting from general methodologies is acknowledged to be significantly inferior to that of the corresponding lock-based algorithms [10, 19, 32].

Two proposals for hardware support for general nonblocking data structures have been presented: *transactional memory* by Herlihy and Moss [11] and the *Oklahoma update* by Stone *et al.* [37]. Neither of these techniques has been implemented on a real machine. The simulation-based experimental results of Herlihy and Moss show performance significantly inferior to that of spin locks. Stone *et al.* did not present experimental results.

3.2. Data-Structure-Specific Nonblocking Algorithms

Treiber [38] proposed a nonblocking implementation of concurrent link-based stacks. It represents the stack as a singly linked list with a *Top* pointer. It uses compare-and-swap to modify the value of *Top* atomically. Commented pseudo-code of Treiber's nonblocking stack algorithm is presented in Fig. 1. No performance results were reported for nonblocking stacks. However, Treiber's stack is very simple and can be expected to be quite efficient. We also observe that a stack derived from Herlihy's general methodology, with unnecessary copying removed, seems to be simple enough to compete with lock-based algorithms.

Valois [41] proposed a nonblocking implementation of linked lists. Simple non-blocking centralized counters can be implemented trivially using a fetch-and-add atomic primitive (if supported by hardware), or a read-modify-check-write cycle using compare-and-swap or load-linked/store-conditional.

```

structure pointer_t {ptr: pointer to node_t, count: unsigned integer}
structure node_t {value: data type, next: pointer_t}
structure stack_t {Top: pointer_t}

INITIALIZE(S: pointer to stack_t)
    S->Top.ptr = NULL                                # Empty stack. Top points to NULL

PUSH(S: pointer to stack_t, value: data type)
    node = new_node()
    node->value = value
    node->next.ptr = NULL
    repeat
        top = S->Top
        node->next.ptr = top.ptr
    until CAS(&S->Top, top, [node, top.count+1])      # Allocate a new node from the free list
                                                       # Copy stacked value into node
                                                       # Set next pointer of node to NULL
                                                       # Keep trying until Push is done
                                                       # Read Top.ptr and Top.count together
                                                       # Link new node to head of list
                                                       # Try to swing Top to new node

POP(S: pointer to stack_t, pvalue: pointer to data type): boolean
    repeat
        top = S->Top
        if top.ptr == NULL
            return FALSE
        endif
    until CAS(&S->Top, top, [top.ptr->next.ptr, top.count+1])  # Keep trying until Pop is done
    *pvalue = top.ptr->value                                # Read Top
    free(top.ptr)                                           # Is the stack empty?
    return TRUE                                            # The stack was empty, couldn't pop
                                                       # Try to swing Top to the next node
                                                       # Pop is done. Read value
                                                       # It is safe now to free the old node
                                                       # The stack was not empty, pop succeeded

```

FIG. 1. Structure and operation of Treiber's nonblocking concurrent stack algorithm [38].

```

ADD(X: pointer to integer, value: integer): integer
repeat
    count = LL(X)                                # Keep trying until SC succeeds
    until SC(X, count+value)                      # Read the current value of X
    return count                                    # Add is done, return previous value

```

FIG. 2. A nonblocking concurrent counter using load-linked and store-conditional.

Figure 2 shows a nonblocking counter implementation using load-linked/store-conditional.

Massalin and Pu [22] presented nonblocking algorithms for array-based stacks, array-based queues, and linked lists. Unfortunately, their algorithms require double-compare-and-swap, a primitive that operates on two arbitrary memory locations simultaneously and that appears to be available only on the Motorola 68020 processor and its direct descendants. No practical nonblocking implementations for array-based stacks or circular queues have been proposed. The general methodologies can be used, but the resulting algorithms would be very inefficient. For these data structures lock-based algorithms seem to be the only option.

In the following section, we continue the discussion of data-structure-specific nonblocking algorithms, concentrating on queues. Our presentation includes two new concurrent queue algorithms. One is non-blocking; the other uses a pair of mutual exclusion locks.

4. CONCURRENT QUEUE ALGORITHMS

4.1. Discussion of Previous Work

Many researchers have proposed lock-free algorithms for concurrent queues. Hwang and Briggs [14], Sites [33], and Stone [34] presented lock-free algorithms based on compare-and-swap. These algorithms are incompletely specified; they omit important details such as the handling of empty or single-item queues or concurrent enqueues and dequeues. Lamport [20] presented a wait-free algorithm that allows only a single enqueueer and a single dequeuer.⁶ Gottlieb *et al.* [8] and Mellor-Crummey [23] presented algorithms that are lock-free but not nonblocking: they do not use locking mechanisms, but they allow a slow process to delay faster processes indefinitely. Treiber [38] presented an algorithm that is nonblocking but inefficient: a dequeue operation takes time proportional to the number of the elements in the queue.

As mentioned above, Massalin and Pu [22] presented a nonblocking array-based algorithm based on double-compare-and-swap, a primitive available only on later members of the Motorola 68000 family of processors. Herlihy and Wing [12] presented an array-based algorithm that requires infinite arrays. Valois [40] presented an array-based algorithm that requires either an unaligned

⁶A *wait-free* algorithm is both nonblocking and starvation free: it guarantees that every active process will make progress within a bounded number of time steps.

compare-and-swap (not supported on any architecture) or a Motorola-like double-compare-and-swap.

Stone [35] presented a queue that is lock-free but nonlinearizable⁷ and not non-blocking. It is nonlinearizable because a slow enqueuer may cause a faster process to enqueue an item and subsequently observe an empty queue, even though the enqueued item has never been dequeued. It is not nonblocking because a slow enqueue can delay dequeues by other processes indefinitely. Our experiments also revealed a race condition in which a certain interleaving of a slow dequeue with faster enqueues and dequeues by other process(es) can cause an enqueued item to be lost permanently. Stone also presented [36] a nonblocking queue based on a circular singly linked list. The algorithm uses one anchor pointer to manage the queue instead of the usual head and tail. Our experiments revealed a race condition in which a slow dequeuer can cause an enqueued item to be lost permanently.

Prakash, Lee, and Johnson [31] presented a linearizable nonblocking algorithm that uses a singly linked list to represent the queue with *Head* and *Tail* pointers. It uses compare-and-swap to enqueue and dequeue nodes at the tail and the head of the list, respectively. A process performing an enqueue or a dequeue operation first takes a snapshot of the data structure and determines if there is another operation in progress. If so it tries to complete the ongoing operation and then takes another snapshot of the data structure. Otherwise it tries to complete its own operation. The process keeps trying until it completes its operation.

Valois [40] presented a list-based nonblocking queue algorithm that avoids the contention caused by the snapshots of Prakash *et al.*'s algorithm and allows more concurrency by keeping a dummy node at the head (dequeue end) of a singly linked list, thus simplifying the special cases associated with empty and single-item queues (a technique suggested by Sites [33]). Unfortunately, the algorithm allows the tail pointer to lag behind the head pointer, thus preventing dequeuing processes from safely freeing or reusing dequeued nodes. If the tail pointer lags behind and a process frees a dequeued node, the linked list can be broken, so that subsequently enqueued items are lost. Since memory is a limited resource, prohibiting memory reuse is not an acceptable option. Valois therefore proposes a special mechanism to free and allocate memory. The mechanism associates a reference counter with each node. Each time a process creates a pointer to a node it increments the node's reference counter atomically. When it does not intend to access a node that it has accessed before, it decrements the associated reference counter atomically. In addition to temporary links from process-local variables, each reference counter reflects the number of links in the data structure that point to the node in question. For a queue, these are the head and tail pointers and linked-list links. A node is freed only when no pointers in the data structure or temporary variables point to it. We discovered and corrected [26] race conditions in the memory management mechanism and the associated nonblocking queue algorithm.

⁷ An implementation of a data structure is *linearizable* if it can always give an external observer, observing only the abstract data structure operations, the illusion that each of these operations takes effect instantaneously at some point between its invocation and its response [13].

Most of the algorithms mentioned above are based on compare-and-swap and must therefore deal with the *ABA* problem: if a process reads a value *A* in a shared location, computes a new value, and then attempts a compare-and-swap operation, the compare-and-swap may succeed when it should not, if between the read and the compare-and-swap some other process(es) change the *A* to a *B* and then back to an *A* again. The most common solution is to associate a modification counter with a pointer, to always access the counter with the pointer in any read-modify-compare-and-swap sequence, and to increment it in each

```

structure pointer.t {ptr: pointer to node.t, count: unsigned integer}
structure node.t {value: data type, next: pointer.t}
structure queue.t {Head: pointer.t, Tail: pointer.t}

INITIALIZE(Q: pointer to queue.t)
    node = new_node()
    node→next.ptr = NULL
    Q→Head.ptr = Q→Tail.ptr = node
                                # Allocate a free node
                                # Make it the only node in the linked list
                                # Both Head and Tail point to it

ENQUEUE(Q: pointer to queue.t, value: data type)
E1:   node = new_node()
E2:   node→value = value
E3:   node→next.ptr = NULL
E4:   loop
E5:     tail = Q→Tail
E6:     next = tail.ptr→next
E7:     if tail == Q→Tail
E8:       if next.ptr == NULL
E9:         if CAS(&tail.ptr→next, next, [node, next.count+1])
E10:        break
E11:      endif
E12:    else
E13:      CAS(&Q→Tail, tail, [next.ptr, tail.count+1])
E14:    endif
E15:  endif
E16: endloop
E17: CAS(&Q→Tail, tail, [node, tail.count+1])
                                # Try to swing Tail to the inserted node

# Try to enqueue node into the linked list

DEQUEUE(Q: pointer to queue.t, pvalue: pointer to data type): boolean
D1:   loop
D2:     head = Q→Head
D3:     tail = Q→Tail
D4:     next = head.ptr→next
D5:     if head == Q→Head
D6:       if head.ptr == tail.ptr
D7:         if next.ptr == NULL
D8:           return FALSE
D9:         endif
D10:        CAS(&Q→Tail, tail, [next.ptr, tail.count+1])
D11:      else
D12:        *pvalue = next.ptr→value
D13:        if CAS(&Q→Head, head, [next.ptr, head.count+1])
D14:          break
D15:        endif
D16:      endif
D17:    endif
D18:  endloop
D19:  free(head.ptr)
D20:  return TRUE
                                # It is safe now to free the old dummy node
                                # Queue was not empty, dequeue succeeded

```

FIG. 3. Structure and operation of a nonblocking concurrent queue.

successful compare-and-swap. This solution does not guarantee that the *ABA* problem will not occur, but makes it extremely unlikely. To implement this solution, one must either employ a double-word compare-and-swap or else use array indices instead of pointers, so that they may share a single word with a counter. Valois's reference counting technique guarantees preventing the *ABA* problem without the need for modification counters or the double-word compare-and-swap. Mellor-Crummey's lock-free queue [23] requires no special precautions to avoid the *ABA* problem because it uses compare-and-swap in a fetch-and-store-modify-compare-and-swap sequence rather than the usual read-modify-compare-and-swap sequence. However, this same feature makes the algorithm blocking.

4.2. New Algorithms

We present two concurrent queue algorithms inspired by ideas in the work described above. Both of the algorithms are simple and practical. One is nonblocking; the other uses a pair of locks. Figure 3 presents commented pseudocode for the nonblocking queue data structure and operations. The algorithm implements the queue as a singly linked list with *Head* and *Tail* pointers. *Head* always points to a

```

structure node_t {value: data type, next: pointer to node_t}
structure queue_t {Head: pointer to node_t, Tail: pointer to node_t, H_lock: lock type, T_lock: lock type}

INITIALIZE(Q: pointer to queue_t)
    node = new_node()          # Allocate a free node
    node->next = NULL          # Make it the only node in the linked list
    Q->Head = Q->Tail = node  # Both Head and Tail point to it
    Q->H_lock = Q->T_lock = FREE # Locks are initially free

ENQUEUE(Q: pointer to queue_t, value: data type)
    node = new_node()          # Allocate a new node from the free list
    node->value = value        # Copy enqueued value into node
    node->next = NULL          # Set next pointer of node to NULL
    lock(&Q->T_lock)          # Acquire T_lock in order to access Tail
    Q->Tail->next = node      # Link node at the end of the linked list
    Q->Tail = node             # Swing Tail to node
    unlock(&Q->T_lock)         # Release T_lock

DEQUEUE(Q: pointer to queue_t, pvalue: pointer to data type): boolean
    lock(&Q->H_lock)          # Acquire H_lock in order to access Head
    node = Q->Head              # Read Head
    new_head = node->next        # Read next pointer
    if new_head == NULL          # Is queue empty?
        unlock(&Q->H_lock)        # Release H_lock before return
        return FALSE             # Queue was empty
    endif
    *pvalue = new_head->value     # Queue not empty. Read value before release
    Q->Head = new_head            # Swing Head to next node
    unlock(&Q->H_lock)          # Release H_lock
    free(node)                   # Free node
    return TRUE                 # Queue was not empty, dequeue succeeded

```

FIG. 4. Structure and operation of a two-lock concurrent queue.

dummy node, which is the first node in the list. *Tail* points to either the last or second to last node in the list. The algorithm uses compare-and-swap with modification counters to avoid the *ABA* problem. To allow dequeuing processes to free and then reuse dequeued nodes, the dequeue operation ensures that *Tail* does not point to the dequeued node or to any of its predecessors.

To obtain consistent values of various pointers we rely on sequences of reads that recheck earlier values to be sure they have not changed. These sequences of reads are similar to, but simpler than, the snapshots of Prakash *et al.* (we need to check only one shared variable rather than two). A similar technique can be used to prevent the race condition in Stone's blocking algorithm. A simple and efficient nonblocking stack algorithm due to Treiber [38] can be used to implement a nonblocking free list.

Figure 4 presents commented pseudocode for the two-lock queue data structure and operations. The algorithm employs separate *Head* and *Tail* locks to allow complete concurrency between enqueues and dequeues. As in the nonblocking queue, we keep a dummy node at the beginning of the list. Because of the dummy node, enqueueers never have to access *Head*, and dequeuers never have to access *Tail*, thus avoiding deadlock problems that might arise from processes trying to acquire the locks in different order.

Experimental results comparing these algorithms with others are presented in Section 5. A discussion of algorithm correctness is presented in Appendix A.

5. EXPERIMENTAL RESULTS

We use a Silicon Graphics Challenge multiprocessor with twelve 100 MHz MIPS R4000 processors to compare the performance of the most promising nonblocking, ordinary lock-based, and preemption-safe lock-based implementations of counters and of link-based queues, stacks, and skew heaps. We use microbenchmarks to compare the performance of the alternative algorithms under various levels of contention. We also use two versions of a parallel quicksort application, together with a parallel solution to the traveling salesman problem, to compare the performance of the algorithms when used in a real application.⁸

To ensure the accuracy of our results regarding the level of multiprogramming, we prevented other users from accessing the multiprocessor during the experiments. To evaluate the performance of the algorithms under different levels of multiprogramming, we used a feature of the Challenge's Irix operating system that allows programmers to pin processes to processors. We then used one of the processors to serve as a pseudoscheduler. Whenever a process is due for preemption, the pseudoscheduler interrupts it, forcing it into a signal handler. The handler spins on a flag which the pseudoscheduler sets when the process can continue computation. The time spent executing the handler represents the time during which the processor is taken from the process and handed over to a process that belongs to some other application. The time quantum is 10 ms.

⁸ C code for all the microbenchmarks and the real applications are available from ftp://ftp.cs.rochester.edu/pub/packages/sched_conscious_synch/multiprogramming.

All ordinary and preemption-safe locks used in the experiments are test-and-test-and-set locks with bounded exponential backoff. All nonblocking algorithms also bounded exponential backoff. The effectiveness of backoff in reducing contention on locks and synchronization data is demonstrated in the literature [3, 24]. The back-off was chosen to yield good overall performance for all algorithms and not to exceed 30 μ s. We emulate both test-and-set and compare-and-swap, using load-linked and store-conditional instructions, as shown in Fig. 5.

In the figures, multiprogramming level represents the number of applications sharing the machine, with one process per processor per application. A multiprogramming level of 1 (the top graph in each figure) therefore represents a dedicated machine; a multiprogramming level of 3 (the bottom graph in each figure) represents a system with a process from each of three different applications on each processor.

5.1. Queues

Figure 6 shows performance results for eight queue implementations on a dedicated system (no multiprogramming), and on multiprogrammed systems with two and three processes per processor. The eight implementations are the usual single-lock algorithm using both ordinary and preemption-safe locks (**single ordinary lock** and **single safe lock**); our two-lock algorithm, again using both ordinary and preemption-safe locks (**two ordinary locks** and **two safe locks**); our nonblocking algorithm (**MS nonblocking**) and those due to Prakash *et al.* [31] (**PLJ nonblocking**) and Valois [40] (**Valois nonblocking**); and Mellor-Crummey's blocking algorithm [23] (**MC blocking**). We include the algorithm of Prakash *et al.* because it appears to be the best of the known nonblocking alternatives. Mellor-Crummey's algorithm represents non-lock-based but blocking alternatives; it is simpler than the code of Prakash *et al.* and could be expected to display lower constant overhead in the absence of unpredictable process delays, but is likely to degenerate on a multiprogrammed system. We include Valois's algorithm to demonstrate that on multiprogrammed systems even a comparatively inefficient nonblocking algorithm can outperform blocking algorithms.

```

TESTANDSET(X: pointer to boolean): boolean
repeat
    local = LL(X)
    if local == TRUE
        return TRUE
    until SC(X, TRUE)
    return FALSE

# Keep trying SC succeeds or X is TRUE
# Read the current value of X
# TAS should return TRUE
# TAS is done, indicate that X was FALSE

COMPAREANDSWAP(X: pointer to integer, expected: integer, new: integer): boolean
repeat
    local = LL(X)
    if local ≠ expected
        return FALSE
    until SC(X, new)
    return TRUE

# Keep trying until SC succeeds or X ≠ expected
# Read the current value of X
# CAS should fail
# CAS succeeded

```

FIG. 5. Implementations of test-and-set and compare-and-swap using load-linked and store-conditional.

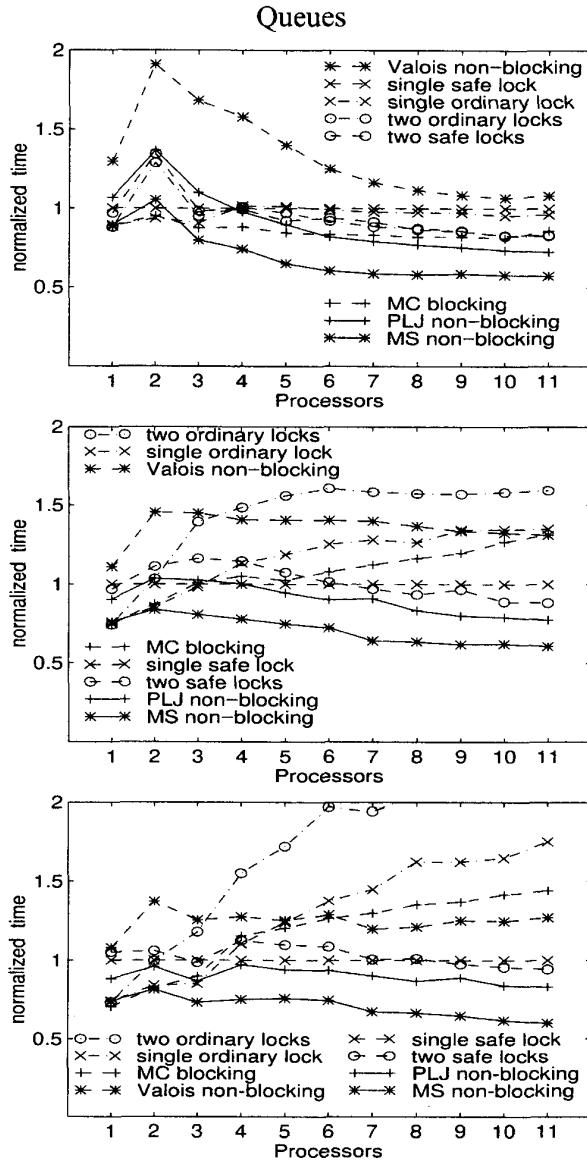


FIG. 6. Normalized execution time for 1,000,000 enqueue/dequeue pairs on a multiprogrammed system, with multiprogramming levels of 1 (top), 2 (middle), and 3 (bottom).

The horizontal axes of the graphs represent the number of processors. The vertical axes represent execution time normalized to that of the preemption-safe single lock algorithm. This algorithm was chosen as the basis of normalization because it yields the median performance among the set of algorithms. We use normalized time in order to show the difference in performance between the algorithms uniformly across different numbers of processors. If we were to use absolute time, the vertical

axes would have to be extended to cover the high absolute execution time on a single processor, making the graph too small to read for larger numbers of processors. The absolute times in seconds for the preemption-safe single-lock algorithm on one and 11 processors, with one, two, and three processes per processor, are 18.2 and 15.6, 38.8 and 15.4, and 57.6 and 16.3, respectively.

The execution time is the time taken by all processors to perform one million pairs of enqueues and dequeues to an initially empty queue (each process performs $1,000,000/p$ enqueue/dequeue pairs, where p is the number of processors). Every process spends $6 \mu\text{s}$ ($\pm 10\%$ randomization) spinning in an empty loop after performing every enqueue or dequeue operation (for a total of $12 \mu\text{s}$ per iteration). This time is meant to represent “real” computation. It prevents one process from dominating the data structure and finishing all its operations while other processes are starved by caching effects and backoff.

The results show that as the level of multiprogramming increases, the performance of ordinary locks and Mellor-Crummey’s blocking algorithm degrades significantly, while the performance of preemption-safe locks and nonblocking algorithms remains relatively unchanged. The “bump” at two processors is due primarily to cache misses, which do not occur on one processor, and to a smaller amount of overlapped computation, in comparison to larger numbers of processors. This effect is more obvious in the multiple lock and nonblocking algorithms, which have a greater potential amount of overlap among concurrent operations.

The two-lock algorithm outperforms the single-lock in the case of high contention since it allows more concurrency, but it suffers more with multiprogramming when using ordinary locks, as the chances are larger that a process will be preempted while holding a lock needed by other processes. On a dedicated system, the two-lock algorithm outperforms a single lock when more than four processors are active in our microbenchmark. With multiprogramming levels of 2 and 3, the crossover points for the one- and two-lock algorithms with preemption-safe locks occur at six and eight processors, respectively. The nonblocking algorithms, except for that of Valois, provide better performance; they enjoy added concurrency without the overhead of extra locks and without being vulnerable to interference from multiprogramming. Valois’s algorithm suffers from the high overhead of the complex memory management technique associated with it.

In the absence of contention, any overhead required to communicate with the scheduler in a preemption-safe algorithm is “wasted,” but the numbers indicate that this overhead is low.

Overall, our nonblocking algorithm yields the best performance. It outperforms the single-lock preemption-safe algorithm by more than 40% on 11 processors with various levels of multiprogramming, since it allows more concurrency and needs to access fewer memory locations. In the case of no contention, it is essentially tied with the single ordinary lock and with Mellor-Crummey’s queue.

5.2. *Stacks*

Figure 7 shows performance results for four stack implementations on a dedicated system and on multiprogrammed systems with two and three processes

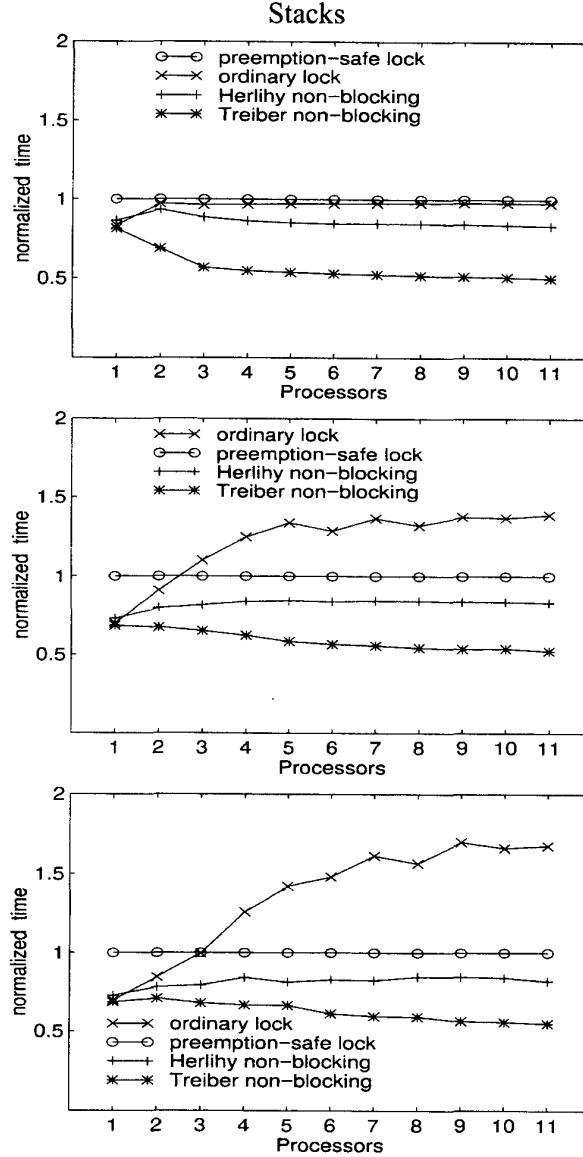


FIG. 7. Normalized execution time for 1,000,000 push/pop pairs on a multiprogrammed system, with multiprogramming levels of 1 (top), 2 (middle), and 3 (bottom).

per processor. The four stack implementations are the usual single-lock algorithm using ordinary and preemption-safe locks, Treiber's nonblocking stack algorithm [38], and an optimized nonblocking algorithm based on Herlihy's general methodology [10].

Like Treiber's nonblocking stack algorithm, the optimized algorithm based on Herlihy's methodology uses a singly linked list to represent the stack with a *Top*

pointer. However, every process has its own copy of *Top* and an operation is successfully completed only when the process uses load-linked/store-conditional to swing a shared pointer to its copy of *Top*. The shared pointer can be considered as pointing to the latest version of the stack.

The axes in the graphs have the same semantics as those in the queue graphs. Execution time is normalized to that of the preemption-safe single lock algorithm. The absolute times in seconds for the preemption-safe lock-based algorithm on one and 11 processors, within one, two, and three processes are 19.0 and 20.3, 40.8 and 20.7, and 60.2 and 21.6, respectively. Each process executes $1,000,000/p$ push/pop pairs on an initially empty stack, with a 6- μ s average delay between successive operations.

As the level of multiprogramming increases, the performance of ordinary locks degrades, while the performance of the preemption-safe and nonblocking algorithms remains relatively unchanged. Treiber's algorithm outperforms all the others even on dedicated systems. It outperforms the preemption-safe algorithm by over 45% on 11 processors with various levels of multiprogramming. This is mainly due to the fact that a push or a pop in Treiber's algorithm typically needs to access only two cache lines in the data structure, while a lock-based algorithm has the overhead of accessing lock variables as well. Accordingly, Treiber's algorithm yields the best performance even with no contention.

5.3. Heaps

Figure 8 shows performance results for three skew heap implementations on a dedicated system and on multiprogrammed systems with two and three processes per processor. The three implementations are the usual single-lock algorithm using ordinary and preemption-safe locks and an optimized nonblocking algorithm due to Herlihy [10].

The optimized nonblocking algorithm due to Herlihy uses a binary tree to represent the heap with a *Root* pointer. Every process has its own copy of *Root*. A process performing a heap operation copies the nodes it intends to modify to local free nodes and finally tries to swing a global shared pointer to its copy of *Root* using load-linked/store-conditional. If it succeeds, the local copies of the copied nodes become part of the global structure and the copied nodes are recycled for use in future operations.

The axes in the graphs have the same semantics as those for the queue and stack graphs. Execution time is normalized to that of the preemption-safe single lock algorithm. The absolute times in seconds for the preemption-safe lock-based algorithm on one and 11 processors, with one, two, and three processes per processor, are 21.0 and 27.7, 43.1 and 27.4, and 65.0 and 27.6, respectively. Each process executes $1,000,000/p$ insert/delete_min pairs on an initially empty heap with a 6- μ s average delay between successive operations. Experiments with nonempty heaps resulted in relative performance similar to that depicted in the graphs.

As the level of multiprogramming increases the performance of ordinary locks degrades, while the performance of the preemption-safe and nonblocking algorithms remains relatively unchanged. The degradation of the ordinary locks is

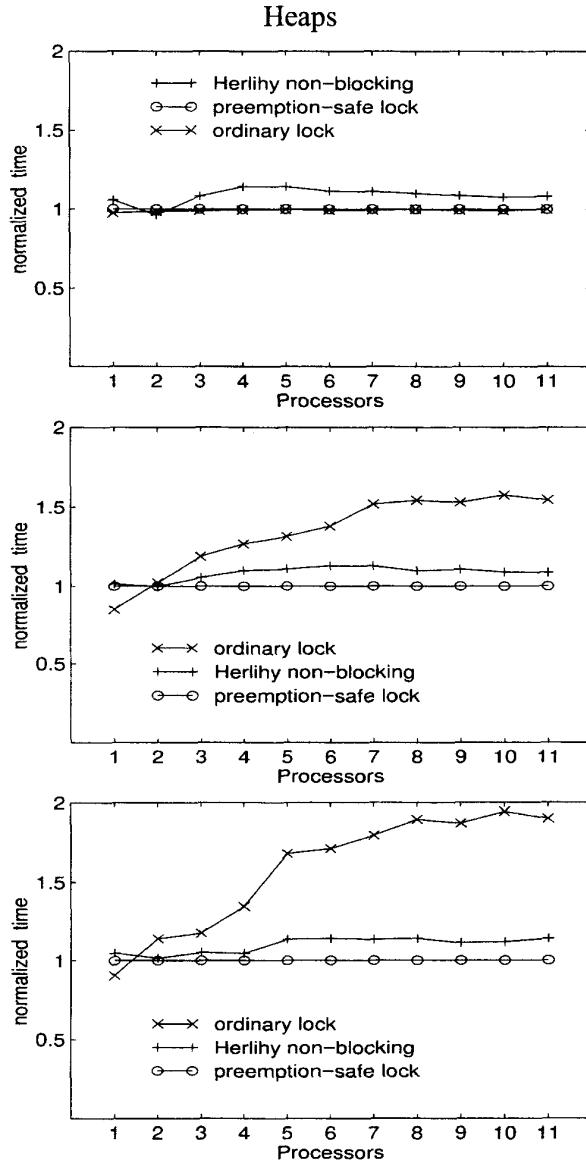


FIG. 8. Normalized execution time for 1,000,000 insert/delete_min pairs on a multiprogrammed system, with multiprogramming levels of 1 (top), 2 (middle), and 3 (bottom).

larger than that suffered by the locks in the queue and stack implementations, because the heap operations are more complex and result in higher levels of contention. Unlike the case for queues and stacks, the nonblocking implementation of heaps is quite complex. It cannot match the performance of the preemption-safe lock implementation on either dedicated or multiprogrammed systems, with or without contention. Heap implementations resulting from general nonblocking

methodologies (without data-structure-specific elimination of copying) are even more complex and could be expected to perform much worse.

5.4. Counters

Figure 9 shows performance results for three implementations of counters on a dedicated system and on multiprogrammed systems with two and three processes per processor. The three implementations are the usual single-lock algorithm using ordinary and preemption-safe locks and the nonblocking algorithm using load-linked/store-conditional.

The axes in the graphs have the same semantics as those for the previous graphs. Execution time is normalized to that of the preemption-safe single-lock algorithm. The absolute times in seconds for the preemption-safe lock-based algorithm on one and 11 processors, with one, two, and three processes per processor, are 17.7 and 10.8, 35.0 and 11.3, and 50.6 and 10.9, respectively. Each process executes $1,000,000/p$ increments on a shared counter with a 6- μ s average delay between successive operations.

The results are similar to those observed for queues and stacks, but are even more pronounced. The nonblocking algorithm outperforms the preemption-safe lock-based counter by more than 55% on 11 processors with various levels of multiprogramming. The performance of a fetch-and-add atomic primitive would be even better [25].

5.5. Quicksort Application

We performed experiments on two versions of a parallel quicksort application, one that uses a link-based queue and another that uses a link-based stack for distributing items to be sorted among the cooperating processes. We used three implementations for each of the queue and the stack: the usual single-lock algorithm using ordinary and preemption-safe locks and our nonblocking queue and Treiber's stack, respectively. In each execution, the processes cooperate in sorting an array of 500,000 pseudorandom numbers using quicksort for intervals of more than 20 elements and insertion sort for smaller intervals.

Figure 10 shows performance results for the three queue-based versions; Fig. 11 shows results for the three stack-based versions. Execution times are normalized to those of the preemption-safe lock-based algorithms. The absolute times in seconds for the preemption-safe lock-based algorithm on one and 11 processors, with one, two, and three processes per processor, are 4.0 and 1.6, 7.9 and 2.3, and 11.6 and 3.3, respectively, for a shared queue, and 3.4 and 1.5, 7.0 and 2.3, and 10.2 and 3.1, respectively, for a shared stack.

The results confirm our observations from experiments on microbenchmarks. Performance with ordinary locks degrades under multiprogramming, though not as severely as before, since more work is being done between atomic operations. Simple nonblocking algorithms yield superior performance even on dedicated systems, making them the algorithm of choice under any level of contention or multiprogramming.

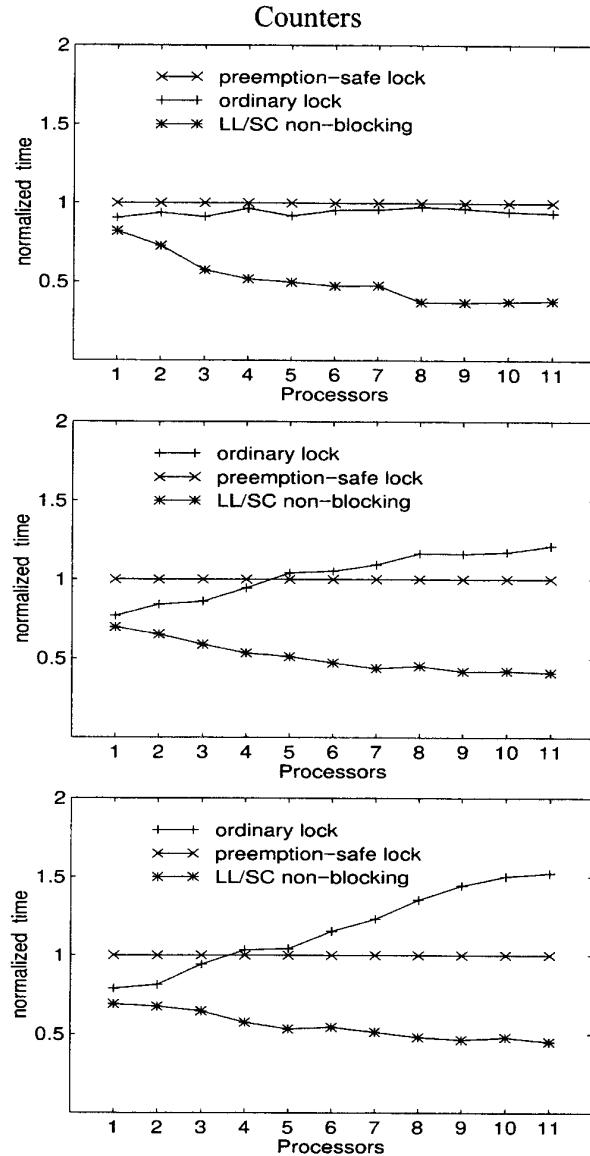


FIG. 9. Normalized execution time for 1,000,000 atomic increments on a multiprogrammed system, with multiprogramming levels of 1 (top), 2 (middle), and 3 (bottom).

5.6. Traveling Salesman Application

We performed experiments on a parallel implementation of a solution to the traveling salesman problem. The program uses a shared heap, stack, and counters. We used three implementations for each of the heap, stack, and counters: the usual single lock algorithm using ordinary and preemption-safe locks and the best respective

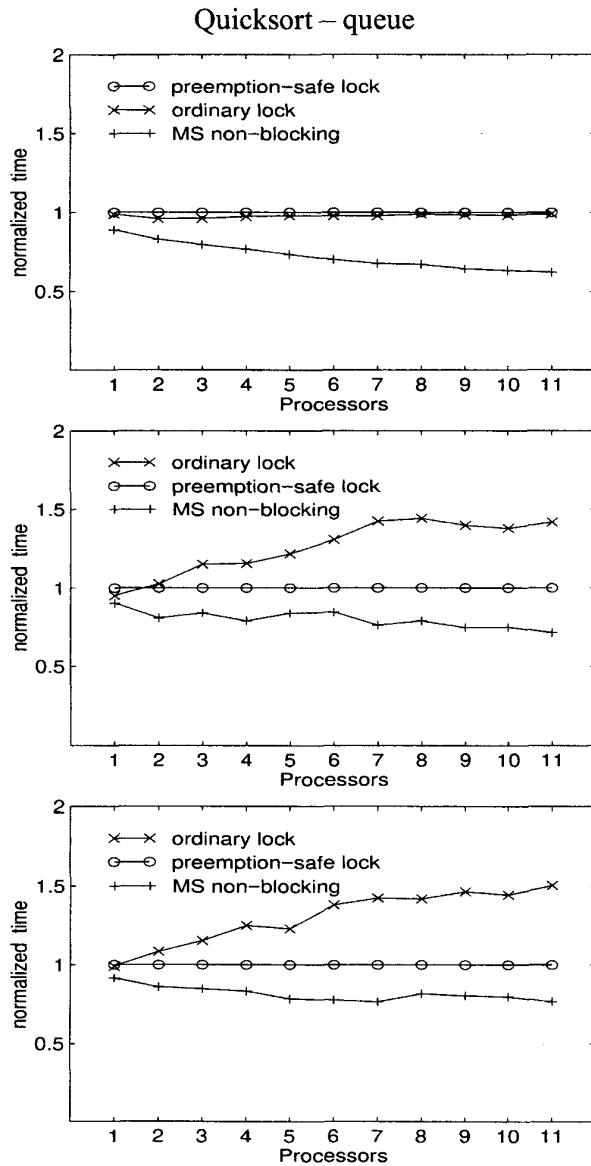


FIG. 10. Normalized execution time for quicksort of 500,000 items using a shared queue on a multiprogrammed system, with multiprogramming levels of 1 (top), 2 (middle), and 3 (bottom).

nonblocking algorithms (Herlihy-optimized, Treiber, and load-linked/store-conditional). In each execution, the processes cooperate to find the shortest tour in a 17-city graph. The processes use the priority queue heap to share information about the most promising tours and the stack to keep track of the tours that are yet to be computed. We ran experiments with each of the three implementations of the data structures. In addition, we ran experiments with a “hybrid” program

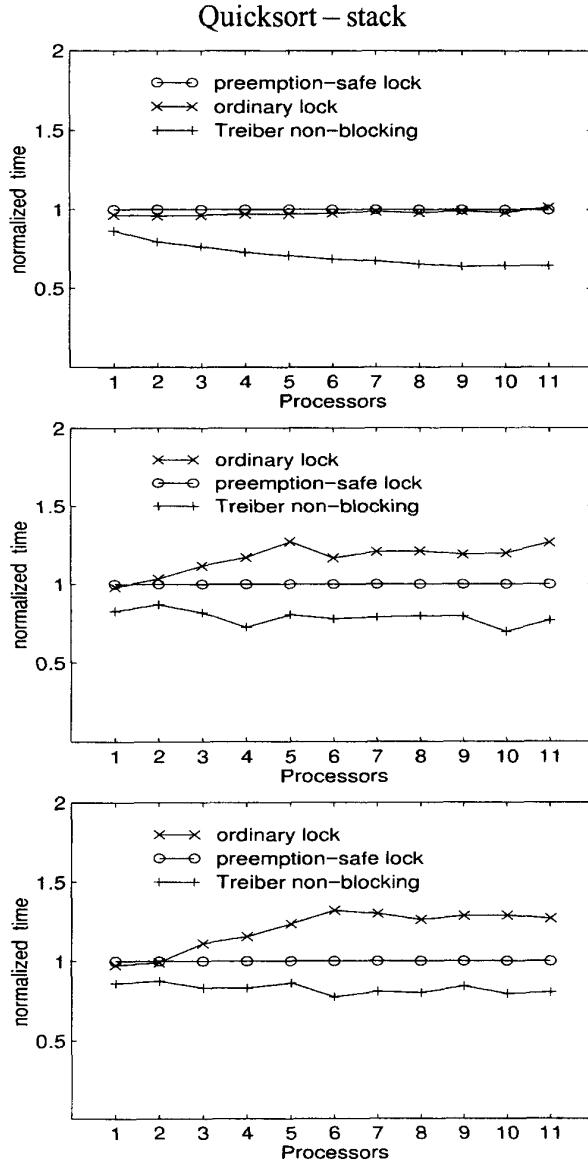


FIG. 11. Normalized execution time for quicksort of 500,000 items using a shared stack on a multiprogrammed system, with multiprogramming levels of 1 (top), 2 (middle), and 3 (bottom).

that uses the version of each data structure that ran the fastest for the micro-benchmarks: nonblocking stacks and counters and a preemption-safe priority queue.

Figure 12 shows performance results for the four different experiments. Execution times are normalized to those of the preemption-safe lock-based experiment. The absolute times in seconds for the preemption-safe lock-based experiment on one

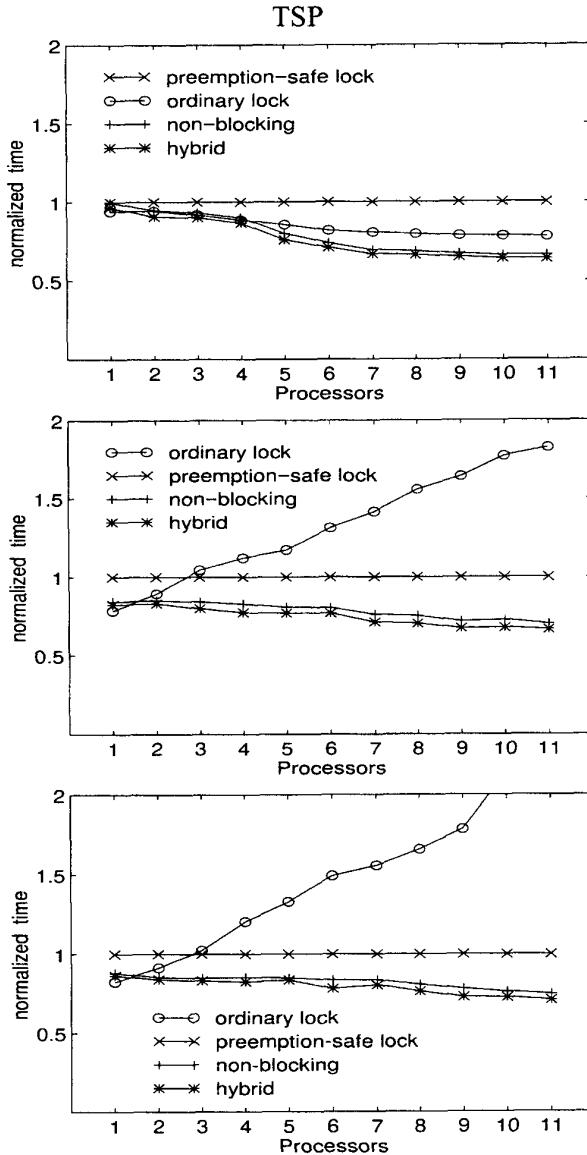


FIG. 12. Normalized execution time for a 17-city traveling salesman problem using a shared priority queue, stack and counters on a multiprogrammed system, with multiprogramming levels of 1 (top), 2 (middle), and 3 (bottom).

and 11 processors, with one, two, and three processes per processor, are 34.9 and 14.3, 71.7 and 15.7, and 108.0 and 18.5, respectively. Confirming our results with microbenchmarks, the experiment based on ordinary locks suffers under multiprogramming. The hybrid experiment yields the best performance, since it uses the best implementation of each of the data structures.

6. CONCLUSIONS

For atomic updates of a shared data structure, the programmer may ensure consistency using (1) a single lock, (2) multiple locks, (3) a general-purpose nonblocking technique, or (4) a special-purpose (data-structure-specific) nonblocking algorithm. The locks in (1) and (2) may or may not be preemption-safe.

Options (1) and (3) are easy to generate, given code for a sequential version of the data structure, but options (2) and (4) must be developed individually for each different data structure. Good data-structure-specific multilock and nonblocking algorithms are sufficiently tricky to devise that each has tended to constitute an individual publishable result.

Our experiments indicate that for simple data structures, special-purpose nonblocking atomic update algorithms will outperform all alternatives, not only on multiprogrammed systems, but on dedicated machines as well. Given the availability of a universal atomic hardware primitive, there seems to be no reason to use any other version of a link-based stack, a link-based queue, or a small, fixed-sized object such as a counter.

For more complex data structures, however, or for machines without universal atomic primitives, preemption-safe locks are clearly important. Preemption-safe locks impose a modest performance penalty on dedicated systems, but provide dramatic savings on time-sliced systems.

For the designers of future systems, we recommend (1) that hardware always include a universal atomic primitive, and (2) that kernel interfaces provide a mechanism for preemption-safe locking. For small-scale machines, the Symunix interface [7] appears to work well. For larger machines, a more elaborate interface may be appropriate [17].

We have presented a concurrent queue algorithm that is simple, nonblocking, practical, and fast. It appears to be the algorithm of choice for any queue-based application on a multiprocessor with a universal atomic primitive. Also, we have presented a two-lock queue algorithm. Because it is based on locks, it will work on machines with such nonuniversal atomic primitives as test-and-set. We recommend it for heavily utilized queues on such machines. For a queue that is usually accessed by only one or two processors, a single lock will perform better.

REFERENCES

1. J. Alemany and E. W. Felten, Performance issues in non-blocking synchronization on shared-memory multiprocessors, in "Proceedings of the Eleventh ACM Symposium on Principles of Distributed Computing," Vancouver, BC, August 1992, pp. 125–134.
2. J. H. Anderson and M. Moir, Universal constructions for multi-object operations, in "Proceedings of the Fourteenth ACM Symposium on Principles of Distributed Computing," Ottawa, Ontario, August 1995, pp. 184–194.
3. T. E. Anderson, The performance of spin lock alternatives for shared-memory multiprocessors, *IEEE Trans. Parallel Distrib. Syst.* **1**, 1 (January 1990), 6–16.
4. T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy, Scheduler activations: Effective kernel support for the user-level management of parallelism, *ACM Trans. Comput. Syst.* **10**, 1 (February 1992), 53–79.

5. G. Barnes, A method for implementing lock-free data structures, in "Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures," Velen, Germany, June–July 1993, pp. 261–270.
6. D. L. Black, Scheduling support for concurrency and parallelism in the mach operating system, *Computer* 23, 5 (May 1990), 35–43.
7. J. Edler, J. Lipkis, and E. Schonberg, Process management for highly parallel UNIX systems, in "Proceedings of the USENIX Workshop on Unix and Supercomputers," Pittsburgh, PA, September 1988.
8. A. Gottlieb, B. D. Lubachevsky, and L. Rudolph, Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors, *ACM Trans. Progrmg. Lang. Syst.* 5, 2 (April 1983), 164–189.
9. M. P. Herlihy, Wait-free synchronization, *ACM Trans. Progrmg. Lang. Syst.* 13, 1 (January 1991), 124–149.
10. M. P. Herlihy, A methodology for implementing highly concurrent data objects, *ACM Trans. Progrmg. Lang. Syst.* 15, 5 (November 1993), 745–770.
11. M. P. Herlihy and J. E. Moss, Transactional memory: Architectural support for lock-free data structures, in "Proceedings of the Twentieth International Symposium on Computer Architecture," San Diego, May 1993, pp. 289–300.
12. M. P. Herlihy and J. M. Wing, Axioms for concurrent object, in "Proceedings of the 14th ACM Symposium on Principles of Programming Languages," January 1987, pp. 13–26.
13. M. P. Herlihy and J. M. Wing, Linearizability: A correctness condition for concurrent objects, *ACM Trans. Progrmg. Lang. Syst.* 12, 3 (July 1990), 463–492.
14. K. Hwang and F. A. Briggs, "Computer Architecture and Parallel Processing," McGraw-Hill, New York, 1984.
15. E. H. Jensen, G. W. Hagensen, and J. M. Broughton, A new approach to exclusive data access in shared memory multiprocessors, Technical Report UCRL-97663, Lawrence Livermore National Laboratory, November 1987.
16. A. R. Karlin, K. Li, M. S. Manasse, and S. S. Owicki, Empirical studies of competitive spinning for a shared-memory multiprocessor, in "Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles," Pacific Grove, CA, October 1991, pp. 41–55.
17. L. I. Kontothanassis, R. W. Wisniewski, and M. L. Scott, Scheduler-conscious synchronization, *ACM Trans. Comput. Syst.* 15, 1 (February 1997).
18. O. Krieger, M. Stumm, and R. C. Unrau, A fair fast scalable reader–writer lock, in "Proceedings of the 1993 International Conference on Parallel Processing," St. Charles, IL, August 1993, pp. II: 201–204.
19. A. LaMarca, A performance evaluation of lock-free synchronization protocols, in "Proceedings of the Thirteenth ACM Symposium on Principles of Distributed Computing," Los Angeles, August 1994, pp. 130–140.
20. L. Lamport, Specifying concurrent program modulus, *ACM Trans. Progrmg. Lang. Syst.* 5, 2 (April 1983), 190–222.
21. B. D. Marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos, First-class user-level threads, in "Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles," Pacific Grove, CA, October 1991, pp. 110–121.
22. H. Massalin and C. Pu, A lock-free multiprocessor OS kernel, Technical report CUCS-005-91, Computer Science Department, Columbia University, 1991.
23. J. M. Mellor-Crummey, Concurrent queues: Practical fetch-and- Φ Algorithms, TR 229, Computer Science Department, University of Rochester, November 1987.
24. J. M. Mellor-Crummey and M. L. Scott, Algorithms for scalable synchronization on shared-memory multiprocessors, *ACM Trans. Comput. Syst.* 9, 1 (February 1991), 21–65.
25. M. M. Michael and M. L. Scott, Implementation of atomic primitives on distributed shared-memory multiprocessors, in "Proceedings of the First International Symposium on High Performance Computer Architecture," Raleigh, NC, January 1995," pp. 222–231.

26. M. M. Michael and M. L. Scott, Correction of a memory management method for lock-free data structures, Technical Report 599, Computer Science Department, University of Rochester, December 1995.
27. M. M. Michael and M. L. Scott, Simple, fast, and practical non-blocking and blocking concurrent queue algorithms, in "Proceedings of the Fifteenth ACM Symposium on Principles of Distributed Computing," Philadelphia, May 1996, pp. 267–275.
28. M. M. Michael and M. L. Scott, Relative performance of preemption-safe locking and non-blocking synchronization on multiprogrammed shared memory multiprocessors, in "Proceedings of the Eleventh International Parallel Processing Symposium," Geneva, Switzerland, April 1997.
29. J. K. Ousterhout, Scheduling techniques for concurrent systems, in "Proceedings of the Third International Conference on Distributed Computing Systems," October 1982, pp. 22–30.
30. S. Prakash, Y. H. Lee, and T. Johnson, Non-blocking algorithms for concurrent data structures, Technical Report 91-002, University of Florida, July 1991.
31. S. Prakash, Y. H. Lee, and T. Johnson, A nonblocking algorithm for shared queues using compare-and-swap, *IEEE Trans. Comput.* **43**, 5 (May 1994), 548–559.
32. N. Shavit and D. Touitou, Transactional memory, in "Proceedings of the Fourteenth ACM Symposium on Principles of Distributed Computing," Ottawa, Ontario, August 1995, pp. 204–213.
33. R. Sites, Operating systems and computer architecture, in "Introduction to Computer Architecture" (H. Stone, Ed.), 2nd ed. Chap. 12, 1980. Science Research Associates, Chicago.
34. H. S. Stone, "High Performance Computer Architecture," Addison-Wesley, Reading, MA, 1993.
35. J. M. Stone, A simple and correct shard-queue algorithm using compare-and-swap, in "Proceedings of Supercomputing '90," Minneapolis, November 1990, pp. 495–504.
36. J. M. Stone, A non-blocking compare-and-swap algorithm for a shared circular queue, in "Parallel and Distributed Computing in Engineering Systems" (S. Tzafestas *et al.*, Eds.), Elsevier Science, Amsterdam/New York, pp. 147–152, 1992.
37. J. M. Stone, H. S. Stone, P. Heidelberger, and J. Turek, Multiple reservations and the Oklahoma update, *IEEE Parallel Distrib. Tech.* **1**, 5 (November 1993), 58–71.
38. R. K. Treiber, Systems programming: Coping with parallelism, "RJ 5118, Almaden Research Center," April 1986.
39. J. Turek, D. Shasha, and S. Prakash, Locking without blocking: Making lock based concurrent data structure algorithms nonblocking, in "Proceedings of the 11th ACM Symposium on Principles of Database Systems," Vancouver, BC, August 1992, pp. 212–222.
40. J. D. Valois, Implementing lock-free queues, in "Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems," Las Vegas, NV, October 1994, pp. 64–69.
41. J. D. Valois, Lock-free linked lists using compare-and-swap, in "Proceedings of the Fourteenth ACM Symposium on Principles of Distributed Computing," Ottawa, Ontario, August 1995, pp. 214–222.
42. J. Zahorjan, E. D. Lazowska, and D. L. Eager, The effect of scheduling discipline on spin overhead in shared memory parallel systems, *IEEE Trans. Parallel Distrib. Syst.* **2**, 2 (April 1991), 180–198.

MAGED M. MICHAEL is a research staff member in the Multiprocessor Communication Architecture Department at the IBM Thomas J. Watson Research Center. He received a Ph.D. in computer science from the University of Rochester in 1997. His research interests include shared memory multiprocessor synchronization, multiprocessor architecture, execution-driven simulation, and cache coherence on large-scale multiprocessors. He devised concurrent algorithms for mutual exclusion locks, barriers and shared data structures. He is a co-designer of the Augmint multiprocessor execution-driven simulation environment.

MICHAEL L. SCOTT is Chair of the Computer Science Department at the University of Rochester, where he co-leads the Cashmere shared memory project. He received his Ph.D. in computer sciences in 1985 from the University of Wisconsin—Madison. His research interests include operating systems, programming languages, and program development tools for parallel and distributed computing. His textbook on programming language design and implementation is scheduled to be published by Morgan Kaufmann in 1998. Other recent publications have addressed scalable synchronization algorithms and software cache coherence.

Obstruction-Free Synchronization: Double-Ended Queues as an Example

Maurice Herlihy
Computer Science Department
Brown University, Box 1910
Providence, RI 02912

Victor Luchangco Mark Moir
Sun Microsystems Laboratories
1 Network Drive, UBUR02-311
Burlington, MA 01803

Abstract

We introduce obstruction-freedom, a new nonblocking property for shared data structure implementations. This property is strong enough to avoid the problems associated with locks, but it is weaker than previous nonblocking properties—specifically lock-freedom and wait-freedom—allowing greater flexibility in the design of efficient implementations. Obstruction-freedom admits substantially simpler implementations, and we believe that in practice it provides the benefits of wait-free and lock-free implementations.

To illustrate the benefits of obstruction-freedom, we present two obstruction-free CAS-based implementations of double-ended queues (deques); the first is implemented on a linear array, the second on a circular array. To our knowledge, all previous nonblocking deque implementations are based on unrealistic assumptions about hardware support for synchronization, have restricted functionality, or have operations that interfere with operations at the opposite end of the deque even when the deque has many elements in it. Our obstruction-free implementations have none of these drawbacks, and thus suggest that it is much easier to design obstruction-free implementations than lock-free and wait-free ones. We also briefly discuss other obstruction-free data structures and operations that we have implemented.

1. Introduction

The traditional way to implement shared data structures is to use mutual exclusion (locks) to ensure that concurrent operations do not interfere with one another. Locking has a number of disadvantages with respect to software engineering, fault-tolerance, and scalability (see [8]). In response, researchers have investigated a variety of alternative synchronization techniques that do not employ mutual exclusion. A synchronization technique is *wait-free* if it ensures that every thread will continue to make progress in the face

of arbitrary delay (or even failure) of other threads. It is *lock-free* if it ensures only that some thread always makes progress. While wait-free synchronization is the ideal behavior (thread starvation is unacceptable), lock-free synchronization is often good enough for practical purposes (as long as starvation, while possible in principle, never happens in practice).

The synchronization primitives provided by most modern architectures, such as *compare-and-swap* (CAS) or *load-locked/store-conditional* (LL/SC) are powerful enough to achieve wait-free (or lock-free) implementations of any linearizable data object [9]. Nevertheless, with a few exceptions (such as queues [16]), wait-free and lock-free data structures are rarely used in practice. The underlying problem is that conventional synchronization primitives such as CAS and LL/SC are an awkward match for lock-free synchronization. These primitives lend themselves most naturally to *optimistic* synchronization, which works only in the absence of synchronization conflicts. For example, the natural way to use CAS for synchronization is to read a value v from an address a , perform a multistep computation to derive a new value w , and then to call CAS to reset the value of a from v to w . The CAS is successful if the value at a has not been changed in the meantime. Progress guarantees typically rely on complex and inefficient “helping” mechanisms, that pose a substantial barrier to the wider use of lock-free synchronization.

In this paper, we propose an alternative nonblocking condition. A synchronization technique is *obstruction-free* if it guarantees progress for any thread that eventually executes in isolation. Even though other threads may be in the midst of executing operations, a thread is considered to execute in isolation as long as the other threads do not take any steps. (Pragmatically, it is enough for the thread to run long enough without encountering a synchronization conflict from a concurrent thread.) Like the wait-free and lock-free conditions, obstruction-free synchronization ensures that no thread can be blocked by delays or failures of other threads. This property is weaker than lock-free synchronization, because it does not guarantee progress when

two or more conflicting threads are executing concurrently.

The most radical way in which our approach of implementing obstruction-free algorithms differs from the usual approach of implementing their lock-free and wait-free counterparts is that we think that ensuring progress should be considered a problem of engineering, not of mathematics. We believe that commingling correctness and progress has inadvertently resulted in unnecessarily inefficient and conceptually complex algorithms, creating a barrier to widespread acceptance of nonblocking forms of synchronization. We believe that a clean separation between the two concerns promises simpler, more efficient, and more effective algorithms.

To support our case, we have implemented several obstruction-free shared data structures that display properties not yet achieved by comparable lock-free implementations. In this paper, we present two obstruction-free double-ended queue (deque) implementations. Elsewhere [11], we describe a software transactional memory implementation used to construct an obstruction-free red-black tree [5]. To our knowledge, there are no lock-free implementations of any data structure as complicated as a red-black tree.

As an aside, we note that there is no “obstruction-free hierarchy” comparable to the “wait-free consensus hierarchy”: One can solve obstruction-free consensus using only read/write registers by derandomizing randomized wait-free consensus algorithms such as the one in [4].

Because obstruction-freedom does not guarantee progress in the presence of contention, we need to provide some mechanism to reduce the contention so that progress is achieved. However, lock-free and wait-free implementations typically also require such mechanisms to get satisfactory performance. We can use these same mechanisms with obstruction-free implementations, as we discuss below. Because obstruction-freedom guarantees safety regardless of the contention, we can change mechanisms, even dynamically, without changing the underlying nonblocking implementation.

One simple and well-known method to reduce contention is for operations to “back off” when they encounter interference by waiting for some time before retrying. Various choices for how long to wait are possible; randomized exponential backoff is one scheme that is effective in many contexts. Other approaches to reducing contention include queuing and timestamping approaches, in which threads agree amongst themselves to “wait” for each other to finish. While simplistic applications of these ideas would give rise to some of the same problems that the use of locks does, we have much more freedom in designing sophisticated approaches for contention control than when using locks, because correctness is not jeopardized by interrupting an operation at any time and allowing another operation to continue execution.

In fact, it is possible to design contention management mechanisms that guarantee progress to every operation that takes enough steps, provided the system satisfies some very weak (and reasonable) assumptions. Thus, the strong progress properties of wait-free implementations can be achieved in practice by combining obstruction-free implementations with appropriate contention managers. In scenarios in which contention between operations is rare, we will benefit from the simple and efficient obstruction-free designs; the more heavy-weight contention resolution mechanisms will rarely be invoked. In contrast, in most lock-free and wait-free implementations, the mechanisms that are used to ensure the respective progress properties impose significant overhead *even in the absence of contention*. A study of the performance of various contention managers in practice, and tradeoffs between system assumptions, progress guarantees and performance is beyond the scope of this paper.

In some contexts, explicit contention reduction mechanisms may even be unnecessary. For example, in a uniprocessor where threads are scheduled by time slice, relatively short obstruction-free operations will be guaranteed to run alone for long enough to complete. Similarly, in priority-scheduled uniprocessors, an operation runs in isolation unless it is preempted by a higher priority operation. (As an aside, it has been shown previously that the consensus hierarchy collapses in such systems [2, 17]. However, in these results, *correctness*, as well as progress, depends on the system assumptions.)

In Section 2, we discuss previous work on nonblocking deques. Section 3 presents a simple obstruction-free deque implementation on a linear array, and Section 4 extends this algorithm to circular arrays. A detailed formal proof for the extended algorithm is given in a full version of the paper [10]. We conclude in Section 5.

2. Related Work on Nonblocking Deques

In this section, we briefly summarize related work on nonblocking deques. Double-ended queues (deques) are formally defined in [6, 13]. Informally, deques generalize FIFO queues and LIFO stacks by supporting a sequence of values and operations for adding (pushing) a value to or removing (popping) a value from either end. Thus, implementing a shared deque combines all of the intricacies of implementing queues and stacks.

Arora, *et al.* proposed a limited-functionality CAS-based lock-free deque implementation [3]. Their deque allows only one process to access one end, and only pop operations to be done on the other. Thus, they did not face the difficult problem of concurrent pushes and pops on the same end of the deque. They further simplified the problem by allowing some concurrent operations to simply abort and report

failure.

Greenwald proposed two lock-free deque implementations [7]. Both implementations depend on a hardware DCAS (double compare-and-swap) instruction, which is not widely supported in practice, and one of them does not support noninterfering concurrent operations at opposite ends of the deque. Various members of our group have also proposed several lock-free deque implementations that depend on DCAS [1, 6, 14].

Michael proposed a simple and efficient lock-free, CAS-based deque implementation [15]. However, the technique used by this algorithm fundamentally causes all operations to interfere with each other. Therefore, it offers no insight into designing scalable nonblocking data structures in which noninterfering operations can proceed in parallel.

3. Obstruction-Free Deque Implementation

In this section we present our array-based obstruction-free deque implementation. This algorithm is extremely simple, but it is not a real deque in the sense that it does not really generalize queues: if we only push on one end and pop from the other, we will exhaust the space in the array and will not be able to push any more items. In the next section, we show how to extend the algorithm to “wrap around” in the array in order to overcome this problem.

The data declarations and right-side push and pop operations are shown in Figure 1; the left-side operations are symmetric with the right-side ones.

We assume the existence of two special “null” values LN and RN (left null and right null) that are never pushed onto the deque. We use an array A to store the current state of the deque. The deque can contain up to MAX values, and the array is of size MAX+2 to accommodate a leftmost location that always contains LN and a rightmost location that always contains RN. (These extra locations are not strictly necessary, but they simplify the code.) Our algorithm maintains the invariant that the sequence of values in $A[0].val \dots A[MAX+1].val$ always consists of at least one LN, followed by zero or more data values, followed by at least one RN. The array can be initialized any way that satisfies this invariant. To simplify our presentation, we assume the existence of a function `oracle()`, which accepts a parameter `left` or `right` and returns an array index. The intuition is that this function attempts to return the index of the leftmost RN value in A when invoked with the parameter `right`, and attempts to return the index of the rightmost LN value in A when invoked with the parameter `left`. The algorithm is linearizable [12] even if `oracle` can be incorrect (we assume that it at least always returns a value between 1 and MAX+1, inclusive, when invoked with the parameter `right` and always returns a value between 0 and MAX, inclusive, when invoked with the pa-

rameter `left`; clearly it is trivial to implement a function that satisfies this property). Stronger properties of the oracle are required to prove obstruction-freedom; we discuss these properties and how they can be achieved later.

As explained in more detail below, we attach version numbers to each value in order to prevent concurrent operations that potentially interfere from doing so. The version numbers are updated atomically with the values using a compare-and-swap (CAS) instruction.¹ As usual with version numbers, we assume that sufficient bits are allocated for the version numbers to ensure that they cannot “wrap around” during the short interval in which one process executes a single iteration of a short loop in our algorithm.

The reason our obstruction-free deque implementation is so simple (and the reason we believe obstruction-free implementations in general will be significantly simpler than their lock-free and wait-free counterparts) is that there is no progress requirement when any interference is detected. Thus, provided we maintain basic invariants, we can simply retry when we detect interference. In our deque implementation, data values are changed only at the linearization point of successful push and pop operations. To prevent concurrent operations from interfering with each other, we increment version numbers of adjacent locations (without changing their associated data values). As a result of this technique, two concurrent operations can each cause the other to retry: this explains why our implementation is so simple, and also why it is not lock-free. To make this idea more concrete, we describe our implementation in more detail below.

The basic idea behind our algorithm is that a `rightpush(v)` operation changes the leftmost RN value to v, and a `rightpop()` operation changes the rightmost data value to RN and returns that value (the left-side operations are symmetric, so we do not discuss them further except when dealing with interactions between left- and right-side operations). Each `rightpush(v)` operation that successfully pushes a data value (as opposed to returning “full”) is linearized to the point at which it changes an RN value to v. Similarly, each `rightpop` operation that returns a value v (as opposed to returning “empty”) is linearized to the point at which it changes the `val` field of some array location from v to RN. Furthermore, the `val` field of an array location does not change unless an operation is linearized as discussed above. The `rightpush` operation returns “full” only if it observes a non-RN value in $A[MAX].val$. Given these observations, it is easy to see that our algorithm is linearizable if we believe the following

¹A `CAS(a, e, n)` instruction takes three parameters: an address `a`, an expected value `e`, and a new value `n`. If the value currently stored at address `a` matches the expected value `e`, then CAS stores the new value `n` at address `a` and returns `true`; we say that the CAS *succeeds* in this case. Otherwise, CAS returns `false` and does not modify the memory; we say that the CAS *fails* in this case.

```

type element = record val: valtype; ctr: int end

A: array[0..MAX+1] of element initially there is some k in [0,MAX]
                                         such that A[i] = <LN,0> for all i in [0,k]
                                         and A[i] = <RN,0> for all i in [k+1,MAX+1].
```

```

rightpush(v)
RH0: while (true) {
RH1:   k := oracle(right);                                // v is not RN or LN
RH2:   prev := A[k-1];                                     // find index of leftmost RN
RH3:   cur := A[k];                                       // read (supposed) rightmost non-RN value
RH4:   if (prev.val != RN and cur.val = RN) {              // read (supposed) leftmost RN value
RH5:     if (k = MAX+1) return "full";                      // oracle is right
RH6:     if CAS(&A[k-1],prev,<prev.val,prev.ctr+1>) // A[MAX] != RN
RH7:     if CAS(&A[k],cur,<v,cur.ctr+1>)                // try to bump up prev.ctr
RH8:     return "ok";                                      // try to push new value
RH9:   } // end if (prev.val != RN and cur.val = RN)      // it worked!
} // end while
```

```

rightpop()
RP0: while (true) {
RP1:   k := oracle(right);                                // keep trying till return val or empty
RP2:   cur := A[k-1];                                     // find index of leftmost RN
RP3:   next := A[k];                                      // read (supposed) value to be popped
RP4:   if (cur.val != RN and next.val = RN) {             // read (supposed) leftmost RN
RP5:     if (cur.val = LN and A[k-1] = cur);               // oracle is right
RP6:     return "empty";                                   // adjacent LN and RN
RP7:     if CAS(&A[k],next,<RN,next.ctr+1>)           // try to bump up next.ctr
RP8:     if CAS(&A[k-1],cur,<RN,cur.ctr+1>)           // try to remove value
RP9:     return cur.val;                                  // it worked; return removed value
} // end if (cur.val != RN and next.val = RN)
} // end while
```

Figure 1. Obstruction-free deque implementation: Data declarations and right-side operations

three claims (and their symmetric counterparts):

- At the moment that line RH7 of a `rightpush(v)` operation successfully changes `A[k].val` for some `k` from RN to `v`, `A[k-1].val` contains a non-RN value (i.e., either a data value or LN).
- At the moment that line RP8 of the `rightpop` operation successfully changes `A[k-1].val` for some `k` from some value `v` to RN, `A[k].val` contains RN.
- If a `rightpop` operation returns “empty”, then at the moment it executed line RP3, `A[k].val=RN` and `A[k-1].val=LN` held for some `k`.

Using the above observations and claims, a proof by simulation to an abstract deque in an array of size MAX is straightforward. Below we briefly explain the synchronization techniques that we use to ensure that the above claims hold. The techniques all exploit the version numbers in the array locations.

The empty case (the third claim above) is the simplest: `rightpop` returns “empty” only if it reads the same value from `A[k-1]` at lines RP2 and RP5. Because every CAS that modifies an array location increments that location’s version number, it follows that `A[k-1]` maintained the same value throughout this interval (recall our assumption about version numbers not wrapping around). Thus, in particular, `A[k-1].val` contained LN at the moment that line RP3 read RN in `A[k].val`.

The techniques used to guarantee the other two claims are essentially the same, so we explain only the first one. The basic idea is to check that the neighbouring location (i.e., `A[k-1]`) contains the appropriate value (line RH2; see also line RH4), and to increment its version number (without changing its value; line RH6) between reading the location to be changed (line RH3) and attempting to change it (line RH7). If any of the attempts to change a location fail, then we have encountered some interference, so we can simply restart. Otherwise, it can be shown easily that the neighbouring location did not change to RN between

the time it was read (line RH2) and the time the location to be changed is changed (line RH7). The reason is that a `rightpop` operation—the only operation that changes locations to RN—that was attempting to change the neighbouring location to RN would increment the version number of the location the `rightpush` operation is trying to modify, so one of the operations would cause the other to retry.

3.1. Oracle Implementations

The requirements for the `oracle` function assumed in the previous section are quite weak, and therefore a number of implementations are possible. We first describe the requirements, and then outline some possible implementations. For linearizability, the only requirement on the oracle is that it always returns an index from the appropriate range depending on its parameter as stated earlier; satisfying this requirement is trivial. However, to guarantee obstruction-freedom, we require that the oracle is *eventually accurate if repeatedly invoked in the absence of interference*. By “accurate”, we mean that it returns the index of the leftmost RN when invoked with `right`, and the index of the rightmost LN when invoked with `left`. It is easy to see that if any of the operations executes an entire loop iteration in isolation, and the `oracle` function returns the index specified above, then the operation completes in that iteration. Because the oracle has no obligation (except for the trivial range constraint) in the case that it encounters interference, we have plenty of flexibility in implementing it. One simple and correct implementation is to search the array linearly from one end looking for the appropriate value. Depending on the maximum deque size, however, this solution might be very inefficient. One can imagine several alternatives to avoid this exhaustive search. For example, we can maintain “hints” for the left and right ends, with the goal of keeping the hints approximately accurate; then we could read those hints, and search from the indicated array position (we’ll always be able to tell which direction to search using the values we read). Because these hints do not have to be perfectly accurate at all times, we can choose various ways to update them. For example, if we use CAS to update the hints, we can prevent slow processes from writing out-of-date values to hints, and therefore keep hints almost accurate all the time. It may also be useful to loosen the accuracy of the hints, thereby synchronizing on them less often. In particular, we might consider only updating the hint when it is pointing to a location that resides in a different cache line than the location that really contains the leftmost RN for example, as in this case the cost of the inaccurate hint would be much higher.

4. Extension to Circular Arrays

In this section, we show how to extend the algorithm in the previous section to allow the deque to “wrap around” the array, so that the array appears to be circular. In other words, $A[0]$ is “immediately to the right” of $A[MAX+1]$. As before, we maintain at least two null entries in the array: we use the array $A[0..MAX+1]$ for a deque with at most MAX elements. The array can be initialized arbitrarily provided it satisfies the main invariant for the algorithm, stated below. One option is to use the initial conditions for the algorithm in the previous section.

We now describe the new aspects of the algorithm. Code for the right-side operations of the wrap-around deque implementation are shown in Figure 2. The left-side operations are symmetric, and we do not discuss them further except as they interact with the right-side operations. All arithmetic on array indices is done modulo MAX+2.

There are two main differences between this algorithm and the one in the previous section. First, it is more difficult to tell whether the deque is full; we must determine that there are exactly two null entries. Second, `rightpush` operations may encounter LN values as they “consume” the RN values and wrap around the array (similarly, `leftpush` operations may encounter RN values). We handle this second problem by enabling a `rightpush` operation to “convert” LN values into RN values. This conversion uses an extra null value, which we denote DN, for “dummy null”. We assume that LN, RN and DN are never pushed onto the deque.

Because the array is circular, the algorithm maintains the following invariants instead of the simpler invariant maintained by the algorithm in the previous section:

- All null values are in a contiguous sequence of locations in the array. (Recall that the array is circular, so the sequence can wrap around the array.)
- The sequence of null values consists of zero or more RN values, followed by zero or one DN value, followed by zero or more LN values.
- There are at least two different types of null values in the sequence of null values.

Thus, there is always at least one LN or DN entry, and at least one RN or DN entry.

Instead of invoking `oracle(right)` directly, the push and pop operations invoke a new auxiliary procedure, `rightcheckedoracle`. In addition to an array index k , `rightcheckedoracle` returns `left` and `right`, the contents it last saw in $A[k-1]$ and $A[k]$ respectively. It guarantees `right.val = RN` and `left.val != RN`. Thus, if it runs in isolation, `rightcheckedoracle` always returns the correct index, together with contents of the

```

/* Returns k, left, right, where left = A[k-1] at some time t, and right = A[k]
   at some time t' > t during the execution, with left.val != RN and right.val = RN.
*/
rightcheckedoracle()
R00: while (true) {
R01:   k := oracle(right);
R02:   left := A[k-1];                                // order important for check
R03:   right := A[k];                                //      for empty in rightpop
R04:   if (right.val = RN and left.val != RN)        // correct oracle
R05:     return k, left, right;
R06:   if (right.val = DN and !(left.val in {RN,DN})) // correct oracle, but no RNs
R07:     if CAS(&A[k-1], left, <left.val, left.ctr+1>)
R08:       if CAS(&A[k], right, <RN, right.ctr+1>)    // DN -> RN
R09:       return k, <left.val, left.ctr+1>, <RN, right.ctr+1>;
} // end while

rightpush(v)                                         // !(v in {LN,RN,DN})
RH0: while (true) {
RH1:   k, prev, cur := rightcheckedoracle();          // cur.val = RN and prev.val != RN
RH2:   next := A[k+1];
RH3:   if (next.val = RN)
RH4:     if CAS(&A[k-1], prev, <prev.val, prev.ctr+1>)
RH5:       if CAS(&A[k], cur, <v, cur.ctr+1>)         // RN -> v
RH6:       return "ok";
RH7:   if (next.val = LN)
RH8:     if CAS(&A[k], cur, <RN, cur.ctr+1>)
RH9:       CAS(&A[k+1], next, <DN, next.ctr+1>);      // LN -> DN
RH10:  if (next.val = DN) {
RH11:    nextnext := A[k+2];
RH12:    if !(nextnext.val in {RN,LN,DN})
RH13:      if (A[k-1] = prev)
RH14:        if (A[k] = cur) return "full";
RH15:    if (nextnext.val = LN)
RH16:      if CAS(&A[k+2], nextnext, <nextnext.val, nextnext.ctr+1>)
RH17:        CAS(&A[k+1], next, <RN, next.ctr+1>);      // DN -> RN
} // end if (next.val = DN)
} //end while

rightpop()
RP0: while (true) {
RP1:   k, cur, next := rightcheckedoracle();           // next.val = RN and cur.val != RN
RP2:   if (cur.val in {LN,DN} and A[k-1] = cur)        // depends on order of R02 & R03.
RP3:     return "empty";
RP4:   if CAS(&A[k], next, <RN, next.ctr+1>)
RP5:     if CAS(&A[k-1], cur, <RN, cur.ctr+1>)          // v -> RN
RP6:     return cur.val;
} // end while

```

Figure 2. Wraparound deque implementation: Right-side operations.

appropriate array entries that prove that the index is correct. If no RN entry exists, then by the third invariant above, there is a DN entry and an LN entry; `rightcheckedoracle` attempts to convert the DN into an RN before returning.

Other than calling `rightcheckedoracle` instead of `oracle(right)` (which also eliminates the need to read and check the `cur` and `next` values again), the only change in the `rightpop` operation is that, in checking whether the deque is empty, `cur.val` may be either LN or DN, because there may be no LN entries.

Because the array is circular, a `rightpush` operation cannot determine whether the array is full by checking whether the returned index is at the end of the array. Instead, it ensures that there is space in the array by checking that $A[k+1].val = \text{RN}$. In that case, by the third invariant above, there are at least two null entries other than $A[k]$ (which also contains RN), so the deque is not full. Otherwise, `rightpush` first attempts to convert $A[k]$ into an RN entry. We discuss how this conversion is accomplished below.

When a `rightpush` operation finds only one RN entry, it tries to convert the next null entry—we know there is one by the third invariant above—into an RN. If the next null entry is an LN entry, then `rightpush` first attempts to convert it into a DN entry. When doing this, `rightpush` checks that `cur.val = RN`, which ensures there is at most one DN entry, as required by the second invariant above. If the next null entry is a DN entry, `rightpush` will try to convert it into an RN entry, but only if the entry to the right of the one being converted (the `nextnext` entry) is an LN entry. In this case, it first increments the version number of the `nextnext` entry, ensuring the failure of any concurrent `leftpush` operation trying to push a value into that entry. If the `nextnext` entry is a deque value, then the `rightpush` operation checks whether the right end of the deque is still at k (by rereading $A[k-1]$ and $A[k]$), and if so, the deque is full. If not, or if the `nextnext` entry is either an RN or DN entry, then some other operation is concurrent with the `rightpush`, and the `rightpush` operation retries.

Assuming the invariants above, it is easy to see that this new algorithm is linearizable in exactly the same way as the algorithm in the previous section, except that a `rightpush` operation that returns “full” linearizes at the point that `nextnext` is read (line RH11). Because we subsequently confirm (line RH13) that $A[k-1]$ and $A[k]$ have not changed since they were last read, we know the deque extends from $A[k+2]$ to $A[k-1]$ (with $A[k-1]$ as its rightmost value), so that $A[k]$ and $A[k+1]$ are the only nonnull entries, and thus, the deque is full.

Proving that the invariants above are maintained by the new algorithm is nontrivial, and we defer a complete proof to the full paper [10]. The main difficulty is verifying that

when a `rightpush` actually pushes the new value onto the deque (line RH5), either the `next` entry is an RN entry, or it is a DN entry and the `nextnext` entry is an LN entry. This is necessary to ensure that after the push, there are still at least two null entries, one of which is an RN or DN entry. One key to the proof is to note that the value of an entry is changed only by lines RO8, RH5, RH9, RH17, RP5, and their counterparts in the left-side operations. Furthermore, these lines only change an entry if the entry has not changed since it was most recently read. These lines are annotated in Figure 2 with how they change the value of the entry.

Time complexity The obvious measure of the time complexity of an obstruction-free algorithm (without regard to the particular contention manager and system assumptions) is the worst-case number of steps that an operation must take in isolation in order to be guaranteed to complete. For our algorithms, this is a constant plus the obstruction-free time complexity of the particular oracle implementation used.

5. Concluding Remarks

We have introduced obstruction-freedom—a new non-blocking condition for shared data structures that weakens the progress requirements of traditional nonblocking conditions, and as a result admits solutions that are significantly simpler and more efficient in the typical case of low contention. We have demonstrated the merits of obstruction-freedom by showing how to implement an obstruction-free double-ended queue that has better properties than any previous nonblocking deque implementation of which we are aware. We are also exploring other obstruction-free algorithms and techniques. Based on our progress to date, we are convinced that obstruction-freedom is a significant breakthrough in the search for scalable and efficient non-blocking data structures. Apart from ongoing work on other obstruction-free algorithms, an important part of our work is in investigating the use of various mechanisms to manage contention in order to allow obstruction-free implementations to make progress even under contention. There has been much work on similar issues, for example in the context of databases. We are working on evaluating existing and new schemes for this purpose. The beauty of obstruction-freedom is that we can modify and experiment with the contention management mechanisms without needing to modify (and therefore reverify) the underlying non-blocking algorithm. In contrast, work on different mechanisms for guaranteeing progress in the context of lock-free and wait-free algorithms has been hampered by the fact that modifications to the “helping” mechanisms has generally required the proofs for the entire algorithm to be done again.

References

- [1] O. Ageson, D. Detlefs, C. Flood, A. Garthwaite, P. Martin, M. Moir, N. Shavit, and G. Steele. DCAS-based concurrent deques. *Theory of Computing Systems*, 2003. To appear. A preliminary version appeared in the Proceedings of the 12th ACM Symposium on Parallel Algorithms and Architectures.
- [2] J. Anderson and M. Moir. Wait-free synchronization in multiprogrammed systems: Integrating priority-based and quantum-based scheduling. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing*, pages 123–132, 1999.
- [3] N. S. Arora, B. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 119–129, 1998.
- [4] J. Aspnes and M. Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 11(3):441–461, 1990.
- [5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. McGraw Hill, 1st edition, 1989.
- [6] D. Detlefs, C. Flood, A. Garthwaite, P. Martin, N. Shavit, and G. Steele. Even better DCAS-based concurrent deques. In *Proceedings of the 14th International Conference on Distributed Computing*, pages 59–73, 2000.
- [7] M. Greenwald. *Non-Blocking Synchronization and System Design*. PhD thesis, Stanford University Technical Report STAN-CS-TR-99-1624, Palo Alto, CA, August 1999.
- [8] M. Greenwald and D. Cheriton. The synergy between non-blocking synchronization and operating system structure. In *Proceedings of the Second Symposium on Operating System Design and Implementation*, pages 123–136, 1996.
- [9] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, 1993.
- [10] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In preparation, 2003.
- [11] M. Herlihy, V. Luchangco, and M. Moir. Software transactional memory for supporting dynamic data structures. In preparation, 2003.
- [12] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.
- [13] D. E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*. Addison-Wesley, 1968.
- [14] P. Martin, M. Moir, and G. Steele. DCAS-based concurrent deques supporting bulk allocation. Technical Report TR-2002-111, Sun Microsystems Laboratories, 2002.
- [15] M. Michael. Dynamic lock-free deques using single-address, double-word cas. Technical report, IBM TJ Watson Research Center, January 2002.
- [16] M. Michael and M. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th Annual ACM Symposium on the Principles of Distributed Computing*, pages 267–276, 1996.
- [17] S. Ramamurthy, M. Moir, and J. Anderson. Real-time object sharing with minimal system support. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, pages 233–242, 1996.

Bringing Practical Lock-Free Synchronization to 64-Bit Applications

Simon Doherty

School of Mathematical and Computing Sciences
Victoria University
Wellington, New Zealand
simon.doherty@mcs.vuw.ac.nz

Victor Luchangco

Sun Microsystems Laboratories
1 Network Drive
Burlington, MA 01803, USA
victor.luchangco@sun.com

Maurice Herlihy

Department of Computer Science
Brown University
Providence, RI 02912, USA
mph@cs.brown.edu

Mark Moir

Sun Microsystems Laboratories
1 Network Drive
Burlington, MA 01803, USA
mark.moir@sun.com

ABSTRACT

Many lock-free data structures in the literature exploit techniques that are possible only because state-of-the-art 64-bit processors are still running 32-bit operating systems and applications. As software catches up to hardware, “64-bit-clean” lock-free data structures, which cannot use such techniques, are needed.

We present several 64-bit-clean lock-free implementations: *load-linked/store-conditional* variables of arbitrary size, a FIFO queue, and a freelist. In addition to being portable to 64-bit software, our implementations also improve on previous ones in that they are space-adaptive and do not require knowledge of the number of threads that will access them.

Categories and Subject Descriptors

D.1.3 [Programming Techniques]: Concurrent Programming; D.4.2 [Operating Systems]: Storage Management; E.1 [Data]: Data Structures

General Terms

Algorithms, Theory

Keywords

Multiprocessors, 64-bit architectures, 64-bit-clean software, nonblocking synchronization, lock-free, memory management, population-oblivious, space-adaptive, compare-and-swap (CAS), load-linked/store-conditional (LL/SC), queues, freelists

1. INTRODUCTION

For more than a decade, 64-bit architectures have been available [14, 16, 18, 21]. These architectures support 64-bit addresses, allowing direct access to huge virtual address

spaces [3]. They also support atomic access to 64-bit quantities using synchronization primitives such as compare-and-swap (CAS) and the load-linked/store conditional (LL/SC) pair, which provide powerful tools for implementing lock-free data structures.

Predictably, operating systems and application software that exploit these 64-bit capabilities have been slower to emerge. Thus, many important 32-bit operating systems and applications are still in common use, and most 64-bit architectures support them. As a result, for a period of several years, techniques that use 64-bit synchronization primitives to atomically manipulate 32-bit pointers together with other information, such as version numbers, have been widely applicable. Many practical lock-free data structures exploit such techniques (e.g., [13, 19]).

The increasing prevalence of 64-bit operating systems and applications (in which pointers are 64 bits) signals the end of this convenient era: *64-bit-clean* lock-free data structures that do not require synchronization primitives that can atomically manipulate a pointer and a version number are increasingly important.

We present 64-bit-clean¹ implementations of several important lock-free data structures: arbitrary-size variables supporting LL and SC operations, FIFO queues, and freelists.

Our implementations are based on 64-bit CAS, but it is straightforward to modify them for use in architectures that support LL/SC instead of CAS [15]. Our LL/SC implementation is useful even in architectures that provide LL/SC in hardware, because it eliminates numerous restrictions on the size of variables accessed by LL/SC and the way in which they are used. For example, in some architectures, the program must perform only register operations between an LL and the following SC; no such restriction is imposed by our implementation. Our results therefore will help programmers to develop portable code, because they can ignore the different restrictions imposed by different architectures on the use of

¹Our techniques target any architecture that can perform CAS on a pointer-sized variable (of at least 64 bits); for concreteness, we present our techniques assuming that this size is 64 bits.

LL/SC. Furthermore, our implementations are portable between 32-bit and 64-bit applications, while many previous lock-free data structure implementations are not.

The only previous 64-bit-clean CAS-based implementation of LL/SC is due to Jayanti and Petrovic [10]. While their implementation is wait-free [5], it requires $O(mn)$ space, where m is the number of variables that support LL/SC and n is the number of threads that can access those variables; ours uses only $O(m+n)$ space. Furthermore, the implementation in [10] requires *a priori* knowledge of a fixed upper bound on the number of threads that will access an LL/SC variable. We call such implementations *population-aware* [8]. In contrast, our implementation has no such requirement; it is *population-oblivious*. However, our implementation guarantees only lock-freedom, a weaker progress guarantee than wait-freedom, but one that is generally considered adequate in practice.

Our lock-free FIFO queue implementation is the first that is 64-bit-clean, population-oblivious and *space-adaptive* (i.e., its space consumption depends only on the number of items in the queue and the number of threads currently accessing the queue). Previous lock-free FIFO queue implementations have at most one of these advantages.

A freelist manages memory resources and can be used to avoid the cost of using a general malloc/free implementation for this purpose. Previous lock-free freelists [19] are not 64-bit-clean, and can be prevented by a single thread failure from ever freeing memory blocks to the system. Ours overcomes both of these problems.

We provide some background in Section 2, and present our LL/SC implementation in detail in Section 3. In Sections 4 and 5, we explain how to adapt the techniques used in the LL/SC implementation to achieve our queue and freelist implementations. We conclude in Section 6.

2. BACKGROUND

A data structure implementation is *linearizable* [9] if for each operation, there is some point—called the *linearization point*—during the execution of that operation at which the operation appears to have taken place atomically. It is *lock-free* [5] if it guarantees that after a finite number of steps of any operation on the data structure, *some* operation completes. It is *population-oblivious* [8] if it does not depend on the number of threads that will access the data structure. Finally, it is *space-adaptive* if at all times, the space used is proportional to the size of the abstract data structure (and the number of threads currently accessing it)².

The CAS operation, defined in Figure 1, takes the address of a memory location, an expected value, and a new value. If the location contains the expected value, then the CAS atomically stores the new value into the location and returns *true*. Otherwise, the contents of the location remain unchanged, and the CAS returns *false*. We say that the CAS *succeeds* if it returns *true*, and that it *fails* if it returns *false*.

A typical way to use CAS is to read a value—call it A—from a location, and to then use CAS to attempt to change the location from A to a new value. The intent is often to ensure that the CAS succeeds only if the location’s value does not change between the read and the CAS. However, the location might change to a different value B and then back

²There are other variants of space adaptivity [8], but this simple definition suffices for this paper.

```
bool CAS(a, e, n) {
    atomically {
        if (*a == e) {
            *a = n;
            return true;
        } else
            return false;
    }
}
```

Figure 1: The CAS operation.

to A again between the read and the CAS, in which case the CAS can succeed. This phenomenon is known as the *ABA problem* [17] and is a common source of bugs in CAS-based algorithms. The problem can be avoided by storing the variable being accessed together with a version number in a CAS-able word: the version number is incremented with each modification of the variable, eliminating the ABA problem (at least in practice; see [15] for more detail). However, if the variable being modified is a 64-bit pointer, then this technique cannot be used in architectures that can perform CAS only on 64-bit variables. A key contribution of this paper is a novel solution to the ABA problem that can be used in such architectures.

The LL and SC operations are used in pairs: An SC operation is *matched* with the preceding LL operation by the same thread to the same variable; there must be such an LL operation for each SC operation, and no LL operation may match more than one SC operation. LL loads the value of a location, and SC conditionally stores a value to a location, succeeding (returning *true*) if and only if no other store to the location has occurred since the matching LL³. Thus the ABA problem does not arise when using LL and SC—instead of read and CAS—to modify a variable.

For simplicity, we require every LL to be matched. If a thread decides not to invoke a matching SC for a previous LL, it instead invokes UNLINK, which has no semantic effect on the variable. An LL operation is said to be *outstanding* from its linearization point until its matching SC or UNLINK returns. It is straightforward to eliminate the need for an explicit thread-visible UNLINK operation by having LL invoke UNLINK whenever a previous LL operation by the same thread to the same location is already outstanding.

Related work

Anderson and Moir [2] describe a wait-free implementation of a multiword LL/SC that requires $O(mn^2)$ space, where m is the number of variables and n is the number of threads that may access the variables; Jayanti and Petrovic present another that uses $O(mn)$ space [10]. These algorithms are impractical for applications that require LL/SC on many variables. In addition, they are not population-oblivious; they require n to be known in advance, a significant drawback for applications with dynamic threads. Moir [15] presents a lock-free algorithm that uses only $O(m)$ space, but his algorithm is not 64-bit-clean.

Our queue algorithm is similar to the algorithms of Valois [20] and of Michael and Scott [13]. However, those algorithms are not 64-bit-clean. Furthermore, they cannot free nodes removed from the queue for general reuse (though the

³We describe the *ideal* LL/SC semantics here. Hardware LL/SC implementations are usually weaker; in particular, they allow SC to fail even in the absence of an intervening store [15].

nodes can be reused by subsequent enqueue operations), and are therefore not space adaptive.

Herlihy et al. [7] and Michael [11] independently proposed general techniques to enable memory to be freed from lock-free data structures, and they applied these techniques to the Michael-and-Scott queue [6, 11]. However, the resulting algorithms are not population-oblivious. Although they can be made population-oblivious [8], the resulting solutions are still not space adaptive; in the worst case, they require space proportional to the number of all threads that ever access the queue.

Treiben [19] provides two freelist implementations, neither of which is 64-bit-clean. Furthermore, the first does not provide operations for expanding and contracting the freelist. Modifying the implementation to expand the freelist is straightforward, but enabling contraction is not. Treiben’s second freelist implementation does allow contraction, but a single delayed thread can prevent all contraction. In contrast, our freelist implementation is 64-bit-clean and does not allow thread delays to prevent other threads from contracting the freelist.

3. OUR LL/SC IMPLEMENTATION

With unbounded memory, it is straightforward to implement a lock-free, population-oblivious, arbitrary-sized LL/SC variable using the standard *pointer-swinging* technique: We store values in contiguous regions of memory called *nodes* and maintain a pointer to the *current node*. LL simply reads the pointer to the current node and returns the contents of the node it points to. SC allocates a new node, initializes it with the value to be stored, and then uses CAS to attempt to “swing” the pointer from the previously current node to the new one; the SC succeeds if and only if the CAS succeeds. If every SC uses a new node, then the CAS in an SC succeeds if and only if there is no change to the pointer between the CAS and the read in the preceding LL. This technique is well-known and used in systems that use garbage collection to provide the illusion of unbounded memory (see the JSR-166 library [1], for example).

Our implementation builds on this simple idea, but is complicated by the need to free and reuse nodes in order to bound memory consumption. Reclaiming nodes too late results in excessive space overhead. However, reclaiming them too soon leads to other problems. First, an LL reading the contents of a node might in fact read part or all of a value stored by an SC that is reusing the node. Second, the CAS might succeed despite changes since the previous read because of the recycling of a node: the ABA problem. Our implementation maintains additional information that allows nodes to be reclaimed promptly enough to bound its space complexity, but avoids the problems above by preventing premature reclamation.

We first describe the basic implementation assuming that nodes are never reclaimed prematurely and ignoring those parts related only to node reclamation (Section 3.1). Then we describe how nodes are reclaimed, argue why they are never reclaimed prematurely, and analyze the space complexity of the algorithm (Section 3.2). To simplify the code and discussion, the implementation presented here restricts threads to have at most one outstanding LL operation at a time. However, extending it to allow each thread to have multiple outstanding LL operations is straightforward.

```

typedef struct {
    Node *ptr0, *ptr1;
    EntryTag entry;
} LLSCvar;

typedef struct {
    Data d;
    Node *pred;
    ExitTag exit;
} Node;

typedef struct {
    int ver;
    int count;
} EntryTag;

typedef struct {
    int count;
    bool n1C;
    bool n1P;
} ExitTag;

```

Figure 2: Data types used in the LL/SC algorithm. The EntryTag and ExitTag types fit into 64 bits, so can be atomically accessed using cas.

3.1 The basic implementation

Rather than storing a pointer to the current node in a single location, we alternate between two locations, `ptr0` and `ptr1`. At any time, one of these pointers is the *current pointer*—it points to the current node—and the other is the *noncurrent pointer*. Which pointer is current is determined by a version number `entry.ver`:⁴ if `entry.ver` is even, then `ptr0` is the current pointer; otherwise `ptr1` is.

An LL operation determines the current node and returns the data value it contains. An SC operation attempts to change the noncurrent pointer to point to a new node—initialized with the data value to be stored—and then increments `entry.ver`, making this pointer current. If the SC successfully installs the new pointer but is delayed before incrementing the version number, then another thread can “help” by incrementing the version number on its behalf. The successful SC is linearized at the point at which the version number is incremented (either by the thread executing that SC or by a helping thread), causing the newly installed node to become current.

Our algorithm guarantees the following *alternating property*: In any execution, the sequence of events that modify `ptr0` and `ptr1` and `entry.ver` strictly alternates between

- modifying the noncurrent pointer to point to the new node of an SC operation; and
- incrementing `entry.ver`, thereby causing the current pointer to become noncurrent and vice versa.

With this property, it is easy to see that the algorithm described above provides the correct semantics: neither `ptr0` nor `ptr1` ever changes while it is the current pointer; the noncurrent pointer is changed exactly once (by a successful SC operation) between consecutive increments of `entry.ver`; and each time we increment the version number, and therefore linearize a successful SC (the unique SC that changed the noncurrent pointer since the previous time the version number was incremented), the new node installed by the successful SC becomes the current node.

Figure 2 shows the types used in our implementation. An `LLSCvar` structure consists of three fields, `ptr0`, `ptr1` and

⁴In addition to determining which pointer is current, the version number eliminates the ABA problem in practice, provided the version number has enough bits to ensure that it does not repeat a value during the interval in which some thread executes a short code sequence. In our algorithm, we can easily allocate 32 or more bits to the version number, which we believe is sufficient. For further discussion of the number of bits required to avoid the ABA problem, see [15].

```

ptr0→d = d0
ptr0→pred = ptr1
ptr0→exit = ⟨0, false, false⟩
ptr1→exit = ⟨0, true, false⟩
entry.ver = 0
entry.count = 0

```

Figure 3: Initial state of an LL/SC location, where d_0 is the initial value of the location.

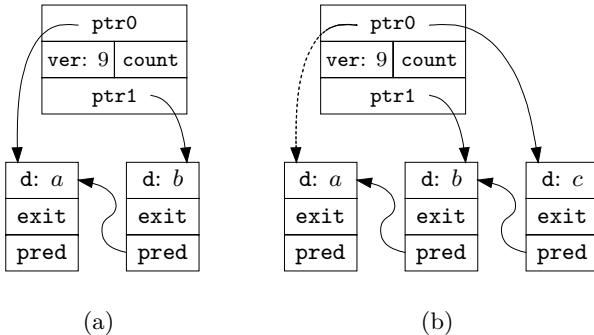


Figure 4: Two states of the LL/SC implementation. Dotted pointer indicates previous value (see text).

`entry`, each of which is 64 bits (so the CAS operation can be applied to each field, though not to the entire `LLSCvar` structure). In addition to the fields already mentioned, the `entry` field of an LL/SC variable has a `count` field, and each node has `pred` and `exit` fields. The `pred` field of each node contains a pointer to the node that was current immediately before this node. The other fields are concerned only with node reclamation, and are discussed later.

Figure 3 shows how an LL/SC variable is initialized and Figure 4 illustrates two classes of states of our algorithm. In both illustrations, `entry.ver` is odd, so `ptr1` is the current pointer and `ptr0` is the noncurrent pointer. In Figure 4(a), the noncurrent pointer points to the current node's predecessor (i.e., the node that was current before the node that is current in the figure). In Figure 4(b), the noncurrent pointer points to a new node whose `pred` field points to the current node. From a state like the one in Figure 4(a), installing a pointer to a new node whose `pred` field points to the current node into the noncurrent pointer results in a state like the one in Figure 4(b). Furthermore, from a state like the one in Figure 4(b), incrementing `entry.ver` results in a state like the one in Figure 4(a), because incrementing `entry.ver` changes its parity, thereby reversing the roles of `ptr0` and `ptr1`. The key to understanding our algorithm is to notice that it alternates between states like that in Figure 4(a) and states like that in Figure 4(b). This behavior is captured by the alternating property, which is central to the correctness proof for our algorithm.

We now present our algorithm in more detail and explain how it preserves the alternating property; we ignore for now details related to node reclamation. Pseudocode for the LL, SC and UNLINK operations is presented in Figure 5. Each thread has two persistent local variables, `mynode` and `myver`, which are set by the LL operation, and retain their values while that LL is outstanding. The `CURRENT` and `NONCURADDR` macros determine the current and noncurrent pointers based on the `ptr0` or `ptr1` fields and the `entry.ver` fields, as explained above. Specifically, if `loc→entry.ver == version`, then `CURRENT(loc,version)` gives the current pointer of `loc`, and `NONCURADDR(loc,version)` gives the address of the non-

current pointer. The RELEASE and TRANSFER procedures, the `entry.count` field, and the `exit` field and its initialization value `INIT_EXIT` are relevant only to node reclamation, as are the effects of `UNLINK`. We defer further discussion of these procedures and fields until Section 3.2.

Ignoring for now the effect of the CAS at line L5 on the `entry.count` field, we see that a thread p executing LL records `entry.ver` in its persistent local `myver` variable and the current node indicated by this value in its `mynode` variable. To ensure a consistent view of the current node and version number, LL retries if `entry.ver` changes while it determines the current node (lines L2 and L5). The LL operation is linearized at the (unique) point at which p successfully executes the CAS at line L5.

To execute an SC operation, p allocates and initializes a new node⁵ with the value to be stored, and stores the node observed as current by the previous LL in the node's `pred` field (lines S1 and S2). Then, p uses CAS to attempt to change the noncurrent pointer to point to the new node (line S4). We do not simply read the contents of the noncurrent pointer in order to determine the expected value for this CAS. If we did, two different SC operations could install new nodes in the noncurrent pointer, without the noncurrent pointer becoming current as the result of an increment of `entry.ver`. Such behavior would violate the alternating property.

To avoid this problem, we instead determine the expected value for the CAS by reading the `pred` field of the node observed as current (line S3). Recall that when a node becomes current, its `pred` field points to the node that was current immediately before it. Thus, the `pred` field of the current node is the same as the noncurrent pointer before a new node is installed. Once an SC has successfully changed the noncurrent pointer to point to a new node, no other SC can do so again before `entry.ver` is incremented. This could happen only if some thread previously saw the newly installed node as the predecessor of some node. As we explain later, our node reclamation technique precludes this possibility.

After the execution of line S4, either p 's SC has succeeded in changing the noncurrent pointer, or some other SC has. In either case, the `entry.ver` field should now be incremented in order to make the successful SC that installed a new node take effect. The CAS at line S7 ensures that the version number is incremented. (The loop at lines S6 through S8 is necessary because the CAS at line S7 may fail for reasons other than another thread having incremented `entry.ver`; this possibility is explained below.)

3.2 Memory reclamation

If nodes are never reclaimed, then values stored to `ptr0` and `ptr1` are all distinct, and it is easy to see the correctness of the algorithm as described. We now explain how our implementation reclaims and reuses nodes and why the algorithm is correct despite this. For simplicity, we defer consideration of `UNLINK` until later in this section; for now, we assume that every LL is matched by an SC.

After an LL successfully executes the CAS at line L5, it reads the contents of the node it determined to be current at lines L6 and S3. We ensure that the node is not reclaimed before this happens. Specifically, after a thread successfully executes the CAS at line L5, we ensure that the node is not

⁵If no suitable lock-free memory allocator is available, then nodes can be allocated from a freelist. The implications of this approach are described in Section 3.3.

```

Macros:
CURRENT(loc, ver)  $\equiv$  (ver%2 == 0 ? loc→ptr0 : loc→ptr1)
NONCURADDR(loc, ver)  $\equiv$  (ver%2 == 0 ? &loc→ptr1 : &loc→ptr0)
INIT_EXIT  $\equiv$  (0, false, false)

Data LL(LLSCvar *loc) {
L1. do {
L2.   EntryTag e = loc→entry;
L3.   myver = e.ver;
L4.   mynode = CURRENT(loc, e.ver);
L5. } while (!CAS(&loc→entry, e, {e.ver, e.count + 1}));
L6. return mynode→d;
}

void UNLINK(LLSCvar *loc) {
U1. while ((e = loc→entry).ver == myver)
U2.   if (CAS(&loc→entry, e, {e.ver, e.count - 1})) return;
U3. RELEASE(mynode);
}
}

bool SC(LLSCvar *loc, Data newd) {
S1. Node *new_nd = alloc(Node);
S2. new_nd→d = newd;
new_nd→pred = mynode;
new_nd→exit = INIT_EXIT;
S3. Node *pred_nd = mynode→pred;
S4. success = CAS(NONCURADDR(loc, myver), pred_nd, new_nd);
S5. if (!success) free(new_nd);
S6. while ((e = loc→entry).ver == myver)
S7.   if (CAS(&loc→entry, e, {e.ver + 1, 0}))
S8.     TRANSFER(mynode, e.count);
S9. RELEASE(mynode);
S10. return success;
}

```

Figure 5: The LL, SC, and UNLINK operations.

```

Macros:
CLEAN(exit)  $\equiv$  (exit.count == 0  $\wedge$  pre.nlC)
FREEABLE(exit)  $\equiv$  (CLEAN(exit)  $\wedge$  exit.nlP)

void RELEASE(Node *nd) {
R1. Node *pred_nd = nd→pred;
R2. do {
R3.   ExitTag pre = nd→exit;
R4.   ExitTag post = {pre.count - 1, pre.nlC, pre.nlP};
R5. } while (!CAS(&nd→exit, pre, post));
R6. if (CLEAN(post)) SETNLPRED(pred_nd);
R7. if (FREEABLE(post)) free(nd);
}
}

void TRANSFER(Node *nd, int count) {
T1. do {
T2.   ExitTag pre = nd→exit;
T3.   ExitTag post = {pre.count + count, true, pre.nlP};
T4. } while (!CAS(&nd→exit, pre, post));
}

void SETNLPRED(Node *pred_nd) {
P1. do {
P2.   ExitTag pre = pred_nd→exit;
P3.   ExitTag post = {pre.count, pre.nlC, true};
P4. } while (!CAS(&pred_nd→exit, pre, post));
P5. if (FREEABLE(post)) free(pred_nd);
}

```

Figure 6: Helper procedures for the LL/SC implementation.

reclaimed before that thread invokes RELEASE on that node at line S9. Also, to avoid the ABA problem, we ensure that a node is not reclaimed if some thread might still see it as the predecessor of another node (at line S3), and therefore use it as the expected value for the CAS at line S4.

We avoid both premature reclamation scenarios by recording information in `entry.count` and the `exit` field of each node that allows us to determine when it is safe to reclaim a node. First, we use `entry.count` to count the number of threads that successfully execute the CAS at line L5 while `entry.ver` contains a particular value. (Note that `entry.count` is reset to zero whenever `entry.ver` is incremented at line S7.) When a thread increments `entry.count`, we say the thread *pins* the node that is current at that time.

One might think that we could maintain an accurate count of the number of threads that have pinned a node and not subsequently released it by simply decrementing `entry.count` in RELEASE. However, this approach does not work because by the time a thread invokes RELEASE for a particular node, that node is no longer current, so `entry.count` is being used for a different node—the one that is now current. Therefore, we instead use a node’s `exit.count` field to count the number of threads that have released the node; this counter starts at zero and is decremented by each releasing thread (see lines R4 and R5 in Figure 6).

We use the TRANSFER procedure to reconcile the number of threads that pinned the node with the number that have since released it. TRANSFER adds the value of `entry.count` when a node is replaced as the current node to that node’s

`exit.count` field (lines S7, S8, and T1 through T4). When `exit.count` contains zero after this transfer has happened, all threads that pinned this node have since released it.

To distinguish the initial zero state of the `exit.count` field from the state in which `entry.count` has been transferred and all threads have executed RELEASE, we use a flag `nlC` in the node’s `exit` field; TRANSFER sets `exit.nlC` (see line T3) to indicate that the transfer has occurred (`nlC` stands for “no longer current”; TRANSFER is invoked by the thread that makes the node noncurrent). We say that a node with `exit.nlC` set and `exit.count == 0` is *clean* (as captured by the CLEAN macro).

For the UNLINK operation, a thread could simply invoke RELEASE, as on line U3. However, if `entry.ver` has not changed since the thread pinned a node, we can instead decrement `entry.count` (see lines U1 and U2); it is still being used to keep track of the number of threads that pinned the node pinned by the thread that invoked UNLINK.

In our algorithm as described so far, no thread accesses a clean node. However, it is not always safe to free a clean node: recall that we must also prevent a node from being reclaimed while a thread might still determine it to be the predecessor of another node. For this purpose, we use one more flag in the `exit` field called `nlP` (for “no longer predecessor”). At any time, each node is the predecessor of only one node, so we simply need to determine when that node’s `pred` field will no longer be accessed by any thread, that is, when that node is clean. A thread that makes a node clean invokes the SETNLPRED procedure to set the `nlP` flag of the

node’s predecessor (line R6). When a node is clean *and* has its `exit.nlp` flag set, as expressed by the `FREEABLE` macro, it is safe to free the node (lines R7 and P5).

Let us analyze the space requirements for an application using our implementation for LL/SC variables. Each variable requires $O(1)$ space for its **LLSCvar** structure, and has two nodes that cannot be reclaimed (the nodes pointed to by its `ptr0` and `ptr1` fields). In addition, each LL/SC sequence in progress can prevent the reclamation of three nodes: the node pinned by the thread between an LL operation and its matching SC or UNLINK, the predecessor of the pinned node, and the new node used by an SC operation. Thus, in an application with m LL/SC variables, the space used by our algorithm at any time is $O(m + k)$, where k is the number of outstanding LL operations *at that time*. In the worst case, when all n threads have outstanding LL operations, the space used is $O(m + n)$. Note that this space complexity is asymptotically optimal, and that the space used adapts to the number of threads actually accessing the LL/SC variables at any time. In particular, only $O(m)$ space is needed when no threads are accessing these variables. The only previous 64-bit-clean implementation [10] always uses $O(mn)$ space, a clear limitation in practice. Furthermore, it requires *a priori* knowledge of n ; our algorithm does not.

3.3 Optimizations and Extensions

Our LL/SC implementation can be made more efficient by observing that if `FREEABLE(post)` holds before the `CAS` on line R5 or line P4, then the `CAS` does not need to be executed; `mynode` can simply be freed because there are no threads that still have to `RELEASE` this node. Similarly, a thread that calls `TRANSFER` at line S8 will always subsequently call `RELEASE` at line S9. Therefore, we can combine the effect of the two cases in those two procedures into a single `CAS`.

It is easy to extend our implementation to allow threads to have multiple outstanding LL operations: each thread simply maintains separate `mynode` and `myver` local variables for each outstanding LL. In the resulting extension, a thread may pin several nodes simultaneously (one for each outstanding LL). The space complexity of this extension is still $O(m + k)$, but now there may be more outstanding LL operations than threads (i.e., we may have $k > n$). In the unlikely case that all n threads *simultaneously* have outstanding LL operations on all m variables, then $O(mn)$ space is used. However, this much space is used only while $O(mn)$ LL operations are outstanding. As before, if no threads are accessing the LL/SC variables, then the space consumed is $O(m)$.

We can also extend our implementation to provide an operation that “validates” the previous LL, that is, determines whether its future matching SC can still succeed. A `validate` operation simply determines whether the noncurrent pointer still points to the predecessor of the node stored in `mynode` by the LL operation. If so, a future SC can replace it with a new node, thereby ensuring its success.

If our algorithm is used with a memory allocator that is not lock-free, then neither is our LL/SC implementation. While lock-free allocators exist [4, 12], most standard allocators are not lock-free. An alternative means for achieving a lock-free implementation is to use a lock-free freelist to manage nodes. (We present a suitable freelist implementation in Section 5.) The idea is to populate the freelist with enough nodes that one is always available for an SC operation to use. The number of nodes needed depends on the number

```

void ENQUEUE(Value v) {
E1.   Node *nd = alloc(Node);
E2.   nd->v = v;
      nd->next = null;
      nd->exit = INIT_EXIT;
E3.   while (true) {
E4.     Node *tail = LL(&Tail);
E5.     nd->pred = tail;
E6.     if (CAS(&tail->next, null, nd)) {
E7.       SC(&Tail, nd);
E8.       return;
E9.     } else
E10.    SC(&Tail, tail->next);
      }
}

Value DEQUEUE() {
D1.   while (true) {
D2.     Node *head = LL(&Head);
D3.     Node *next = head->next;
D4.     if (next == null) {
D5.       UNLINK(&Head);
D6.       return null;
      }
D7.     if (SC(&Head, next)) {
D8.       Value v = next->v;
D9.       SETTOBEFREED(next);
D10.      return v;
      }
}
}

```

Figure 7: Queue operations.

of threads that simultaneously access the implemented variable. If we cannot bound this number in advance, we can resort to the standard memory allocator to increase the size of the freelist upon thread creation, and remove nodes from the freelist and free them upon thread destruction. While this approach involves locking when creating or destroying a thread, we avoid locking during the lifetime of each thread.

4. QUEUE

In this section, we describe a 64-bit-clean lock-free FIFO queue implementation that is population-oblivious and consumes space proportional only to the number of items in the queue (and the number of threads currently accessing the queue). Our queue implementation is similar in structure to that of Michael and Scott [13], but overcomes two important drawbacks. First, the implementation of [13] uses version numbers on its `Head` and `Tail` pointers, and is therefore not 64-bit-clean. Second, it cannot free nodes that have been dequeued; instead it stores them in a freelist for subsequent reuse, resulting in space consumption proportional to the historical maximum size of the queue.

Figure 7 presents our queue code. Rather than modifying the `Head` and `Tail` pointers with `CAS` and using version numbers to avoid the ABA problem (as in [13]), we use LL and SC. If we ignore memory management issues for a moment, and assume that the LL and SC operations used are the standard hardware-supported ones, then this implementation is essentially the one in [13]. To facilitate the memory management required to achieve a 64-bit-clean space-adaptive implementation, we use LL and SC operations similar to those presented in the previous section in place of the standard operations.

```

typedef struct {
    Value v;
    Node *next;
    Node *pred;
    ExitTag exit;
} Node;

typedef struct {
    int count;
    int transfersLeft;
    bool nLP;
    bool toBeFreed;
} ExitTag;

```

`INIT_EXIT` $\equiv \langle 0, 2, \text{false}, \text{false} \rangle$
`CLEAN(exit)` $\equiv (\text{exit}.count == 0 \wedge \text{exit}.transfersLeft == 0)$
`FREEABLE(exit)` $\equiv (\text{CLEAN}(\text{exit}) \wedge \text{exit}.nLP \wedge \text{exit}.toBeFreed)$

Figure 8: Modified datatypes and macros for queue algorithm.

The LL and SC operations used here differ from those in the previous section in several ways. First, because the values stored in `Head` and `Tail` are just pointers, the level of indirection used to support variables of arbitrary size in the previous section is unnecessary: we deal with node pointers directly. Thus, we embed the `exit` and `pred` fields in the queue node structure, as shown in Figure 8.

Second, SC does not allocate and initialize a new node, but rather uses the node passed to it by `ENQUEUE` or `DEQUEUE`. Nodes are allocated and initialized by `ENQUEUE`.

Third, we modify `ExitTag` to support node reclamation appropriate for the queue. In the queue implementation, a node should not be reclaimed until it has been replaced as the `Tail` node *and* it has been replaced as the `Head` node. Each of the SC operations that effect these changes must transfer a count of pinning threads to the node. To detect when both of these transfers have occurred, we replace the boolean flag `nLC` of the `ExitTag` structure in the previous section with a counter `transfersLeft`. This counter is initialized to 2 and decremented by each transfer: when the counter is zero, both transfers have occurred. The `CLEAN` macro is also modified to check whether `transfersLeft` is zero rather than whether `nLC` is set, as shown in Figure 8.

Finally, as before, we use the `exit.nLP` field to avoid the ABA problem when changing the noncurrent pointer to point to a new node on line S4. However, observe that line D8 reads a value from a node that may not be pinned by any LL operation. We must also ensure that this node is not reclaimed before this read occurs. Because only one thread (the one that changes `Head` to point to this node) reads this value, a single additional flag `toBeFreed` suffices (set on line D9 by invoking `SETTOBEFREED`). As shown in Figure 8, the `FREEABLE` macro is modified to check that the `toBeFreed` flag is also set.

These changes to the LL and SC operations necessitate modifications to the other helper procedures used in the implementation; these modifications are straightforward, and the full code for the LL, SC and other helper procedures can be found in Figure 10.

As with the LL/SC implementation in the previous section, we can avoid the overhead of a general-purpose allocator by using a freelist to store dequeued nodes for future reuse. If we know a bound on the maximum size of the queue, we can populate the freelist in advance and avoid using the general-purpose allocator at all. Otherwise, enough `ENQUEUE` operations will inevitably require us to allocate new nodes.

5. FREELIST

In this section, we describe how to adapt the queue algorithm of the previous section to implement a lock-free freelist.

```

Value * GET() {
    Node *nd = DEQUEUE();
    return (Value *)nd;
}

void PUT(Value *v) {
    SETTOBEENQUEUED((Node *)v);
}

void EXPAND() {
    Node *nd = alloc(Node);
    ENQUEUE(nd);
}

void CONTRACT() {
    Node *nd = DEQUEUE();
    if (nd != null) SETTOBEFREED(nd);
}

```

Figure 9: Freelist operations.

ist. Our freelist implementation provides four operations: `GET`, which removes a memory block from the freelist and returns a pointer to that block (or `null` if no block is available); `PUT`, which takes a pointer to a memory block and puts the block on the freelist;⁶ `EXPAND`, which allocates a new memory block and puts it on the freelist; and `CONTRACT`, which removes a block from the freelist (if one is available) and frees it. An application using the freelist must guarantee that it will not access the memory block pointed to by a pointer passed to `PUT` until it is subsequently returned by `GET`, and that any pointer passed to `PUT` was returned by some previous invocation of `GET`.

The freelist is basically a queue of nodes, and the `v` field of each node contains a memory block managed by the freelist. The `GET` operation returns a pointer to the `v` field of a node; applications should be oblivious to the presence of the other fields of the node. The `PUT` operation takes a pointer to the `v` field; from which we assume it can derive a pointer to the node containing the field.⁷

The freelist code appears in Figure 9. This code invokes `ENQUEUE` and `DEQUEUE` operations, which are similar to the corresponding operations of the previous section except that these operations take and return pointers to the `v` field of nodes rather than the values to be stored in those nodes. Because the `ENQUEUE` operation takes a node, it no longer allocates a new node.

The principal difference between the queue and freelist implementations is that the `ExitTag` type has yet another flag, `toBeEnqueued`. This extra field is necessary because when a node becomes `FREEABLE`, there are two possible actions: If the node was most recently dequeued by the `CONTRACT` operation then it should be freed, but if it was dequeued by a `GET` operation and has subsequently been passed back to a `PUT` operation, then it should be enqueued into the freelist again. Using separate `toBeFreed` and `toBeEnqueued` fields allows us to distinguish the two cases.

⁶We allow the actual placement of memory blocks on the freelist to be delayed. That is, a memory block passed to `PUT` may not actually be put on the freelist until some time after the `PUT` operation has returned. The user of the freelist may notice this discrepancy if a `GET` operation returns `null` after some `PUT` operation completes.

⁷In our code, we assume that the `v` field is placed at the beginning of the node, so we can use type casting to convert between pointers to nodes and the values they contain.

```

Node LL(LLSCvar *loc) {
    do {
        EntryTag e = loc->entry;
        myver = e.ver;
        mynode = CURRENT(loc, e.ver);
    } while (!CAS(&loc->entry, e, {e.ver, e.count + 1}));
    return mynode;
}

bool SC(LLSCvar *loc, Node nd) {
    Node *pred_nd = mynode->pred;
    success = CAS(NONCURADDR(loc, myver), pred_nd, nd);
    if (!success) free(new_nd);
    while ((e = loc->entry).ver == myver)
        if (CAS(&loc->entry, e, {e.ver + 1, 0}))
            TRANSFER(mynode, e.count);
    RELEASE(mynode);
    return success;
}

void UNLINK(LLSCvar *loc) {
    while ((e = loc->entry).ver == myver)
        if (CAS(&loc->entry, e, {e.ver, e.count - 1})) return;
    RELEASE(mynode);
}

void TRANSFER(Node *nd, int count) {
    do {
        ExitTag pre = nd->exit;
        ExitTag post = {pre.count + count, pre.transfersLeft - 1,
                        pre.nlP, pre.toBeFreed};
    } while (!CAS(&nd->exit, pre, post));
}

void RELEASE(Node *nd) {
    Node *pred_nd = nd->pred;
    do {
        ExitTag pre = nd->exit;
        ExitTag post = {pre.count - 1, pre.transfersLeft,
                        pre.nlP, pre.toBeFreed};
    } while (!CAS(&nd->exit, pre, post));
    if (CLEAN(post)) SETNLPRED(pred_nd);
    if (FREEABLE(post)) free(nd);
}

void SETNLPRED(Node *pred_nd) {
    do {
        ExitTag pre = pred_nd->exit;
        ExitTag post = {pre.count, pre.transfersLeft,
                        true, pre.toBeFreed};
    } while (!CAS(&pred_nd->exit, pre, post));
    if (FREEABLE(post)) free(pred_nd);
}

void SETTOBEFREED(Node *pred_nd) {
    do {
        ExitTag pre = pred_nd->exit;
        ExitTag post = {pre.count, pre.transfersLeft,
                        pre.nlP, true};
    } while (!CAS(&pred_nd->exit, pre, post));
    if (FREEABLE(post)) free(pred_nd);
}

```

Figure 10: Helper procedures for queue

6. CONCLUDING REMARKS

We have presented a lock-free, CAS-based, 64-bit-clean LL/SC implementation that improves on the only previous one by substantially reducing space requirements, as well as eliminating the need for advance knowledge of the number of threads that will access it. We have also presented the first lock-free 64-bit-clean FIFO queue and freelist implementations that do not require advance knowledge of the number of threads or impose a maximum size on the data structure: their space usage adapts to current requirements. All of these factors are important for portability and practicality.

The difficulty of achieving lock-free 64-bit-clean implementations of such mundane data structures strongly suggests that improved hardware support is necessary before practical lock-free data structures will be widely available. However, we do not believe that 128-bit synchronization primitives are sufficient to achieve this goal; we need synchronization primitives that allow atomic access to multiple, independent memory locations.

7. REFERENCES

- [1] Java Specification Request for Concurrent Utilities (JSR166). <http://jcp.org>.
- [2] J. Anderson and M. Moir. Universal constructions for large objects. *IEEE Transactions on Parallel and Distributed Systems*, 10(12):1317–1332, 1999.
- [3] J. Chase, M. Baker-Harvey, H. Levy, and E. Lazowska. Opal: A single address space system for 64-bit architectures (abstract). *Operating Systems Review*, 26(2):9, 1992.
- [4] D. Dice and A. Garthwaite. Mostly lock-free malloc. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management*, 2002.
- [5] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.
- [6] M. Herlihy, V. Luchangco, P. Martin, and M. Moir. Dynamic-sized lock-free data structures. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing*, page 131, July 2002.
- [7] M. Herlihy, V. Luchangco, and M. Moir. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures. In *Proceedings of 16th International Symposium on Distributed Computing*, Oct. 2002.
- [8] M. Herlihy, V. Luchangco, and M. Moir. Space- and time-adaptive nonblocking algorithms. In *Proceedings of Computing: The Australasian Theory Symposium*, 2003.
- [9] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, Nov. 1990.
- [10] P. Jayanti and S. Petrovic. Efficient and practical constructions of LL/SC variables. In *Proceedings of the 22nd Annual ACM Symposium on the Principles of Distributed Computing*, July 2003.
- [11] M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(8), Aug. 2004.

- A preliminary version appeared in *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing*, 2002.
- [12] M. Michael. Scalable lock-free dynamic memory allocation. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, June 2004.
 - [13] M. Michael and M. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *Journal of Parallel and Distributed Computing*, 51(10):1–26, 1998.
 - [14] *MIPS R4000 Microprocessor User’s Manual*. MIPS Computer Systems, Inc., 1991.
 - [15] M. Moir. Practical implementations of non-blocking synchronization primitives. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, pages 219–228, 1997.
 - [16] *PowerPC 601 RISC Microprocessor User’s Manual*. Motorola, Inc., 1993.
 - [17] S. Prakash, Y. Lee, and T. Johnson. A non-blocking algorithm for shared queues using compare-and-swap. *IEEE Transactions on Computers*, 43(5):548–559, 1994.
 - [18] R. L. Sites. *Alpha Architecture Reference Manual*. Digital Press and Prentice-Hall, 1992.
 - [19] R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, 1986.
 - [20] J. Valois. Implementing lock-free queues. In *Proceedings of the 7th International Conference on Parallel and Distributed Computing Systems*, Oct. 1994.
 - [21] D. Weaver and T. Germond. *The SPARC Architecture Manual Version 9*. PTR Prentice Hall, 1994.

Are Lock-Free Concurrent Algorithms Practically Wait-Free?

Dan Alistarh
MSR Cambridge*

Keren Censor-Hillel
Technion[†]

Nir Shavit
MIT & Tel-Aviv University[‡]

Abstract

Lock-free concurrent algorithms guarantee that *some* concurrent operation will always make progress in a finite number of steps. Yet programmers prefer to treat concurrent code as if it were *wait-free*, guaranteeing that *all* operations always make progress. Unfortunately, designing wait-free algorithms is generally a very complex task, and the resulting algorithms are not always efficient. While obtaining efficient wait-free algorithms has been a long-time goal for the theory community, most non-blocking commercial code is only lock-free.

This paper suggests a simple solution to this problem. We show that, for a large class of lock-free algorithms, under scheduling conditions which approximate those found in commercial hardware architectures, lock-free algorithms behave as if they are wait-free. In other words, programmers can keep on designing simple lock-free algorithms instead of complex wait-free ones, and in practice, they will get wait-free progress.

Our main contribution is a new way of analyzing a general class of lock-free algorithms under a *stochastic scheduler*. Our analysis relates the individual performance of processes with the global performance of the system using *Markov chain lifting* between a complex per-process chain and a simpler system progress chain. We show that lock-free algorithms are not only wait-free with probability 1, but that in fact a general subset of lock-free algorithms can be closely bounded in terms of the average number of steps required until an operation completes.

To the best of our knowledge, this is the first attempt to analyze progress conditions, typically stated in relation to a worst case adversary, in a stochastic model capturing their expected asymptotic behavior.

1 Introduction

The introduction of multicore architectures as today’s main computing platform has brought about a renewed interest in concurrent data structures and algorithms, and a considerable amount of research has focused on their modeling, design and analysis.

The behavior of concurrent algorithms is captured by *safety properties*, which guarantee their correctness, and *progress properties*, which guarantee their termination. Progress properties can be quantified using two main criteria. The first is whether the algorithm is *blocking* or *non-blocking*, that is, whether the delay of a single process will cause others to be blocked, preventing them from terminating. Algorithms that use locks are blocking, while algorithms that do not use locks are non-blocking. Most of the code in the world today is lock-based, though the fraction of code without locks is steadily growing [11].

The second progress criterion, and the one we will focus on in this paper, is whether a concurrent algorithm guarantees *minimal* or *maximal progress* [12]. Intuitively, minimal progress means that *some*

*Part of this work was performed while the author was a Postdoctoral Associate at MIT CSAIL. Email: daalista@microsoft.com.

[†]ckeren@cs.technion.ac.il. Shalon Fellow.

[‡]shanir@csail.mit.edu.

process is always guaranteed to make progress by completing its operations, while maximal progress means that *all* processes always complete all their operations.

Most non-blocking commercial code is *lock-free*, that is, provides minimal progress without using locks [6, 12]. Most blocking commercial code is *deadlock-free*, that is, provides minimal progress when using locks. Over the years, the research community has devised ingenious, technically sophisticated algorithms that provide *maximal progress*: such algorithms are either *wait-free*, i.e. provide maximal progress without using locks [9], or *starvation-free* [14], i.e. provide maximal progress when using locks. Unexpectedly, maximal progress algorithms, and wait-free algorithms in particular, are not being adopted by practitioners, despite the fact that the completion of all method calls in a program is a natural assumption that programmers implicitly make.

Recently, Herlihy and Shavit [12] suggested that perhaps the answer lies in a surprising property of lock-free algorithms: in practice, they often behave as if they were wait-free (and similarly, deadlock-free algorithms behave as if they were starvation-free). Specifically, most operations complete in a timely manner, and the impact of long worst-case executions on performance is negligible. In other words, in real systems, the scheduler that governs the threads’ behavior in long executions does not single out any particular thread in order to cause the theoretically possible bad behaviors. This raises the following question: could the choice of *wait-free* versus *lock-free* be based simply on what assumption a programmer is willing to make about the underlying scheduler, and, with the right kind of scheduler, one will not need wait-free algorithms except in very rare cases?

This question is important because the difference between a wait-free and a lock-free algorithm for any given problem typically involves the introduction of specialized “helping” mechanisms [9], which significantly increase the complexity (both the design complexity and time complexity) of the solution. If one could simply rely on the scheduler, adding a helping mechanism to guarantee wait-freedom (or starvation-freedom) would be unnecessary.

Unfortunately, there is currently no analytical framework which would allow answering the above question, since it would require predicting the behavior of a concurrent algorithm over long executions, under a scheduler that is not adversarial.

Contribution. In this paper, we take a first step towards such a framework. Following empirical observations, we introduce a *stochastic scheduler* model, and use this model to predict the long-term behavior of a general class of concurrent algorithms. The stochastic scheduler is similar to an adversary: at each time step, it picks some process to schedule. The main distinction is that, in our model, the scheduler’s choices *contain some randomness*. In particular, a stochastic scheduler has a probability threshold $\theta > 0$ such that every (non-faulty) process is scheduled with probability at least θ in each step.

We start from the following observation: under *any stochastic scheduler*, every *bounded lock-free* algorithm is actually *wait-free with probability 1*. (A *bounded* lock-free algorithm guarantees that *some* process always makes progress within a finite progress bound.) In other words, for any such algorithm, the schedules which prevent a process from ever making progress must have probability mass 0. The intuition is that, with probability 1, each specific process eventually takes enough consecutive steps, implying that it completes its operation. This observation generalizes to any bounded minimal/maximal progress condition [12]: we show that under a stochastic scheduler, bounded minimal progress becomes maximal progress, with probability 1. However, this intuition is insufficient for explaining why lock-free data structures are *efficient* in practice: because it works for arbitrary algorithms, the upper bound it yields on the number of steps until an operation completes is unacceptably high.

Our main contribution is analyzing a general class of lock-free algorithms under a specific stochastic scheduler, and showing that not only are they wait-free with probability 1, but that in fact they provide a

pragmatic bound on the number of steps until each operation completes.

We address a refined *uniform* stochastic scheduler, which schedules each non-faulty process with uniform probability in every step. Empirical data suggests that, in the long run, the uniform stochastic scheduler is a reasonable approximation for a real-world scheduler (see Figures 1 and 2). We emphasize that we do not claim real schedulers are uniform stochastic, but only that such a scheduler gives a good approximation of what happens in practice for our complexity measures, over long executions.

We call the algorithmic class we analyze *single compare-and-swap universal* (SCU). An algorithm in this class is divided into a *preamble*, and a *scan-and-validate* phase. The preamble executes auxiliary code, such as local updates and memory allocation. In the second phase, the process first determines the data structure state by scanning the memory. It then locally computes the updated state after its method call would be performed, and attempts to commit this state to memory by performing an atomic *compare-and-swap* (CAS) operation. If the CAS operation succeeds, then the state has been updated, and the method call completes. Otherwise, if some other process changes the state in between the scan and the attempted update, then the CAS operation fails, and the process must restart its operation.

This algorithmic class is widely used to design lock-free data structures. It is known that every sequential object has a lock-free implementation in this class using a lock-free version of Herlihy’s universal construction [9]. Instances of this class are used to obtain efficient data structures such as stacks [19], queues [16], or hash tables [6]. The read-copy-update (RCU) [7] synchronization mechanism employed by the Linux kernel is also an instance of this pattern.

We examine the class *SCU* under a uniform stochastic scheduler, and first observe that, in this setting, every such algorithm behaves as a Markov chain. The computational cost of interest is *system steps*, i.e. shared memory accesses by the processes. The complexity metrics we analyze are *individual latency*, which is the expected number of steps of the system until a specific process completes a method call, and *system latency*, which is the expected number of steps of the system to complete *some* method call. We bound these parameters by studying the stationary distribution of the Markov chain induced by the algorithm.

We prove two main results. The first is that, in this setting, all algorithms in this class have the property that the individual latency of any process is n times the system latency. In other words, the expected number of steps for any two processes to complete an operation is *the same*; moreover, the expected number of steps for the system to complete any operation is the expected number of steps for a specific process to complete an operation, divided by n . The second result is an upper bound of $O(q + s\sqrt{n})$ on the system latency, where q is the number of steps in the preamble, s is the number of steps in the scan-and-validate phase, and n is the number of processes. This bound is asymptotically tight.

The key mathematical tool we use is *Markov chain lifting* [3, 8]. More precisely, for such algorithms, we prove that there exists a function which *lifts* the complex Markov chain induced by the algorithm to a simplified *system* chain. The asymptotics of the system latency can be determined directly from the minimal progress chain. In particular, we bound system latency by characterizing the behavior of a new type of *iterated balls-into-bins* game, consisting of iterations which end when a certain condition on the bins first occurs, after which some of the bins change their state and a new iteration begins. Using the lifting, we prove that the individual latency is always n times the system latency.

In summary, our analysis shows that, under an approximation of the real-world scheduler, a large class of lock-free algorithms provide virtually the same progress guarantees as wait-free ones, and that, roughly, the system completes requests at a rate that is n times that of individual processes. More generally, it provides for the first time an analytical framework for predicting the behavior of a class of concurrent algorithms, over long executions, under a scheduler that is not adversarial.

Related work. To the best of our knowledge, the only prior work which addresses a probabilistic sched-

uler for a shared memory environment is that of Aspnes [2], who gave a fast consensus algorithm under a probabilistic scheduler model different from the one considered in this paper. The observation that many lock-free algorithms behave as wait-free in practice was made by Herlihy and Shavit in the context of formalizing minimal and maximal progress conditions [12], and is well-known among practitioners. For example, reference [1, Figure 6] gives empirical results for the latency distribution of individual operations of a lock-free stack.

Roadmap. We describe the model, progress guarantees, and complexity metrics in Section 2. In particular, Section 2.3 defines the stochastic scheduler. We show that minimal progress becomes maximal progress with probability 1 in Section 3. Section 4 defines the class $SCU(q, s)$, while Section 5.1 analyzes individual and global latency. The Appendix, to be viewed at the committee’s discretion, contains empirical justification for the model, the complete analysis, and a comparison between the predicted behavior of an algorithm and its practical performance.

2 System Model

2.1 Preliminaries

Processes and Objects. We consider a shared-memory model, in which n processes p_1, \dots, p_n , communicate through registers, on which they perform atomic read, write, and compare-and-swap (CAS) operations. A CAS operation takes three arguments $(R, exp\ Val, newVal)$, where R is the register on which it is applied, $exp\ Val$ is the expected value of the register, and $newVal$ is the new value to be written to the register. If $exp\ Val$ matches the value of R , then we say that the CAS is successful, and the value of R is updated to $newVal$. Otherwise, the CAS fails. The operation returns *true* if it successful, and *false* otherwise.

We assume that each process has a unique identifier. Processes follow an algorithm, composed of shared-memory steps and local computation. The order of process steps is controlled by the *scheduler*. A set of at most $n - 1$ processes may fail by crashing. A crashed process stops taking steps for the rest of the execution. A process that is not crashed at a certain step is *correct*, and if it never crashes then it takes an infinite number of steps in the execution.

The algorithms we consider are implementations of shared objects. A shared object O is an abstraction providing a set of *methods* M , each given by its sequential specification. In particular, an implementation of a method m for object O is a set of n algorithms, one for each executing process. When process p_i invokes method m of object O , it follows the corresponding algorithm until it receives a response from the algorithm. In the following, we do not distinguish between a method m and its implementation. A method invocation is *pending* if has not received a response. A method invocation is *active* if it is made by a *correct* process (note that the process may still crash in the future).

Executions, Schedules, and Histories. An execution is a sequence of operations performed by the processes. To represent executions, we assume discrete time, where at every time unit only one process is scheduled. In a time unit, a process can perform any number of local computations or coin flips, after which it issues a *step*, which consists of a single shared memory operation. Whenever a process becomes active, as decided by the scheduler, it performs its local computation and then executes a step. The *schedule* is a (possibly infinite) sequence of process identifiers. If process p_i is in position $\tau \geq 1$ in the sequence, then p_i is active at time step τ .

Raising the level of abstraction, we define a *history* as a finite sequence of method invocation and response events. Notice that each schedule has a corresponding history, in which individual process steps are mapped to method calls. On the other hand, a history can be the image of several schedules.

2.2 Progress Guarantees

We now define minimal and maximal progress guarantees. We partly follow the unified presentation from [12], except that we do not specify progress guarantees for each method of an object. Rather, for ease of presentation, we adopt the simpler definition which specifies progress provided by an implementation. Consider an execution e , with the corresponding history H_e . An implementation of an object O provides *minimal progress* in the execution e if, in every suffix of H_e , some pending active instance of some method has a matching response. Equivalently, there is no point in the corresponding execution from which all the processes take an infinite number of steps without returning from their invocation.

An implementation provides *maximal progress* in an execution e if, in every suffix of the corresponding history H_e , *every* pending active invocation of a method has a response. Equivalently, there is no point in the execution from which a process takes infinitely many steps without returning.

Scheduler Assumptions. We say that an execution is *crash-free* if each process is always correct, i.e. if each process takes an infinite number of steps. An execution is *uniformly isolating* if, for every $k > 0$, every correct process has an interval where it takes at least k consecutive steps.

Progress. An implementation is *deadlock-free* if it guarantees minimal progress in every crash-free execution, and maximal progress in some crash-free execution.¹ An implementation is *starvation-free* if it guarantees maximal progress in every crash-free execution. An implementation is *clash-free* if it guarantees minimal progress in every uniformly isolating history, and maximal progress in some such history [12]. An implementation is *obstruction-free* if it guarantees maximal progress in every uniformly isolating execution². An implementation is *lock-free* if it guarantees minimal progress in every execution, and maximal progress in some execution. An implementation is *wait-free* if it guarantees maximal progress in every execution.

Bounded Progress. While the above definitions provide reasonable measures of progress, often in practice more explicit progress guarantees may be desired, which provide an upper bound on the number of steps until some method makes progress. To model this, we say that an implementation guarantees *bounded minimal progress* if there exists a bound $B > 0$ such that, for any time step t in the execution e at which there is an active invocation of some method, some invocation of a method returns within the next B steps by all processes. An implementation guarantees *bounded maximal progress* if there exists a bound $B > 0$ such that *every* active invocation of a method returns within B steps by all processes. We can specialize the definitions of bounded progress guarantees to the scheduler assumptions considered above to obtain definitions for *bounded deadlock-freedom*, *bounded starvation-freedom*, and so on.

2.3 Stochastic Schedulers

We define a stochastic scheduler as follows.

Definition 1 (Stochastic Scheduler). *For any $n \geq 0$, a scheduler for n processes is defined by a triple $(\Pi_\tau, A_\tau, \theta)$. The parameter $\theta \in [0, 1]$ is the threshold. For each time step $\tau \geq 1$, Π_τ is a probability distribution for scheduling the n processes at τ , and A_τ is the subset of possibly active processes at time step τ . At time step $\tau \geq 1$, the distribution Π_τ gives, for every $i \in \{1, \dots, n\}$ a probability γ_τ^i , with which process p_i is scheduled. The distribution Π_τ may depend on arbitrary outside factors, such as the current*

¹According to [12], the algorithm is required to guarantee maximal progress in some execution to rule out pathological cases where a thread locks the object and never releases the lock.

²This is the definition of obstruction freedom from [12]; it is weaker than the one in [10] since it assumes uniformly isolating schedules only, but we use it here as it complies with our requirements of providing maximal progress.

state of the algorithm being scheduled. A scheduler $(\Pi_\tau, A_\tau, \theta)$ is stochastic if $\theta > 0$. For every $\tau \geq 1$, the parameters must ensure the following:

1. (Well-formedness) $\sum_{i=1}^n \gamma_\tau^i = 1$;
2. (Weak Fairness) For every process $p_i \in A_\tau$, $\gamma_\tau^i \geq \theta$;
3. (Crashes) For every process $p_i \notin A_\tau$, $\gamma_\tau^i = 0$;
4. (Crash Containment) $A_{\tau+1} \subseteq A_\tau$.

The well-formedness condition ensures that some process is always scheduled. Weak fairness ensures that, for a stochastic scheduler, possibly active processes do get scheduled with some non-zero probability. The crash condition ensures that failed processes do not get scheduled. The set $\{p_1, p_2, \dots, p_n\} \setminus A_\tau$ can be seen as the set of crashed processes at time step τ , since the probability of scheduling these processes at every subsequent time step is 0.

An Adversarial Scheduler. Any classic asynchronous shared memory adversary can be modeled by “encoding” its adversarial strategy in the probability distribution Π_τ for each step. Specifically, given an algorithm A and a worst-case adversary \mathcal{A}_A for A , let p_i^τ be the process that is scheduled by \mathcal{A}_A at time step τ . Then we give probability 1 in Π_τ to process p_i^τ , and 0 to all other processes. Things are more interesting when the threshold θ is strictly more than 0, i.e., there is some randomness in the scheduler’s choices.

The Uniform Stochastic Scheduler. A natural scheduler is the *uniform* stochastic scheduler, for which, assuming no process crashes, we have that Π_τ has $\gamma_i^\tau = 1/n$, for all i and $\tau \geq 1$, and $A_\tau = \{1, \dots, n\}$ for all time steps $\tau \geq 1$. With crashes, we have that $\gamma_i^\tau = 1/|A_\tau|$ if $i \in A_\tau$, and $\gamma_i^\tau = 0$ otherwise.

2.4 Complexity Measures

Given a concurrent algorithm, standard analysis focuses on two measures: *step complexity*, the worst-case number of steps performed by a single process in order to return from a method invocation, and *total step complexity*, or *work*, which is the worst-case number of system steps required to complete invocations of all correct processes when performing a task together. In this paper, we focus on the analogue of these complexity measures for long executions. Given a stochastic scheduler, we define (*average*) *individual latency* as the maximum over all inputs of the expected number of steps taken by the system between the returns times of two consecutive invocations of the same process. Similarly, we define the (*average*) *system latency* as the maximum over all inputs of the expected number of system steps between consecutive returns times of any two invocations.

2.5 Background on Markov Chains

We now give a brief overview of Markov chains. Our presentation follows standard texts, e.g. [15, 17]. The definition and properties of Markov chain lifting are lifted from [8].

Given a set S , a sequence of random variables $(X_t)_{t \in \mathbb{N}}$, where $X_t \in S$, is a (discrete-time) *stochastic process* with states in S . A *discrete-time Markov chain* over the state set S is a discrete-time stochastic process with states in S that satisfies the *Markov condition*

$$\Pr[X_t = i_t | X_{t-1} = i_{t-1}, \dots, X_0 = i_0] = \Pr[X_t = i_t | X_{t-1} = i_{t-1}].$$

The above condition is also called the *memoryless property*. A Markov chain is *time-invariant* if the equality $\Pr[X_t = j | X_{t-1} = i] = \Pr[X_{t'} = j | X_{t'-1} = i]$ holds for all times $t, t' \in \mathbb{N}$ and all $i, j \in S$. This allows us

to define the *transition matrix* P of a Markov chain as the matrix with entries

$$p_{ij} = \Pr[X_t = j | X_{t-1} = i].$$

The *initial distribution* of a Markov chain is given by the probabilities $\Pr[X_0 = i]$, for all $i \in S$. We denote the time-invariant Markov chain X with initial distribution λ and transition matrix P by $M(P, \lambda)$.

The random variable $T_{ij} = \min\{n \geq 1 | X_n = j, \text{ if } X_0 = i\}$ counts the number of steps needed by the Markov chain to get from i to j , and is called the *hitting time* from i to j . We set $T_{i,j} = \infty$ if state j is unreachable from i . Further, we define $h_{ij} = E[T_{ij}]$, and call $h_{ii} = E[T_{ii}]$ the (expected) return time for state $i \in S$.

Given P , the transition matrix of $M(P, \lambda)$, a *stationary distribution* of the Markov chain is a state vector π with $\pi = \pi P$. (We consider *row* vectors throughout the paper.) The intuition is that if the state vector of the Markov chain is π at time t , then it will remain π for all $t' > t$. Let $P^{(k)}$ be the transition matrix P multiplied by itself k times, and $p_{ij}^{(k)}$ be element (i, j) of $P^{(k)}$. A Markov chain is *irreducible* if for all pairs of states $i, j \in S$ there exists $m \geq 0$ such that $p_{ij}^{(m)} > 0$. (In other words, the underlying graph is strongly connected.) This implies that $T_{ij} < \infty$, and all expectations h_{ij} exist, for all $i, j \in S$. Furthermore, the following is known.

Theorem 1. *An irreducible finite Markov chain has a unique stationary distribution π , namely*

$$\pi_j = \frac{1}{h_{jj}}, \forall j \in S.$$

The periodicity of a state j is the maximum positive integer α such that $\{n \in \mathbb{N} | p_{jj}^{(n)} > 0\} \subseteq \{i\alpha | i \in \mathbb{N}\}$. A state with periodicity $\alpha = 1$ is called *aperiodic*. A Markov chain is *aperiodic* if all states are aperiodic. If a Markov chain has at least one self-loop, then it is aperiodic. A Markov chain that is irreducible and aperiodic is *ergodic*. Ergodic Markov chains converge to their stationary distribution as $t \rightarrow \infty$ independently of their initial distributions.

Theorem 2. *For every ergodic finite Markov chain $(X_t)_{t \in \mathbb{N}}$ we have independently of the initial distribution that $\lim_{t \rightarrow \infty} q_t = \pi$, where π denotes the chain's unique stationary distribution, and q_t is the distribution on states at time $t \in \mathbb{N}$.*

Ergodic Flow. It is often convenient to describe an ergodic Markov chain in terms of its *ergodic flow*: for each (directed) edge ij , we associate a flow $Q_{ij} = \pi_i p_{ij}$. These values satisfy $\sum_i Q_{ij} = \sum_i Q_{ji}$ and $\sum_{i,j} Q_{ij} = 1$. It also holds that $\pi_j = \sum_i Q_{ij}$.

Lifting Markov Chains. Let M and M' be ergodic Markov chains on finite state spaces S, S' , respectively. Let P, π be the transition matrix and stationary distribution for M , and P', π' denote the corresponding objects for M' . We say that M' is a *lifting* of M [8] if there is a function $f : S' \rightarrow S$ such that

$$Q_{ij} = \sum_{x \in f^{-1}(i), y \in f^{-1}(j)} Q'_{xy}, \forall i, j \in S.$$

Informally, M' is collapsed onto M by clustering several of its states into a single state, as specified by the function f . The above relation specifies a homomorphism on the ergodic flows. An immediate consequence of this relation is the following connection between the stationary distributions of the two chains.

Lemma 1. *For all $v \in S$, we have that*

$$\pi(v) = \sum_{x \in f^{-1}(v)} \pi'(x).$$

```

1 Shared: registers  $R, R_1, R_2, \dots, R_{s-1}$ 
2 procedure method-call()
3 Take preamble steps  $O_1, O_2, O_q$  /* Preamble region */
4 while true do
    /* Scan region: */
    5  $v \leftarrow R.\text{read}()$ 
    6  $v_1 \leftarrow R_1.\text{read}(); v_2 \leftarrow R_2.\text{read}(); \dots; v_{s-1} \leftarrow R_{s-1}.\text{read}()$ 
    7  $v' \leftarrow \text{new proposed state based on } v, v_1, v_2, \dots, v_{s-1}$ 
    /* Validation step: */
    8  $\text{flag} \leftarrow \text{CAS}(R, v, v')$ 
    9 if  $\text{flag} = \text{true}$  then
        output success

```

Algorithm 1: The structure of the lock-free algorithms in $SCU_{q,s}$.

3 From Minimal Progress to Maximal Progress

We now formalize the intuition that, under a stochastic scheduler, all algorithms ensuring bounded minimal progress guarantee in fact maximal progress with probability 1. We also show the *bounded* minimal progress assumption is necessary: if minimal progress is not bounded, then maximal progress may not be achieved.

Theorem 3 (Min to Max Progress). *Let S be a stochastic scheduler with probability threshold $1 \geq \theta > 0$. Let A be an algorithm ensuring bounded minimal progress with a bound T . Then A ensures maximal progress with probability 1. Moreover, the expected maximal progress bound of A is at most $(1/\theta)^T$.*

The proof, which can be found in the Appendix, is based on the fact that, for every correct process p_i , eventually, the scheduler will produce a solo a schedule of length T . On the other hand, since the algorithm ensures minimal progress with bound T , we show that p_i must complete its operation during this interval.

We then prove that the finite bound for minimal progress is necessary. For this, we devise an *unbounded* lock-free algorithm which is not wait-free with probability > 0 . The main idea is to have processes that fail to change the value of a CAS repeatedly increase the number of steps they need to take to complete an operation. The argument is given in the Appendix.

Lemma 2. *There exists an unbounded lock-free algorithm that is not wait-free with high probability.*

4 The Class of Algorithms $SCU(q, s)$

In this section, we define the class of algorithms $SCU(q, s)$. An algorithm in this class is structured as follows. (See Algorithm 1 for the pseudocode.) The first part is the *preamble*, where the process performs a series of q steps. The algorithm then enters a *loop*, divided into a *scan* region, which reads the values of s registers, and a *validation* step, where the process performs a CAS operation, which attempts to change the value of a register. The goal of the scan region is to obtain a view of the data structure state. In the validation step, the process checks that this state is still valid, and attempts to change it. If the CAS is successful, then the operation completes. Otherwise, the process restarts the loop. We say that an algorithm with the above structure with parameters q and s is in $SCU(q, s)$.

We assume that steps in the preamble may perform memory updates, including to registers R_1, \dots, R_{s-1} , but do not change the value of the decision register R . Also, two processes never propose the same value for the register R . (This can be easily enforced by adding a timestamp to each request.) The order of steps in the

scan region can be changed without affecting our analysis. Such algorithms are used in several CAS-based concurrent implementations. In particular, the class can be used to implement a concurrent version of every sequential object [9]. It has also been used to obtain efficient implementations of several concurrent objects, such as fetch-and-increment [4], stacks [19], and queues [16].

5 Analysis of the Class $SCU(q, s)$

We analyze the performance of algorithms in $SCU(q, s)$ under the uniform stochastic scheduler. We assume that all threads execute the same method call with preamble of length q , and scan region of length s . Each thread executes an infinite number of such operations. To simplify the presentation, we assume all n threads are correct in the analysis. The claim is similar in the crash-failure case, and will be considered separately.

We examine two parameters: system latency, i.e., how often (in terms of system steps) does a new operation complete, and individual latency, i.e., how often does a *certain thread* complete a new operation. Notice that the worst-case latency for the whole system is $\Theta(q + sn)$ steps, while the worst-case latency for an individual thread is ∞ , as the algorithm is not wait-free. We will prove the following result:

Theorem 4. *Let A be an algorithm in $SCU(q, s)$. Then, under the uniform stochastic scheduler, the system latency of A is $O(q + s\sqrt{n})$, and the individual latency is $O(n(q + s\sqrt{n}))$.*

We prove the upper bound by splitting the class $SCU(q, s)$ into two separate components, and analyzing each under the uniform scheduler. The first part is the loop code, which we call the *scan-validate* component. The second part is the *parallel code*, which we use to characterize the performance of the preamble code. In other words, we first consider $SCU(0, s)$ and then $SCU(q, 0)$.

5.1 The Scan-Validate Component

Without loss of generality, we can simplify the pseudocode to contain a single read step before the CAS. We obtain the performance bounds for this simplified algorithm, and then multiply them by s , the number scan steps. That is, we start by analyzing $SCU(0, 1)$ and then generalize to $SCU(0, s)$.

Proof Strategy. We start from the Markov chain representation of the algorithm, which we call the *individual chain*. We then focus on a simplified representation, which only tracks *system-wide progress*, irrespective of which process is exactly in which state. We call this the *system chain*. We first prove the individual chain can be related to the system chain via a lifting function, which allows us to relate the individual latency to the system latency (Lemma 3). We then focus on bounding system latency. We describe the behavior of the system chain via an iterated balls-and-bins game, whose stationary behavior we analyze in Lemmas 5 and 6. Finally, we put together these claims to obtain an $O(\sqrt{n})$ upper bound on the system latency of $SCU(0, 1)$.

Due to space limitations, we only present an outline of the argument. The complete version can be found in the Appendix.

5.1.1 Markov Chain Representations

We define the *extended local state* of a process in terms of the state of the system, and of the type of step it is about to take. Thus, a process can be in one of three states: either it performs a read, or it CAS-es with the current value of R , or it CAS-es with an invalid value of R . The state of the system after each step is completely described by the n extended local states of processes. We emphasize that this is different than what is typically referred to as the “local” state of a process, in that the extended local state is described

from the viewpoint of the entire system. That is, a process that has a pending CAS operation can be in either of two different extended local states, depending on whether its CAS will succeed or not. This is determined by the state of the entire system. A key observation is that, although the “local” state of a process can only change when it takes a step, its extended local state can change also when another process takes a step.

The individual chain. Since the scheduler is uniform, the system can be described as a Markov chain, where each state specifies the extended local state of each process. Specifically, a process is in state *OldCAS* if it is about to CAS with an old (invalid) value of R , it is in state *Read* if it is about to read, and is in state *CCAS* if it about to CAS with the current value of R . (Once CAS-ing the process returns to state *Read*.)

A state S of the individual chain is given by a combination of n states $S = (P_1, P_2, \dots, P_n)$, describing the extended local state of each process, where, for each $i \in \{1, \dots, n\}$, $P_i \in \{\text{OldCAS}, \text{Read}, \text{CCAS}\}$ is the extended local state of process p_i . There are $3^n - 1$ possible states, since the state where each process CAS-es with an old value cannot occur. In each transition, each process takes a step, and the state changes correspondingly. Recall that every process p_i takes a step with probability $1/n$. Transitions are as follows. If the process p_i taking a step is in state *Read* or *OldCAS*, then all other processes remain in the same extended local state, and p_i moves to state *CCAS* or *Read*, respectively. If the process p_i taking a step is in state *CCAS*, then all processes in state *CCAS* move to state *OldCAS*, and p_i moves to state *Read*.

The system chain. To reduce the complexity of the individual Markov chain, we introduce a simplified representation, which tracks system-wide progress. More precisely, each state of the system chain tracks the number of processes in each state, irrespective of their identifiers: for any $a, b \in \{0, \dots, n\}$, a state x is defined by the tuple (a, b) , where a is the number of processes that are in state *Read*, and b is the number of processes that are in state *OldCAS*. Notice that the remaining $n - a - b$ processes must be in state *CCAS*. The initial state is $(n, 0)$, i.e. all processes are about to read. The state $(0, n)$ does not exist. The transitions in the system chain are as follows. $\Pr[(a+1, b-1)|(a, b)] = b/n$, where $0 \leq a \leq n$ and $b > 0$. $\Pr[(a+1, b)|(a, b)] = 1 - (a+b)/n$, where $0 \leq a < n$. $\Pr[(a-1, b)|(a, b)] = 1 - a/n$, where $0 < a \leq n$. (See Figure 3 in the Appendix for an illustration of the two chains in the two-process case.)

5.1.2 Lifting the Individual Chain

We start from the observation that both the individual and the system chains are ergodic. Let π be the stationary distribution of the system chain, and let π' be the stationary distribution for the individual chain. For any state $j = (a, b)$ in the system chain, let π_j be its probability in the stationary distribution. Similarly, for state x in the individual chain, let π'_x be its probability in the stationary distribution.

We now define a *lifting* from the individual chain to the system chain. Let \mathcal{S} be the set of states of the individual chain, and \mathcal{M} be the set of states of the system chain. We define $f : \mathcal{S} \rightarrow \mathcal{M}$ such that each state $S = (P_1, \dots, P_n)$, where a processes are in state *Read* and b processes are in state *OldCAS*, is taken into state (a, b) of the system chain. Intuitively, f collapses all states in which a processes are about to read and b processes are about to CAS with an old value, into state (a, b) from the system chain. We prove that the function $f : \mathcal{S} \rightarrow \mathcal{M}$ defined above induces a lifting from the individual chain to the system chain.

Lemma 3. *The system Markov chain is a lifting of the individual Markov chain.*

We then use the fact that the code is symmetric and the previous Lemma to obtain an upper bound on the expected time between two successes for a specific process.

Lemma 4. *Let W be the expected system steps between two successes in the stationary distribution of the system chain. Let W_i be the expected system steps between two successes of process p_i in the stationary distribution of the individual chain. For every process p_i , $W = nW_i$.*

Proof (Sketch). Let μ be the probability that a step is a success by *some* process. Expressed in the system chain, we have that $\mu = \sum_{j=(a,b)} (1 - (a+b)/n) \pi_j$. Let X_i be the set of states in the individual chain in which $P_i = CCAS$. Consider the event that a system step is a step in which p_i succeeds. This must be a step by p_i from a state in X_i . The probability of this event in the stationary distribution of the individual chain is $\eta_i = \sum_{x \in X_i} \pi'_x / n$.

Recall that the lifting function f maps all states x with a processes in state *Read* and b processes in state *OldCAS* to state $j = (a, b)$. Therefore, $\eta_i = (1/n) \sum_{j=(a,b)} \sum_{x \in f^{-1}(j) \cap X_i} \pi'_x$. By symmetry, we have that $\pi'_x = \pi'_y$, for every states $x, y \in f^{-1}(j)$. The fraction of states in $f^{-1}(j)$ that have p_i in state *CCAS* (and are therefore also in X_i) is $(1 - (a+b)/n)$. Therefore, $\sum_{x \in f^{-1}(j) \cap X_i} \pi'_x = (1 - (a+b)/n) \pi_j$.

Finally, we get that, for every process p_i , $\eta_i = (1/n) \sum_{j=(a,b)} (1 - (a+b)/n) \pi_j = (1/n) \mu$. On the other hand, since we consider the stationary distribution, from a straightforward extension of Theorem 1, we have that $W_i = 1/\eta_i$, and $W = 1/\mu$. Therefore, $W_i = nW$, as claimed. \square

5.1.3 System Latency Bound

In this section we provide an upper bound on the system latency. We prove the following.

Theorem 5. *The expected number of steps between two successes in the system chain is $O(\sqrt{n})$.*

An iterated balls-into-bins game. To bound W , we model the evolution of the system as a balls-into-bins game. We will associate each process with a bin. At the beginning of the execution, each bin already contains one ball. At each time step, we throw a new ball into a uniformly chosen random bin. Essentially, whenever the process takes a step, its bin receives an additional ball. We continue to distribute balls until the first time a bin acquires *three* balls. We call this event a *reset*. When a reset occurs, we set the number of balls in the bin containing three balls to one, and all the bins containing two balls become empty. The game then continues until the next reset.

This game models the fact that initially, each process is about to read the shared state. To change its value, it must take two steps without the state changing in between. A process which changes the shared state by CAS-ing successfully causes all other processes which were about to CAS with the correct value to fail their operations. These processes now need to take *three* steps to change the shared state. We therefore reset the number of balls in the corresponding bins to 0. We define the game in terms of *phases*. A phase is the interval between two resets. For phase i , we denote by a_i the number of bins with one ball at the beginning of the phase, and by b_i the number of bins with 0 balls at the beginning of the phase. Since there are no bins with two or more balls at the start of a phase, we have that $a_i + b_i = n$.

Notice that this iterated game evolves in the same way as the system Markov chain. In particular, W is the expected length of a phase. To prove Theorem 5, we first obtain a bound on the length of a phase in terms of the parameters a_i and b_i .

Lemma 5. *Let $\alpha \geq 4$ be a constant. The expected length of phase i is at most $\min(2\alpha n / \sqrt{a_i}, 3\alpha n / b_i^{1/3})$. The phase length is $2\alpha \min(n\sqrt{\log n} / \sqrt{a_i}, n(\log n)^{1/3} / b_i^{1/3})$, with probability at least $1 - 1/n^\alpha$. The probability that the length of a phase is less than $\min(n / \sqrt{a_i}, n / (b_i)^{1/3}) / \alpha$ is at most $1/(4\alpha^2)$.*

Next, we analyze the dynamics of the phases $i \geq 1$ based on the value of a_i at the beginning of the phase. Fix c a large constant. Phase i is in *the first range* if $a_i \in [n/3, n]$, in the *second range* if $n/c \leq a_i < n/3$, and is in the *third range* if $0 \leq a_i < n/c$. We analyze the probability of moving between ranges by carefully bounding the change in ball counts for the bins during a phase as a function of a_i 's initial value.

Lemma 6. *For $i \geq 1$, if phase i is in the first two ranges, then the probability that phase $i + 1$ is in the third range is at most $1/n^\alpha$. Let $\beta > 2c^2$ be a constant. The probability that $\beta\sqrt{n}$ consecutive phases are in the third range is at most $1/n^\alpha$.*

Final argument. To complete the proof of Theorem 5, we group the states of the game according to their range: state $S_{1,2}$ contains all states (a_i, b_i) in the first two ranges, i.e. with $a_i \geq n/c$. State S_3 contains all states (a_i, b_i) with $a_i < n/c$. Using Lemmas 5 and 6, we obtain that the collective probability of states in $S_{1,2}$ is at least $1 - \beta\sqrt{n}/n^\alpha$, while the probability of states in S_3 is at most $\beta\sqrt{n}/n^\alpha$. Therefore, the expected length of a phase is at most $2\alpha\sqrt{n}(1 - \beta\sqrt{n}/n^\alpha) + \beta n^{2/3}\sqrt{n}/n^\alpha = O(\sqrt{n})$, as claimed. This completes the proof of Theorem 5. Note that, per Lemma 5, this bound is asymptotically tight.

5.2 General Bounds for $SCU(q, s)$

We now put together the results of the previous sections to obtain a bound on individual and system latency. First, we notice that Theorem 5 can be easily extended to the case where the loop contains s scan steps, as the extended local state of a process p can be changed by a step of another process $q \neq p$ only if p is about to perform a CAS operation. We obtain that both the system and the individual latency bounds are multiplied by s . We then use the lifting method to analyze the parallel code, i.e. $SCU(q, 0)$. The key observation is that, in this case, the stationary distribution for the individual chain is uniform. The argument can be found in Section D.2.

Lemma 7. *For any $1 \leq i \leq n$ and $q \geq 0$, given an algorithm in $SCU(q, 0)$, its individual latency is $W_i = nq$, and its system latency is $W = q$.*

Clearly, an algorithm in $SCU(q, s)$ is a sequential composition of parallel code followed by s loop steps. Fix a process p_i . By Lemma 7, using linearity of expectation, we obtain that the expected individual latency for process p_i to complete an operation is $O(n(q + s\sqrt{n}))$. We define the individual chain and the system chain for the general algorithm, and show that a lifting exists. This again implies that the system latency is $O(q + s\sqrt{n})$, which completes the proof of Theorem 4.

We note that the above argument also gives an upper bound on the expected number of (individual) steps a process p_i needs to complete an operation (similar to the standard measure of individual *step complexity*). Since the scheduler is uniform, this is also $O(q + s\sqrt{n})$. Finally, we note that, if only $k \leq n$ processes are correct in the execution, we obtain the same latency bounds in terms of k : since we consider the behavior of the algorithm at infinity, the stationary latencies are only influenced by correct processes.

Corollary 1. *Given an algorithm in $SCU(q, s)$ on k correct processes under a uniform stochastic scheduler, the system latency is $O(q + s\sqrt{k})$, and the individual latency is $O(k(q + s\sqrt{k}))$.*

6 Discussion

This paper is motivated by the fundamental question of relating the theory of concurrent programming to real-world algorithm behavior. We give a framework for analyzing concurrent algorithms which partially explains the wait-free behavior of lock-free algorithms, and their good performance in practice. Our work is a first step in this direction, and opens the door to many additional questions.

In particular, we are intrigued by the goal of obtaining a realistic model for the unpredictable behavior of system schedulers. Even though it has some foundation in empirical results, our uniform stochastic model is a rough approximation, and can probably be improved. We believe that some of the elements of

our framework (such as the existence of liftings) could still be applied to non-uniform stochastic scheduler models, while others may need to be further developed. A second direction for future work is studying other types of algorithms, and in particular implementations which export several distinct methods. The class of algorithms we consider is *universal*, i.e., covers any sequential object, however there may exist object implementations which do not fall in this class. Finally, it would be interesting to explore whether there exist concurrent algorithms which avoid the $\Theta(\sqrt{n})$ contention factor in the latency, and whether such algorithms are efficient in practice.

Acknowledgements. We thank George Giakkoupis, William Hasenplaugh, Maurice Herlihy, and Yuval Peres for useful discussions, and Faith Ellen for helpful comments on an earlier version of the paper.

References

- [1] Samy Al-Bahra. Nonblocking algorithms and scalable multicore programming. *Commun. ACM*, 56(7):50–61, 2013.
- [2] James Aspnes. Fast deterministic consensus in a noisy environment. *J. Algorithms*, 45(1):16–39, 2002.
- [3] Fang Chen, László Lovász, and Igor Pak. Lifting markov chains to speed up mixing. In *Proceedings of the thirty-first annual ACM symposium on Theory of computing*, STOC ’99, pages 275–281, New York, NY, USA, 1999. ACM.
- [4] Dave Dice, Yossi Lev, and Mark Moir. Scalable statistics counters. In *25th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA ’13, Montreal, QC, Canada , 2013*, pages 43–52, 2013.
- [5] Philippe Flajolet, Peter J. Grabner, Peter Kirschenhofer, and Helmut Prodinger. On Ramanujan’s Q-function. *J. Comput. Appl. Math.*, 58(1):103–116, March 1995.
- [6] Keir Fraser. Practical lock-freedom. Technical Report UCAM-CL-TR-579, University of Cambridge, Computer Laboratory, February 2004.
- [7] D. Guniguntala, P.E. McKenney, J. Triplett, and J. Walpole. The read-copy-update mechanism for supporting real-time applications on shared-memory multiprocessor systems with linux. *IBM Systems Journal*, 47(2):221–236, 2008.
- [8] Thomas P. Hayes and Alistair Sinclair. Liftings of tree-structured markov chains. In *Proceedings of the 13th international conference on Approximation, and 14 the International conference on Randomization, and combinatorial optimization: algorithms and techniques*, APPROX/RANDOM’10, pages 602–616, Berlin, Heidelberg, 2010. Springer-Verlag.
- [9] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):123–149, January 1991.
- [10] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *23rd International Conference on Distributed Computing Systems (ICDCS 2003)*, pages 522–529, 2003.
- [11] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

- [12] Maurice Herlihy and Nir Shavit. On the nature of progress. In *15th International Conference on Principles of Distributed Systems (OPODIS), Toulouse, France, December 13-16, 2011. Proceedings*, pages 313–328, 2011.
- [13] Donald E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.
- [14] Leslie Lamport. A new solution of dijkstra’s concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974.
- [15] David A. Levin, Yuval Peres, and Elizabeth L. Wilmer. *Markov Chains and Mixing Times*. American Mathematical Society, 2008.
- [16] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*, pages 267–275, 1996.
- [17] Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, New York, NY, USA, 2005.
- [18] David Petrou, John W. Milford, and Garth A. Gibson. Implementing lottery scheduling: matching the specializations in traditional schedulers. In *Proceedings of the annual conference on USENIX Annual Technical Conference, ATEC ’99*, pages 1–1, Berkeley, CA, USA, 1999. USENIX Association.
- [19] R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, 1986.

Structure of the Appendix. Section A presents empirical data for the stochastic scheduler model. Section B gives background on Markov chains. Section C presents the full proofs from Section 3. Section D presents the full latency analysis for $SCU(q, s)$. Section E applies the framework to an augmented CAS counter, while Section E.3 gives compares the predicted and actual performance of an algorithm in SCU .

A The Stochastic Scheduler Model

A.1 Empirical Justification

The real-world behavior of a process scheduler arises as a complex interaction of factors such as the timing of memory requests (influenced by the algorithm), the behavior of the cache coherence protocol (dependent on the architecture), or thread pre-emption (depending on the operating system). Given the extremely complex interactions between these components, the behavior of the scheduler could be seen as *non-deterministic*. However, when recorded for extended periods of time, simple patterns emerge. Figures 1 and 2 present statistics on schedule recordings from a simple concurrent counter algorithm, executed on a system with 16 hardware threads. (The details of the setup and experiments are presented in the next section).

Figure 1 clearly suggests that, in the long run, the scheduler is “fair:” each thread gets to take about the same number of steps. Figure 2 gives an intuition about how the schedule looks like *locally*: assuming process p_i just took a step at time step τ , any process appears to be just as likely to be scheduled in the next step. We note that the structure of the algorithm executed can influence the ratios in Figure 2; also, we only performed tests on an Intel architecture.

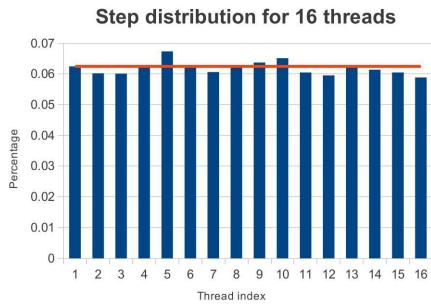


Figure 1: Percentage of steps taken by each process during an execution.

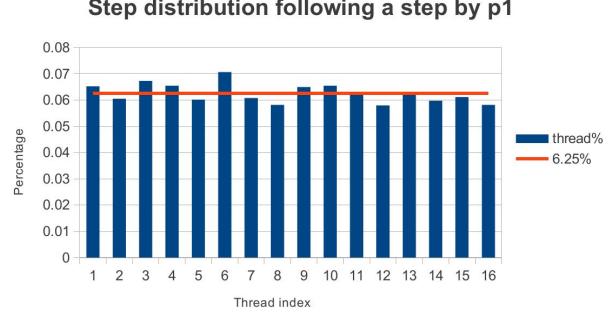


Figure 2: Percentage of steps taken by processes, starting from a step by p_1 . (The results are similar for all threads.)

Our stochastic scheduler model addresses the non-determinism in the scheduler by associating a distribution with each scheduler time step, which gives the probability of each process being scheduled next. In particular, we model our empirical observations by considering the uniform stochastic scheduler, which assigns a probability of $1/n$ with which each process is scheduled. We stress that we do not claim that the schedule behaves uniformly random locally; our claim is that the behavior of the schedule over long periods of time can be approximated reasonably in this way, for the algorithms we consider. We note that randomized schedulers attempting to explicitly implement probabilistic fairness have been proposed in practice, in the form of *lottery scheduling* [18].

A.2 Experimental Setup

The machine we use for testing is a Fujitsu PRIMERGY RX600 S6 server with four Intel Xeon E7-4870 (Westmere EX) processors. Each processor has 10 2.40 GHz cores, each of which multiplexes two hardware threads, so in total our system supports 80 hardware threads. Each core has private write-back L1 and L2 caches; an inclusive L3 cache is shared by all cores. We limited experiments to 20 hardware threads, in order to avoid the effects of non-uniform memory access (NUMA), which appear when hardware threads are located on different cores.

We used two methods to record schedules. The first used an atomic fetch-and-increment operation (available in hardware): each process repeatedly calls this operation, and records the values received. We then sort the values of each process to recover the total order of steps. The second method records timestamps during the execution of an algorithm, and sorts the timestamps to recover the total order. We found that the latter method interferes with the schedule: since the timer call causes a delay to the caller, a process is less likely to be scheduled twice in succession. With this exception, the results are similar for both methods. The statistics of the recorded schedule are summarized in Figures 1 and 2. (The graphs are built using 20 millisecond runs, averaged over 10 repetitions; results for longer intervals and for different thread counts are similar.)

B Background on Markov Chains

We now give a brief overview of Markov chains. Our presentation follows standard texts, e.g. [15, 17]. The definition and properties of Markov chain lifting are adapted from [8].

Given a set S , a sequence of random variables $(X_t)_{t \in \mathbb{N}}$, where $X_t \in S$, is a (discrete-time) *stochastic process* with states in S . A *discrete-time Markov chain* over the state set S is a discrete-time stochastic process with states in S that satisfies the *Markov condition*

$$\Pr[X_t = i_t | X_{t-1} = i_{t-1}, \dots, X_0 = i_0] = \Pr[X_t = i_t | X_{t-1} = i_{t-1}].$$

The above condition is also called the *memoryless property*. A Markov chain is *time-invariant* if the equality $\Pr[X_t = j | X_{t-1} = i] = \Pr[X_{t'} = j | X_{t'-1} = i]$ holds for all times $t, t' \in \mathbb{N}$ and all $i, j \in S$. This allows us to define the *transition matrix* P of a Markov chain as the matrix with entries

$$p_{ij} = \Pr[X_t = j | X_{t-1} = i].$$

The *initial distribution* of a Markov chain is given by the probabilities $\Pr[X_0 = i]$, for all $i \in S$. We denote the time-invariant Markov chain X with initial distribution λ and transition matrix P by $M(P, \lambda)$.

The random variable $T_{ij} = \min\{n \geq 1 | X_n = j, \text{ if } X_0 = i\}$ counts the number of steps needed by the Markov chain to get from i to j , and is called the *hitting time* from i to j . We set $T_{i,j} = \infty$ if state j is unreachable from i . Further, we define $h_{ij} = E[T_{ij}]$, and call $h_{ii} = E[T_{ii}]$ the (expected) return time for state $i \in S$.

Given P , the transition matrix of $M(P, \lambda)$, a *stationary distribution* of the Markov chain is a state vector π with $\pi = \pi P$. (We consider *row* vectors throughout the paper.) The intuition is that if the state vector of the Markov chain is π at time t , then it will remain π for all $t' > t$. Let $P^{(k)}$ be the transition matrix P multiplied by itself k times, and $p_{ij}^{(k)}$ be element (i, j) of $P^{(k)}$. A Markov chain is *irreducible* if for all pairs of states $i, j \in S$ there exists $m \geq 0$ such that $p_{ij}^{(m)} > 0$. (In other words, the underlying graph is strongly connected.) This implies that $T_{ij} < \infty$, and all expectations h_{ij} exist, for all $i, j \in S$. Furthermore, the following is known.

Theorem 6. *An irreducible finite Markov chain has a unique stationary distribution π , namely*

$$\pi_j = \frac{1}{h_{jj}}, \forall j \in S.$$

The periodicity of a state j is the maximum positive integer α such that $\{n \in \mathbb{N} | p_{jj}^{(n)} > 0\} \subseteq \{i\alpha | i \in \mathbb{N}\}$. A state with periodicity $\alpha = 1$ is called *aperiodic*. A Markov chain is *aperiodic* if all states are aperiodic. If a Markov chain has at least one self-loop, then it is aperiodic. A Markov chain that is irreducible and aperiodic is *ergodic*. Ergodic Markov chains converge to their stationary distribution as $t \rightarrow \infty$ independently of their initial distributions.

Theorem 7. *For every ergodic finite Markov chain $(X_t)_{t \in \mathbb{N}}$ we have independently of the initial distribution that $\lim_{t \rightarrow \infty} q_t = \pi$, where π denotes the chain's unique stationary distribution, and q_t is the distribution on states at time $t \in \mathbb{N}$.*

Ergodic Flow. It is often convenient to describe an ergodic Markov chain in terms of its *ergodic flow*: for each (directed) edge ij , we associate a flow $Q_{ij} = \pi_i p_{ij}$. These values satisfy $\sum_i Q_{ij} = \sum_i Q_{ji}$ and $\sum_{i,j} Q_{ij} = 1$. It also holds that $\pi_j = \sum_i Q_{ij}$.

Lifting Markov Chains. Let M and M' be ergodic Markov chains on finite state spaces S, S' , respectively. Let P, π be the transition matrix and stationary distribution for M , and P', π' denote the corresponding objects for M' . We say that M' is a *lifting* of M [8] if there is a function $f : S' \rightarrow S$ such that

$$Q_{ij} = \sum_{x \in f^{-1}(i), y \in f^{-1}(j)} Q'_{xy}, \forall i, j \in S.$$

Informally, M' is collapsed onto M by clustering several of its states into a single state, as specified by the function f . The above relation specifies a homomorphism on the ergodic flows. An immediate consequence of this relation is the following connection between the stationary distributions of the two chains.

Lemma 8. *For all $v \in S$, we have that*

$$\pi(v) = \sum_{x \in f^{-1}(v)} \pi'(x).$$

C Proofs Omitted from Section 3

Theorem 3. *Let \mathcal{S} be a stochastic scheduler with probability threshold $1 \geq \theta > 0$. Let A be an algorithm ensuring bounded minimal progress with a bound T . Then A ensures maximal progress with probability 1. Moreover, the expected maximal progress bound of A is at most $(1/\theta)^T$.*

Proof. Consider an interval of T steps in an execution of algorithm A . Our first observation is that, since A ensures T -bounded minimal progress, any process that performs T consecutive steps in this interval must complete a method invocation. To prove this fact, we consider cases on the minimal progress condition. If the minimal progress condition is T -bounded *deadlock-freedom* or *lock-freedom*, then every sequence of T steps by the algorithm must complete some method invocation. In particular, T steps by a single process must complete a method invocation. Obviously, this completed method invocation must be by the process itself. If the progress condition is T -bounded *clash-freedom*, then the claim follows directly from the definition.

Next, we show that, since \mathcal{S} is a stochastic scheduler with positive probability threshold, each correct process will eventually be scheduled for T consecutive steps, with probability 1. By the weak fairness condition in the definition, for every time step τ , every active process $p_i \in A_\tau$ is scheduled with probability at least $\theta > 0$. A process p_i is *correct* if $p_i \in A_\tau$, for all $\tau \geq 1$. By the definition, at each time step τ , each correct process $p_i \in A_\tau$ is scheduled for T consecutive time units with probability at least $\theta^T > 0$. From the previous argument, it follows that every correct process eventually completes each of its method calls with probability 1. By the same argument, the expected completion time for a process is at most $(1/\theta)^T$.

□*Theorem 3*

As an example for the above result, consider a simple bounded lock-free algorithm in which each operation attempts to increment the value of a single common CAS object. A successful CAS implies that the operation terminates, while after an unsuccessful CAS the operation re-tries to increment the CAS, taking the value it has received as the expected value for its next attempt. It is easy to see that this is a bounded lock-free algorithm, since after at most n attempts to change the value of the CAS object, some operation is successful and terminates, giving $T = n$ and implying a n^n bound for maximal progress when $\theta = 1/n$. However, notice that this algorithm needs a process to perform only two steps in sequence in order to complete an operation, giving $T' = 2$ and implying a n^2 bound for maximal progress when $\theta = 1/n$.

Unbounded minimal progress. We also show that the finite bound for minimal progress is necessary. For this, we convert the above algorithm into an *unbounded* lock-free algorithm which is not wait-free with probability > 0 . For simplicity, we consider the *uniform* stochastic scheduler, defined in Section 2.3. When an operation is unsuccessful in its attempt to increment the CAS, it takes $n^2 \cdot v$ dummy-steps before its next attempt, where v is the value returned by its latest unsuccessful CAS operation (see Algorithm 2).

```

1 Shared: CAS object  $C$ , initially 0
2 Register  $R$ 
3 Local: Integers  $v, val, j$ , initially 0
4 while true do
5      $val \leftarrow \text{CAS}(C, v, v + 1)$ 
6     if  $val = v$  then return
7     else
8          $v \leftarrow val$ 
9         for  $j = 1 \dots n^2v$  do  $\text{read}(R)$ 

```

Algorithm 2: An unbounded lock-free algorithm.

While this is an engineered example, it shows that we cannot hope for a result that is analogous to Theorem 3 for unbounded progress conditions, since such algorithms can have a “rich get richer” property. That is, a process which completes its operation has an increasing chance of completing again and again.

Lemma 2. *There exists an unbounded lock-free algorithm that is not wait-free with high probability.*

Proof. Consider the initial state of Algorithm 2. With probability at least $1/n$, each process p_i can be the first process to take a step, performing a successful CAS operation. Assume process p_1 takes the first step. Conditioned on this event, let P be the probability that p_1 is not the next process that performs a successful CAS operation. If p_1 takes a step in any of the next $n^2 \cdot v$ steps, then it is the next process that wins the CAS. The probability that this does not happen is at most $(1 - 1/n)^{n^2}$. Summing over all iterations, the probability that p_1 ever performs an unsuccessful CAS is therefore at most $\sum_{\ell=1}^{\infty} (1 - 1/n)^{n^2 \cdot \ell} \leq 2(1 - 1/n)^{n^2} \leq 2e^{-n}$. Hence, with probability at least $1 - 2e^{-n}$, process p_1 always wins the CAS, while other processes never do. This implies that the algorithm is not wait-free, with high probability. $\square_{\text{Lemma 2}}$

D Complete Analysis of $SCU(q, s)$

We now analyze the performance of algorithms in $SCU(q, s)$ under the uniform stochastic scheduler. We assume that all threads execute a method call of fixed length: in particular, the preamble is of length q , and the scan region has length s . See Figure 4 for an illustration of the code. Each thread executes an infinite number of such operations. We examine two parameters: system latency, i.e., how often (in terms of system steps) does a new operation complete, and individual latency, i.e., how often does a *certain thread* complete a new operation. Notice that the worst-case bound for the whole system is $\Theta(n)$ steps, while the worst-case bound for an individual thread is infinite, as the algorithm is not wait-free. We will prove the following result:

Theorem 4. *Let A be an algorithm in $SCU(q, s)$. Then, under the uniform stochastic scheduler, the system latency of A is $O(q + s\sqrt{n})$, and the individual latency is $O(n(q + s\sqrt{n}))$.*

We prove the upper bound by splitting the class $SCU_{q,s}$ into two separate parts, and analyzing each under the uniform scheduler. The first part is the loop code, which we call the *scan-validate* component. The second part is the *parallel code*, which we use to characterize the performance of the preamble code. In other words, we first consider $SCU(0, s)$ and then $SCU(q, 0)$.

```

1 Shared: register  $R$ 
2 Local:  $v$ , initially  $\perp$ 
3   procedure scan-validate()
4     while true do
5        $v \leftarrow R.\text{read}(); v' \leftarrow \text{new value based on } v$ 
6        $\text{flag} \leftarrow \text{CAS}(R, v, v')$ 
7       if  $\text{flag} = \text{true}$  then
8         output success

```

Algorithm 3: The scan-validate pattern.

D.1 The Scan-Validate Component

Notice that, without loss of generality, we can simplify the pseudocode to contain a single read step before the CAS. We will obtain the performance bound for this simplified algorithm, and then multiply it by s , the number of steps performed in the scan section. That is, we start by analyzing $SCU(0, 1)$ and then generalize to $SCU(0, s)$.

D.1.1 Markov Chain Representation

We define the *extended local state* of a process in terms of the state of the system, and of the type of step it is about to take. Thus, a process can be in one of three states: either it performs a read, or it CAS-es with the current value of R , or it CAS-es with an invalid value of R . Therefore, the state of the system after each step is completely described by the n extended local states of processes. We emphasize that this is different than what is typically referred to as the “local” state of a process, in that the extended local state is a local state as described from the viewpoint of the entire system. That is, a process that has a pending CAS operation can be in either of two different extended local states, depending on whether its CAS will succeed or not. This is determined by the state of the entire system. A key observation is that, although the local state of a process can only change when it takes a step, its extended local state can change also when another process takes a step.

The individual chain. Since the scheduler is uniform, the state of the system can be described as a Markov chain, where each state specifies the extended local state of each process. More precisely, a process is in state $OldCAS$ if it is about to CAS with an old value, it is in state $Read$ if it is about to read, and is in state $CCAS$ if it about to CAS with the current value. (Once CAS-ing the process returns to state $Read$.)

A state S of the individual chain is given by a combination of n states $S = (P_1, P_2, \dots, P_n)$, describing the extended local state of each process, where, for each $i \in \{1, \dots, n\}$, $P_i \in \{OldCAS, Read, CCAS\}$ is the extended local state of process p_i . There are $3^n - 1$ possible states, since the state where each process CAS-es with an old value cannot occur. In each transition, each process takes a step, and the state changes correspondingly. Recall that, for every $i \in \{1, \dots, n\}$, process p_i takes a step with probability $1/n$. The transitions in the individual chain are as follows. If the process p_i taking a step is in state $Read$ or $OldCAS$, then all other processes remain in the same extended local state, and p_i moves to state $CCAS$ or $Read$, respectively. If the process p_i taking a step is in state $CCAS$, then all processes in state $CCAS$ move to state $OldCAS$, and p_i moves to state $Read$.

The system chain. To reduce the complexity of the individual Markov chain, we introduce a simplified representation, which tracks system-wide progress. More precisely, each state of the system chain tracks

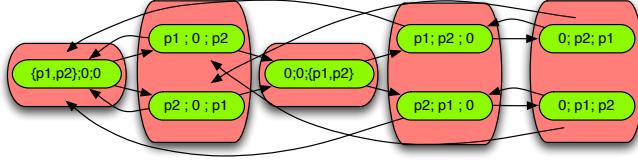


Figure 3: The individual chain and the global chain for two processes. Each transition has probability $1/2$. The red clusters are the states in the system chain. The notation $X; Y; Z$ means that processes in X are in state *Read*, processes in Y are in state *OldCAS*, and processes in Z are in state *CCAS*.

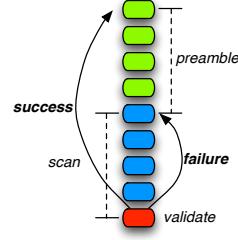


Figure 4: Structure of an algorithm in $SCU(q, s)$.

the number of processes in each state, irrespective of their identifiers: for any $a, b \in \{0, \dots, n\}$, a state x is defined by the tuple (a, b) , where a is the number of processes that are in state *Read*, and b is the number of processes that are in state *OldCAS*. Notice that the remaining $n - a - b$ processes must be in state *CCAS*. The initial state is $(n, 0)$, i.e. all processes are about to read. The state $(0, n)$ does not exist. The transitions in the system chain are as follows.

$$\Pr[(a+1, b-1)|(a, b)] = b/n, \text{ where } 0 \leq a \leq n \text{ and } b > 0.$$

$$\Pr[(a+1, b)|(a, b)] = 1 - (a+b)/n, \text{ where } 0 \leq a < n.$$

$$\Pr[(a-1, b)|(a, b)] = 1 - a/n, \text{ where } 0 < a \leq n.$$

See Figure 3 for a simple example of the two chains, and their lifting.

D.1.2 Analysis Preliminaries

First, we notice that both the individual chain and the system chain are ergodic.

Lemma 9. *For any $n \geq 1$, the individual chain and the system chain are ergodic.*

Let π be the stationary distribution of the system chain, and let π' be the stationary distribution for the individual chain. For any state $k = (a, b)$ in the system chain, let π_k be its probability in the stationary distribution. Similarly, for state x in the individual chain, let π'_x be its probability in the stationary distribution.

We now prove that there exists a *lifting* from the individual chain to the system chain. Intuitively, the lifting from the individual chain to the system chain collapses all states in which a processes are about to read and b processes are about to CAS with an old value (the identifiers of these processes are different for distinct states), into state (a, b) from the system chain.

Definition 2. *Let \mathcal{S} be the set of states of the individual chain, and \mathcal{M} be the set of states of the system chain. We define the function $f : \mathcal{S} \rightarrow \mathcal{M}$ such that each state $S = (P_1, \dots, P_n)$, where a processes are in state *Read* and b processes are in state *OldCAS*, is taken into state (a, b) of the system chain.*

We then obtain the following relation between the stationary distributions of the two chains.

Lemma 10. *For every state k in the system chain, we have $\pi_k = \sum_{x \in f^{-1}(k)} \pi'_x$.*

Proof. We obtain this relation algebraically, starting from the formula for the stationary distribution of the individual chain. We have that $\pi' A = \pi'$, where π' is a row vector, and A is the transition matrix of the individual chain. We partition the states of the individual chain into sets, where $G_{a,b}$ is the set of system states S such that $f(S) = (a, b)$. Fix an arbitrary ordering $(G_k)_{k \geq 1}$ of the sets, and assume without loss of generality that the system states are ordered according to their set in the vector π and in the matrix A , so that states mapping to the same set are consecutive.

Let now A' be the transition matrix across the sets $(G_k)_{k \geq 1}$. In particular, a'_{kj} is the probability of moving from a state in the set G_k to some state in the set G_j . Note that this transition matrix is the same as that of the system chain. Pick an arbitrary state x in the individual chain, and let $f(x) = (a, b)$. In other words, state x maps to set G_k , where $k = (a, b)$. We claim that for every set G_j , $\sum_{y \in G_j} \Pr[y|x] = \Pr[G_j|G_i]$.

To see this, fix $x = (P_0, P_1, \dots, P_n)$. Since $f(x) = (a, b)$, there are exactly b distinct states y reachable from x such that $f(y) = (a+1, b-1)$: the states where a process in extended local state *OldCAS* takes a step. Therefore, the probability of moving to such a state y is b/n . Similarly, the probability of moving to a state y with $f(y) = (a+1, b-1)$ is $1 - (a+b)/n$, and the probability of moving to a state y with $f(y) = (a-1, b)$ is a/n . All other transition probabilities are 0.

To complete the proof, notice that we can collapse the stationary distribution π' onto the row vector $\bar{\pi}$, where the k th element of $\bar{\pi}$ is $\sum_{x \in G_k} \pi'_x$. Using the above claim and the fact that $\pi' A = \pi'$, we obtain by calculation that $\bar{\pi} A' = \bar{\pi}$. Therefore, $\bar{\pi}$ is a stationary distribution for the system chain. Since the stationary distribution is unique, $\bar{\pi} = \pi$, which concludes the proof. \square

In fact, we can prove that the function $f : \mathcal{S} \rightarrow \mathcal{M}$ defined above induces a lifting from the individual chain to the system chain.

Lemma 3. *The system Markov chain is a lifting of the individual Markov chain.*

Proof. Consider a state k in \mathcal{M} . Let j be a neighboring state of k in the system chain. The ergodic flow from k to j is $p_{kj}\pi_k$. In particular, if k is given by the tuple (a, b) , j can be either $(a+1, b-1)$ or $(a+1, b)$, or $(a-1, b)$. Consider now a state $x \in \mathcal{M}$, $x = (P_0, \dots, P_n)$, such that $f(x) = k$. By the definition of f , x has a processes in state *Read*, and b processes in state *OldCAS*.

If j is the state $(a+1, b-1)$, then the flow from k to j , Q_{kj} , is $b\pi_k/n$. The state x from the individual chain has exactly b neighboring states y which map to the state $(a+1, b-1)$, one for each of the b processes in state *OldCAS* which might take a step. Fix y to be such a state. The probability of moving from x to y is $1/n$. Therefore, using Lemma 10, we obtain that

$$\sum_{x \in f^{-1}(k), y \in f^{-1}(j)} Q'_{xy} = \sum_{x \in f^{-1}(k)} \sum_{y \in f^{-1}(j)} \frac{1}{n} \pi'_x = \frac{b}{n} \sum_{x \in f^{-1}(k)} \pi'_x = \frac{b}{n} \pi_k = Q_{kj}.$$

The other cases for state j follow similarly. Therefore, the lifting condition holds. $\square_{\text{Lemma 3}}$

Next, we notice that, since states from the individual chain which map to the same system chain state are symmetric, their probabilities in the stationary distribution must be the same.

Lemma 11. *Let x and x' be two states in \mathcal{S} such that $f(x) = f(y)$. Then $\pi'_x = \pi'_y$.*

Proof (Sketch). The proof follows by noticing that, for any $i, j \in \{1, 2, \dots, n\}$, switching indices i and j in the Markov chain representation maintains the same transition matrix. Therefore, the stationary probabilities for symmetric states (under the swapping of process ids) must be the same. \square

We now put together the previous claims to obtain an upper bound on the expected time between two successes for a specific process.

Lemma 4. *Let W be the expected system steps between two successes in the stationary distribution of the system chain. Let W_i be the expected system steps between two successes of process p_i in the stationary distribution of the individual chain. For every process p_i , $W = nW_i$.*

Proof. Let μ be the probability that a step is a success by *some* process. Expressed in the system chain, we have that $\mu = \sum_{j=(a,b)} (1 - (a+b)/n) \pi_j$. Let X_i be the set of states in the individual chain in which $P_i = CCAS$. Consider the event that a system step is a step in which p_i succeeds. This must be a step by p_i from a state in X_i . The probability of this event in the stationary distribution of the individual chain is $\eta_i = \sum_{x \in X_i} \pi'_x / n$.

Recall that the lifting function f maps all states x with a processes in state *Read* and b processes in state *OldCAS* to state $j = (a, b)$. Therefore, $\eta_i = (1/n) \sum_{j=(a,b)} \sum_{x \in f^{-1}(j) \cap X_i} \pi'_x$. By symmetry, we have that $\pi'_x = \pi'_y$, for every states $x, y \in f^{-1}(j)$. The fraction of states in $f^{-1}(j)$ that have p_i in state *CCAS* (and are therefore also in X_i) is $(1 - (a+b)/n)$. Therefore, $\sum_{x \in f^{-1}(j) \cap X_i} \pi'_x = (1 - (a+b)/n) \pi_j$.

We finally get that, for every process p_i , $\eta_i = (1/n) \sum_{j=(a,b)} (1 - (a+b)/n) \pi_j = (1/n) \mu$. On the other hand, since we consider the stationary distribution, from a straightforward extension of Theorem 1, we have that $W_i = 1/\eta_i$, and $W = 1/\mu$. Therefore, $W_i = nW$, as claimed. $\square_{\text{Lemma 4}}$

D.1.3 System Latency Bound

In this section we provide an upper bound on the quantity W , the expected number of system steps between two successes in stationary distribution of the system chain. We prove the following.

Theorem 5. *The expected number of steps between two successes in the system chain is $O(\sqrt{n})$.*

An iterated balls-into-bins game. To bound W , we model the evolution of the system as a balls-into-bins game. We will associate each process with a bin. At the beginning of the execution, each bin already contains one ball. At each time step, we throw a new ball into a uniformly chosen random bin. Essentially, whenever the process takes a step, its bin receives an additional ball. We continue to distribute balls until the first time a bin acquires *three* balls. We call this event a *reset*. When a reset occurs, we set the number of balls in the bin containing three balls to one, and all the bins containing two balls become empty. The game then continues until the next reset.

This game models the fact that initially, each process is about to read the shared state, and must take two steps in order to update its value. Whenever a process changes the shared state by CAS-ing successfully, all other processes which were CAS-ing with the correct value are going to fail their operations; in particular, they now need to take three steps in order to change the shared state. We therefore reset the number of balls in the corresponding bins to 0.

More precisely, we define the game in terms of *phases*. A phase is the interval between two resets. For phase i , we denote by a_i the number of bins with one ball at the beginning of the phase, and by b_i the number of bins with 0 balls at the beginning of the phase. Since there are no bins with two or more balls at the start of a phase, we have that $a_i + b_i = n$.

It is straightforward to see that this random process evolves in the same way as the system Markov chain. In particular, notice that the bound W is the expected length of a phase. To prove Theorem 5, we first obtain a bound on the length of a phase.

Lemma 5. Let $\alpha \geq 4$ be a constant. The expected length of phase i is at most $\min(2\alpha n/\sqrt{a_i}, 3\alpha n/b_i^{1/3})$. The phase length is $2\alpha \min(n\sqrt{\log n}/\sqrt{a_i}, n(\log n)^{1/3}/b_i^{1/3})$, with probability at least $1 - 1/n^\alpha$. The probability that the length of a phase is less than $\min(n/\sqrt{a_i}, n/(b_i)^{1/3})/\alpha$ is at most $1/(4\alpha^2)$.

Proof. Let A_i be the set of bins with one ball, and let B_i be the set of bins with zero balls, at the beginning of the phase. We have $a_i = |A_i|$ and $b_i = |B_i|$. Practically, the phase ends either when a bin in A_i or a bin in B_i first contains three balls.

For the first event to occur, some bin in A_i must receive two additional balls. Let $c \geq 1$ be a large constant, and assume for now that $a_i \geq \log n$ and $b_i \geq \log n$ (the other cases will be treated separately). The number of bins in A_i which need to receive a ball before some bin receives two new balls is concentrated around $\sqrt{a_i}$, by the birthday paradox. More precisely, the following holds.

Claim 1. Let X_i be random variable counting the number of bins in A_i chosen to get a ball before some bin in A_i contains three balls, and fix $\alpha \geq 4$ to be a constant. Then the expectation of X_i is less than $2\alpha\sqrt{a_i}$. The value of X_i is at most $\alpha\sqrt{a_i \log n}$, with probability at least $1 - 1/n^{\alpha^2}$.

Proof. We employ the Poisson approximation for balls-into-bins processes. In essence, we want to bound the number of balls to be thrown uniformly into a_i bins until two balls collide in the same bin, in expectation and with high probability. Assume we throw m balls into the $a_i \geq \log n$ bins. It is well-known that the number of balls a bin receives during this process can be approximated as a Poisson random variable with mean m/a_i (see, e.g., [17]). In particular, the probability that no bin receives two extra balls during this process is at most

$$2 \left(1 - \frac{e^{-m/a_i} (\frac{m}{a_i})^2}{2} \right)^{a_i} \leq 2 \left(\frac{1}{e} \right)^{\frac{m^2}{2a_i}} e^{-m/a_i}.$$

If we take $m = \alpha\sqrt{a_i}$ for $\alpha \geq 4$ constant, we obtain that this probability is at most

$$2 \left(\frac{1}{e} \right)^{\alpha^2 e^{-\alpha/\sqrt{a_i}}/2} \leq \left(\frac{1}{e} \right)^{\alpha^2/4},$$

where we have used the fact that $a_i \geq \log n \geq \alpha^2$. Therefore, the expected number of throws until some bin receives two balls is at most $2\alpha\sqrt{a_i}$. Taking $m = \alpha\sqrt{a_i \log n}$, we obtain that some bin receives two new balls within $\alpha\sqrt{a_i \log n}$ throws with probability at least $1 - 1/n^{\alpha^2}$. \square

We now prove a similar upper bound for the number of bins in B_i which need to receive a ball before some such bin receives three new balls, as required to end the phase.

Claim 2. Let Y_i be random variable counting the number of bins in B_i chosen to get a ball before some bin in B_i contains three balls, and fix $\alpha \geq 4$ to be a constant. Then the expectation of Y_i is at most $3\alpha b_i^{2/3}$, and Y_i is at most $\alpha(\log n)^{1/3} b_i^{2/3}$, with probability at least $1 - (1/n)^{\alpha^3/54}$.

Proof. We need to bound the number of balls to be thrown uniformly into b_i bins (each of which is initially empty), until some bin gets three balls. Again, we use a Poisson approximation. We throw m balls into the $b_i \geq \log n$ bins. The probability that no bin receives three or more balls during this process is at most

$$2 \left(1 - \frac{e^{-m/b_i} (m/b_i)^3}{6} \right)^{b_i} = 2 \left(\frac{1}{e} \right)^{\frac{m^3}{6b_i^2}} e^{-m/b_i}.$$

Taking $m = \alpha b_i^{2/3}$ for $\alpha \geq 4$, we obtain that this probability is at most

$$2 \left(\frac{1}{e} \right)^{\frac{\alpha^3}{6} e^{-\alpha/b_i^{1/3}}} \leq \left(\frac{1}{e} \right)^{\alpha^3/54}.$$

Therefore, the expected number of ball thrown into bins from B_i until some such bin contains three balls is at most $3\alpha b_i^{2/3}$. Taking $m = \alpha(\log n)^{1/3} b_i^{2/3}$, we obtain that the probability that no bin receives three balls within the first m ball throws in B_i is at most $(1/n)^{\alpha^3/54}$. \square

The above claims bound the number of steps inside the sets A_i and B_i necessary to finish the phase. On the other hand, notice that a step throws a new ball into a bin from A_i with probability a_i/n , and throws it into a bin in B_i with probability b_i/n . It therefore follows that the expected number of steps for a bin in A_i to reach three balls (starting from one ball in each bin) is at most $2\alpha\sqrt{a_i}n/a_i = 2\alpha n/\sqrt{a_i}$. The expected number of steps for a bin in B_i to reach three balls is at most $3\alpha b_i^{2/3}n/b_i = 3\alpha n/b_i^{1/3}$. The next claim provides concentration bounds for these inequalities, and completes the proof of the Lemma.

Claim 3. *The probability that the system takes more than $2\alpha \frac{n}{\sqrt{a_i}} \sqrt{\log n}$ steps in a phase is at most $1/n^\alpha$. The probability that the system takes more than $2\alpha \frac{n}{b_i^{1/3}} (\log n)^{1/3}$ steps in a phase is at most $1/n^\alpha$.*

Proof. Fix a parameter $\beta > 0$. By a Chernoff bound, the probability that the system takes more than $2\beta n/a_i$ steps without throwing at least β balls into the bins in A_i is at most $(1/e)^\beta$. At the same time, by Claim 1, the probability that $\alpha\sqrt{a_i} \log n$ balls thrown into bins in A_i do not generate a collision (finishing the phase) is at most $1/n^{\alpha^2}$.

Therefore, throwing $2\alpha \frac{n}{\sqrt{a_i}} \sqrt{\log n}$ balls fail to finish the phase with probability at most $1/n^{\alpha^2} + 1/e^{\alpha\sqrt{a_i} \log n}$. Since $a_i \geq \log n$ by the case assumption, the claim follows.

Similarly, using Claim 2, the probability that the system takes more than $2\alpha(\log n)^{1/3} b_i^{2/3} n/b_i = 2\alpha(\log n)^{1/3} n/b_i^{1/3}$ steps without a bin in B_i reaching three balls (in the absence of a reset) is at most $(1/e)^{1+(\log n)^{1/3} b_i^{2/3}} + (1/n)^{\alpha^3/54} \leq (1/n)^\alpha$, since $b_i \geq \log n$. \square

We put these results together to obtain that, if $a_i \geq \log n$ and $b_i \geq \log n$, then the expected length of a phase is $\min(2\alpha n/\sqrt{a_i}, 3\alpha n/b_i^{1/3})$. The phase length is $2\alpha \min(\frac{n}{\sqrt{a_i}} \sqrt{\log n}, \frac{n}{b_i^{1/3}} (\log n)^{1/3})$, with high probability.

It remains to consider the case where either a_i or b_i are less than $\log n$. Assume $a_i \geq \log n$. Then $b_i \geq n - \log n$. We can therefore apply the above argument for b_i , and we obtain that with high probability the phase finishes in $2\alpha n(\log n/b_i)^{1/3}$ steps. This is less than $2\alpha \frac{n}{\sqrt{a_i}} \sqrt{\log n}$, since $a_i \leq \log n$, which concludes the claim. The converse case is similar. $\square_{\text{Lemma 5}}$

Returning to the proof, we characterize the dynamics of the phases $i \geq 1$ based on the value of a_i at the beginning of the phase. We say that a phase i is in *the first range* if $a_i \in [n/3, n]$. Phase i is in *the second range* if $n/c \leq a_i < n/3$, where c is a large constant. Finally, phase i is in *the third range* if $0 \leq a_i < n/c$. Next, we characterize the probability of moving between phases.

Lemma 6. *For $i \geq 1$, if phase i is in the first two ranges, then the probability that phase $i+1$ is in the third range is at most $1/n^\alpha$. Let $\beta > 2c^2$ be a constant. The probability that $\beta\sqrt{n}$ consecutive phases are in the third range is at most $1/n^\alpha$.*

Proof. We first bound the probability that a phase moves to the third range from one of the first two ranges.

Claim 4. *For $i \geq 1$, if phase i is in the first two ranges, then the probability that phase $i + 1$ is in the third range is at most $1/n^\alpha$.*

Proof. We first consider the case where phase i is in range two, i.e. $n/c \leq a_i < n/3$, and bound the probability that $a_{i+1} < n/c$. By Lemma 5, the total number of system steps taken in phase i is at most $2\alpha \min(n/\sqrt{a_i} \sqrt{\log n}, n/b_i^{1/3} (\log n)^{1/3})$, with probability at least $1 - 1/n^\alpha$. Given the bounds on a_i , it follows by calculation that the first factor is always the minimum in this range.

Let ℓ_i be the number of steps in phase i . Since $a_i \in [n/c, n/3]$, the expected number of balls thrown into bins from A_i is at most $\ell_i/3$, whereas the expected number of balls thrown into bins from B_i is at least $2\ell_i/3$. The parameter a_{i+1} is a_i plus the bins from B_i which acquire a single ball, minus the balls from A_i which acquire an extra ball. On the other hand, the number of bins from B_i which acquire a single ball during ℓ_i steps is tightly concentrated around $2\ell_i/3$, whereas the number of bins in A_i which acquire a single ball during ℓ_i steps is tightly concentrated around $\ell_i/3$. More precisely, using Chernoff bounds, given $a_i \in [n/c, n/3]$, we obtain that $a_i \geq a_{i+1}$, with probability at least $1 - 1/e^{\alpha\sqrt{n}}$.

For the case where phase i is in range one, notice that, in order to move to range three, the value of a_i would have to decrease by at least $n(1/3 - 1/c)$ in this phase. On the other hand, by Lemma 5, the length of the phase is at most $2\alpha\sqrt{3n \log n}$, w.h.p. Therefore the claim follows. A similar argument provides a lower bound on the length of a phase. \square

The second claim suggests that, if the system is in the third range (a low probability event), it gradually returns to one of the first two ranges.

Claim 5. *Let $\beta > 2c^2$ be a constant. The probability that $\beta\sqrt{n}$ phases are in the third range is at most $1/n^\alpha$.*

Proof. Assume the system is in the third range, i.e. $a_i \in [0, n/c]$. Fix a phase i , and let ℓ_i be its length. Let S_b^i be the set of bins in B_i which get a single ball during phase i . Let T_b^i be the set of bins in B_i which get two balls during phase i (and are reset). Let S_a^i be the set of bins in A_i which get a single ball during phase i (and are also reset). Then $b_i - b_{i+1} \geq |S_b^i| - |T_b^i| - |S_a^i|$.

We bound each term on the right-hand side of the inequality. Of all the balls thrown during phase i , in expectation at least $(1 - 1/c)$ are thrown in bins from B_i . By a Chernoff bound, the number of balls thrown in B_i is at least $(1 - 1/c)(1 - \delta)\ell_i$ with probability at least $1 - \exp(-\delta^2\ell_i(1 - 1/c)/4)$, for $\delta \in (0, 1)$. On the other hand, the majority of these balls do not cause collisions in bins from B_i . In particular, from the Poisson approximation, we obtain that $|S_b^i| \geq 2|T_b^i|$ with probability at least $1 - (1/n)^{\alpha+1}$, where we have used $b_i \geq n(1 - 1/c)$.

Considering S_a^i , notice that, w.h.p., at most $(1 + \delta)\ell_i/c$ balls are thrown in bins from A_i . Summing up, given that $\ell_i \geq \sqrt{n}/c$, we obtain that $b_i - b_{i+1} \geq (1 - 1/c)(1 - \delta)\ell_i/2 - (1 + \delta)\ell_i/c$, with probability at least $1 - \max((1/n)^\alpha, \exp(-\delta^2\ell_i(1 - 1/c)/4))$. For small $\delta \in (0, 1)$ and $c \geq 10$, the difference is at least ℓ_i/c^2 . Notice also that the probability depends on the length of the phase.

We say that a phase is *regular* if its length is at least $\min(n/\sqrt{a_i}, n/(b_i)^{1/3})/c$. From Lemma 5, the probability that a phase is regular is at least $1 - 1/(4c^2)$. Also, in this case, $\ell_i \geq \sqrt{n}/c$, by calculation. If the phase is regular, then the size of b_i decreases by $\Omega(\sqrt{n})$, w.h.p.

If the phase is not regular, we simply show that, with high probability, a_i does not decrease. Assume $a_i < a_{i+1}$. Then, either $\ell_i < \log n$, which occurs with probability at most $1/n^{\Omega(\log n)}$ by Lemma 5, or the inequality $b_i - b_{i+1} \geq \ell_i/c^2$ fails, which also occurs with probability at most $1/n^{\Omega(\log n)}$.

To complete the proof, consider a series of $\beta\sqrt{n}$ consecutive phases, and assume that a_i is in the third range for all of them. The probability that such a phase is regular is at least $1 - 1/(4c^2)$, therefore, by Chernoff, a constant fraction of phases are regular, w.h.p. Also w.h.p., in each such phase the size of b_i goes down by $\Omega(\sqrt{n})$ units. On the other hand, by the previous argument, if the phases are not regular, then it is still extremely unlikely that b_i increases for the next phase. Summing up, it follows that the probability that the system stays in the third range for $\beta\sqrt{n}$ consecutive phases is at most $1/n^\alpha$, where $\beta \geq 2c^2$, and $\alpha \geq 4$ was fixed initially. \square

$\square_{\text{Lemma 6}}$

Final argument. To complete the proof of Theorem 5, recall that we are interested in the expected length of a phase. To upper bound this quantity, we group the states of the game according to their range as follows: state $S_{1,2}$ contains all states (a_i, b_i) in the first two ranges, i.e. with $a_i \geq n/c$. State S_3 contains all states (a_i, b_i) such that $a_i < n/c$. The expected length of a phase starting from a state in $S_{1,2}$ is $O(\sqrt{n})$, from Lemma 5. However, the phase length could be $\omega(\sqrt{n})$ if the state is in S_3 . We can mitigate this fact given that the probability of moving to range three is low (Claim 4), and the system moves away from range three rapidly (Claim 5): intuitively, the probability of states in S_3 in the stationary distribution has to be very low.

To formalize the argument, we define two Markov chains. The first Markov chain M has two states, $S_{1,2}$ and S_3 . The transition probability from $S_{1,2}$ to S_3 is $1/n^\alpha$, whereas the transition probability from S_3 to $S_{1,2}$ is $x > 0$, fixed but unknown. Each state loops onto itself, with probabilities $1 - 1/n^\alpha$ and $1 - x$, respectively. The second Markov chain M' has two states S and R . State S has a transition to R , with probability $\beta\sqrt{n}/n^\alpha$, and a transition to itself, with probability $1 - \beta\sqrt{n}/n^\alpha$. State R has a loop with probability $1/n^\alpha$, and a transition to S , with probability $1 - 1/n^\alpha$.

It is easy to see that both Markov chains are ergodic. Let $[s \ r]$ be the stationary distribution of M' . Then, by straightforward calculation, we obtain that $s \geq 1 - \beta\sqrt{n}/n^\alpha$, while $r \leq \beta\sqrt{n}/n^\alpha$.

On the other hand, notice that the probabilities in the transition matrix for M' correspond to the probabilities in the transition matrix for $M^{\beta\sqrt{n}}$, i.e. M applied to itself $\beta\sqrt{n}$ times. This means that the stationary distribution for M is the same as the stationary distribution for M' . In particular, the probability of state $S_{1,2}$ is at least $1 - \beta\sqrt{n}/n^\alpha$, and the probability of state S_3 is at most $\beta\sqrt{n}$.

To conclude, notice that the expected length of a phase is at most the expected length of a phase in the first Markov chain M . Using the above bounds, this is at most $2\alpha\sqrt{n}(1 - \beta\sqrt{n}/n^\alpha) + \beta n^{2/3}\sqrt{n}/n^\alpha = O(\sqrt{n})$, as claimed. This completes the proof of Theorem 5.

D.2 Parallel Code

We now use the same framework to derive a convergence bound for parallel code, i.e. a method call which completes after the process executes q steps, irrespective the concurrent actions of other processes. The pseudocode is given in Algorithm 4.

Analysis. We now analyze the individual and system latency for this algorithm under the uniform stochastic scheduler. Again, we start from its Markov chain representation. We define the individual Markov chain M_I to have states $S = (C_1, \dots, C_n)$, where $C_i \in \{0, \dots, q-1\}$ is the current step counter for process p_i . At every step, the Markov chain picks i from 1 to n uniformly at random and transitions into the state $(C_1, \dots, (C_i + 1) \bmod q, \dots, C_n)$. A process registers a success every time its counter is reset to 0; the system registers a success every time some process counter is reset to 0. The system latency is the expected number of system steps between two successes, and the individual latency is the expected number of system steps between two successes by a specific process.

```

1 Shared: register  $R$ 
2   procedure call()
3     while true do
4       for  $i$  from 1 to  $q$  do
5         Execute  $i$ th step
6       output success

```

Algorithm 4: Pseudocode for parallel code.

We now define the system Markov chain M_S , as follows. A state $g \in M_S$ is given by q values $(v_0, v_1, \dots, v_{q-1})$, where for each $j \in \{0, \dots, q-1\}$ v_j is the number of processes with step counter value j , with the condition that $\sum_{j=0}^{q-1} v_j = n$. Given a state $(v_0, v_1, \dots, v_{q-1})$, let X be the set of indices $i \in \{0, \dots, q-1\}$ such that $v_i > 0$. Then, for each $i \in X$, the system chain transitions into the state $(v_0, \dots, v_i - 1, v_{i+1} + 1, \dots, v_{q-1})$ with probability v_i/n .

It is easy to check that both M_I and M_S are ergodic Markov chains. Let π be the stationary distribution of M_S , and π' be the stationary distribution of M_I . We next define the mapping $f : M_I \rightarrow M_S$ which maps each state $S = (C_1, \dots, C_n)$ to the state $(v_0, v_1, \dots, v_{q-1})$, where v_j is the number of processes with counter value j from S . Checking that this mapping is a lifting between M_I and M_S is straightforward.

Lemma 12. *The function f defined above is a lifting between the ergodic Markov chains M_I and M_S .*

We then obtain bounds on the system and individual latency.

Lemma 7. *For any $1 \leq i \leq n$ and $q \geq 0$, given an algorithm in $SCU(q, 0)$, its individual latency is $W_i = nq$, and its system latency is $W = q$.*

Proof. We examine the stationary distributions of the two Markov chains. Contrary to the previous examples, it turns out that in this case it is easier to determine the stationary distribution of the individual Markov chain M_I . Notice that, in this chain, all states have in- and out-degree n , and the transition probabilities are uniform (probability $1/n$). It therefore must hold that the stationary distribution of M_I is *uniform*. Further, notice that a $1/nq$ fraction of the edges corresponds to the counter of a specific process p_i being reset. Therefore, for any i , the probability that a step in M_I is a completed operation by p_i is $1/nq$. Hence, the individual latency for the algorithm is nc . To obtain the system latency, we notice that, from the lifting, the probability that a step in M_S is a completed operation by *some* process is $1/q$. Therefore, the individual latency for the algorithm is q . \square Lemma 7

D.3 General Bound for $SCU(q, s)$

We now put together the results of the previous sections to obtain a bound on individual and system latency. First, we notice that Theorem 5 can be easily extended to the case where the loop contains s scan steps, as the extended local state of a process p can be changed by a step of another process $q \neq p$ only if p is about to perform a CAS operation.

Corollary 2. *For $s \geq 1$, given a scan-validate pattern with s scan steps under the stochastic scheduler, the system latency is $O(s\sqrt{n})$, while the individual latency is $O(ns\sqrt{n})$.*

Obviously, an algorithm in $SCU(q, s)$ is a sequential composition of parallel code followed by s loop steps. Fix a process p_i . By Lemma 7 and Corollary 2, by linearity of expectation, we obtain that the

expected individual latency for process p_i to complete an operation is at most $n(q + \alpha s\sqrt{n})$, where $\alpha \geq 4$ is a constant.

Consider now the Markov Chain M_S that corresponds to the sequential composition of the Markov chain for the parallel code M_P , and the Markov chain M_L corresponding to the loop. In particular, a completed operation from M_P does not loop back into the chain, but instead transitions into the corresponding state of M_L . More precisely, if the transition is a step by some processor p_i which completed step number q in the parallel code (and moves to the loop code), then the chain transitions into the state where processor p_i is about to execute the first step of the loop code. Similarly, when a process performs a successful CAS at the end of the loop, the processes' step counter is reset to 0, and its next operation will be the first step of the preamble.

It is straightforward that the chain M_S is ergodic. Let κ_i be the probability of the event that process p_i completes an operation in the stationary distribution of the chain M_S . Since the expected number of steps p_i needs to take to complete an operation is at most $n(q + \alpha s\sqrt{n})$, we have that $\kappa_i \geq 1/(n(q + \alpha s\sqrt{n}))$. Let κ be the probability of the event that *some* process completes an operation in the stationary distribution of the chain M_S . It follows that $\kappa = \sum_{i=1}^n \kappa_i \geq 1/(q + \alpha s\sqrt{n})$. Hence, the expected time until the system completes a new operation is at most $q + \alpha s\sqrt{n}$, as claimed.

E Application - A Fetch-and-Increment Counter using Augmented CAS

We now apply the ideas from the previous section to obtain minimal and maximal progress bounds for other lock-free algorithms under the uniform stochastic scheduler.

Some architectures support richer semantics for the CAS operation, which return the *current* value of the register which the operation attempts to modify. We can take advantage of this property to obtain a simpler fetch-and-increment counter implementation based on compare-and-swap. This type of counter implementation is very widely-used [4].

```

7 Shared: register  $R$ 
8   procedure fetch-and-inc()  $v \leftarrow 0$ 
9     while true do
10     $old \leftarrow v$ 
11     $v \leftarrow \text{CAS}(R, v, v + 1)$ 
12    if  $v = old$  then
13      output success

```

Algorithm 5: A lock-free fetch-and-increment counter based on compare-and-swap.

E.1 Markov Chain Representations

We again start from the observation the algorithm induces an individual Markov chain and a global one. From the point of view of each process, there are two possible states: *Current*, in which the process has the *current* value (i.e. its local value v is the same as the value of the register R), and the *Stale* state, in which the process has an old value, which will cause its CAS call to fail. (In particular, the *Read* and *OldCAS* states from the universal construction are coalesced.)

The Individual Chain. The per-process chain, which we denote by M_I , results from the composition of the automata representing the algorithm at each process. Each state of M_I can be characterized by the set

of processes that have the current value of the register R . The Markov chain has $2^n - 1$ states, since it never happens that *no thread* has the current value.

For each non-empty subset of processes S , let s_S be the corresponding state. The initial state is s_{Π} , the state in which every thread has the current value. We distinguish *winning* states as the states $(s_{\{p_i\}})_i$ in which only *one* thread has the current value: to reach this state, one of the processes must have successfully updated the value of R . There are exactly n winning states, one for each process.

Transitions are defined as follows. From each state s , there are n outgoing edges, one for each process which could be scheduled next. Each transition has probability $1/n$, and moves to state s' corresponding to the set of processes which have the current value at the next time step. Notice that the winning states are the only states with a self-loop, and that from every state s_S the chain either moves to a state s_V with $|V| = |S| + 1$, or to a winning state for one of the threads in S .

The Global Chain. The *global* chain M_G results from clustering the symmetric states from M_I into single states. The chain has n states v_1, \dots, v_n , where state v_i comprises all the states s_S in M_G such that $|S| = i$. Thus, state v_1 is the state in which *some* process just completed a new operation. In general, v_i is the state in which i processes have the current value of R (and therefore may commit an operation if scheduled next).

The transitions in the global chain are defined as follows. For any $1 \leq i \leq n$, from state v_i the chain moves to state v_1 with probability i/n . If $i < n$, the chain moves to state v_{i+1} with probability $1 - i/n$. Again, the state v_1 is the only state with a self-loop. The intuition is that some process among the i possessing the current value wins if scheduled next (and changes the current value); otherwise, if some other thread is scheduled, then that thread will also have the current value.

E.2 Algorithm Analysis

We analyze the stationary behavior of the algorithm under a uniform stochastic scheduler, assuming each process invokes an infinite number of operations.

Strategy. We are interested in the expected number of steps that some process p_i takes between committing two consecutive operations, in the stationary distribution. This is the *individual latency*, which we denote by W_i . As in Section 5.1.2, we proceed by first bounding the system latency W , which is easier to analyze, and then show that $W_i = nW$, i.e. the algorithm is *fair*. We will use the two Markov chain representations from the previous section. In particular, notice that W_i is the expected return time of the “win state” v_i of the global chain M_G , and W is the expected return time of the state s_{p_i} in which p_i just completed an operation.

The first claim is an upper bound on the return time for v_1 in M_G .

Lemma 13. *The expected return time for v_1 is $W \leq 2\sqrt{n}$.*

Proof. For $0 \leq i \leq n - 1$, let $Z(i)$ be the hitting time for state v_1 from the state where $n - i$ processes have the current value. In particular, $Z(0)$ is the hitting time from the state where *all* processes have the correct value, and therefore $Z(0) = 1$. Analyzing the transitions, we obtain that $Z(i) = iZ(i-1)/n + 1$. We prove that $Z(n-1) \leq 2\sqrt{n}$.

We analyze two intervals: k from 0 to $n - \sqrt{n}$, and then up to $n - 1$. We first claim that, for $0 \leq k \leq n - \sqrt{n}$, it holds that $Z(k) \leq \sqrt{n}$. We prove this by induction. The base case obviously holds. For the induction step, notice that $Z(k) \leq Z(k-1)(n - \sqrt{n})/n + 1$ in this interval. By the hypothesis, $Z(k-1) \leq \sqrt{n}$, therefore $Z(k) \leq \sqrt{n}$ for $k \leq n - \sqrt{n}$.

For $k \in \{n - \sqrt{n}, \dots, n\}$, notice that $Z(k)$ can add at most 1 at each iteration, and we are iterating at most \sqrt{n} times. This gives an upper bound of $2\sqrt{n}$, as claimed. \square

Remark. Intuitively, the value $Z(n - 1)$ is related to the birthday paradox, since it counts the number of elements that must be chosen uniformly at random from 1 to n (with replacement) until one of the elements appears twice. In fact, this is the Ramanujan Q function [5], which has been studied previously by Knuth [13] and Flajolet et al. [5] in relation to the performance of linear probing hashing. Its asymptotics are known to be $Z(n - 1) = \sqrt{\pi n/2}(1 + o(1))$ [5].

Markov Chain Lifting. We now analyze W_i , the expected number of total system steps for a specific process p_i to commit a new request. We define a mapping $f : M_I \rightarrow M_G$ between the states of the individual Markov chain. For any non-empty set S of processes, the function maps the state $s_S \in M_I$ to the state v_i of the chain. It is straightforward to prove that this mapping is a correct lifting of the Markov chain, and that both Markov chains are ergodic.

Lemma 14. *The individual chain and the local chain are ergodic. The function f is a lifting between the individual chain and the global chain.*

We then use the lifting and symmetry to obtain the following relation between the stationary distributions of the two Markov chains. The proof is similar to that of Lemma 3. This also implies that every process takes the same number of steps in expectation until completing an operation.

Lemma 15. *Let $\pi = [\pi_1 \dots \pi_n]$ be the stationary distribution of the global chain, and let π' be the stationary distribution of the individual chain. Let π'_i be the probability of $s_{\{p_i\}}$ in π' . Then, for all $i \in \{1, \dots, n\}$, $\pi'_i = \pi/n$. Furthermore, $W_i = nW$.*

This characterizes the asymptotic behavior of the individual latency.

Corollary 3. *For any $i \in \{1, \dots, n\}$, the expected number of system steps between two completed operations by process p_i is $O(n\sqrt{n})$. The expected number of steps by p_i between two completed operations is $O(\sqrt{n})$.*

E.3 Implementation Results

Let the *completion rate* of the algorithm be the total number of successful operations versus the total number of steps taken during the execution. The completion rate approximates the inverse of the system latency. We consider a fetch-and-increment counter implementation which simply reads the value v of a shared register R , and then attempts to increment the value using a $CAS(R, v, v + 1)$ call. The predicted completion rate of the algorithm is $\Theta(1/\sqrt{n})$. The actual completion rate of the implementation is shown in Figure 5 for varying thread counts, for a counter implementation based on the lock-free pattern. The $\Theta(1/\sqrt{n})$ rate predicted by the uniform stochastic scheduler model appears to be close to the actual completion rate. Since we do not have precise bounds on the constant in front of $\Theta(1/\sqrt{n})$ for the prediction, we scaled the prediction to the first data point. The worst-case predicted rate ($1/n$) is also shown.

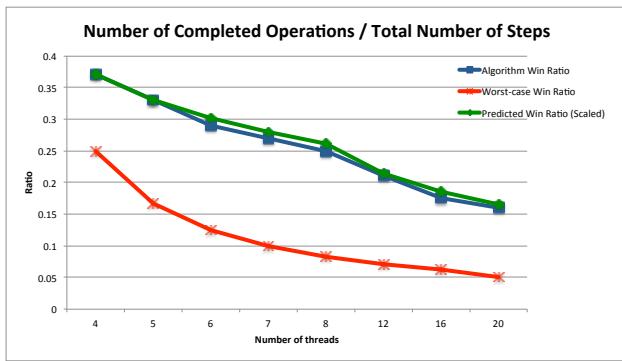


Figure 5: Predicted completion rate of the algorithm vs. completion rate of the implementation vs. worst-case completion rate.

Non-scalable locks are dangerous

Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich
MIT CSAIL

Abstract

Several operating systems rely on non-scalable spin locks for serialization. For example, the Linux kernel uses ticket spin locks, even though scalable locks have better theoretical properties. Using Linux on a 48-core machine, this paper shows that non-scalable locks can cause dramatic collapse in the performance of real workloads, even for very short critical sections. The nature and sudden onset of collapse are explained with a new Markov-based performance model. Replacing the offending non-scalable spin locks with scalable spin locks avoids the collapse and requires modest changes to source code.

1 Introduction

It is well known that non-scalable locks, such as simple spin locks, have poor performance when highly contended [1, 7, 9]. It is also the case that many systems nevertheless use non-scalable locks. However, we have run into multiple situations in which system throughput collapses suddenly due to non-scalable locks: for example, a system that performs well with 25 cores completely collapses with 30. Equally surprising, the offending critical sections are often tiny. This paper argues that non-scalable locks are dangerous. For concreteness, it focuses on locks in the Linux kernel.

One piece of the argument is that non-scalable locks can seriously degrade overall performance, and that the situations in which they may do so are likely to occur in real systems. We exhibit a number of situations in which the performance of plausible activities collapses dramatically with more than a few cores' worth of concurrency; the cause is a rapid growth in locking cost as the number of contending cores grows.

Another piece of the argument is that the onset of performance collapse can be sudden as cores are added. A system may have good measured performance with N cores, but far lower total performance with just a few

more cores. The paper presents a predictive model of non-scalable lock performance that explains this phenomenon.

A third element of the argument is that critical sections which appear to the eye to be very short, perhaps only a few instructions, can nevertheless trigger performance collapses. The paper's model explains this phenomenon as well.

Naturally we argue that one should use scalable locks [1, 7, 9], particularly in operating system kernels where the workloads and levels of contention are hard to control. As a demonstration, we replaced Linux's spin locks with scalable MCS locks [9] and re-ran the software that caused performance collapse. For 3 of the 4 benchmarks the changes in the kernel were simple. For the 4th case the changes were more involved because the directory cache uses a complicated locking plan and the directory cache in general is complicated. The MCS lock improves scalability dramatically, because it avoids the performance collapse, as expected. We experimented with other scalable locks, including hierarchical ones [8], and observe that the improvements are negligible or small—the big win is going from non-scalable locks to scalable locks.

An objection to this approach is that non-scalable behavior should be fixed by modifying software to eliminate serialization bottlenecks, and that scalable locks merely defer the need for such modification. That observation is correct. However, in practice it is not possible to eliminate all potential points of contention in the kernel all at once. Even if a kernel is very scalable at some point in time, the same kernel is likely to have scaling bottlenecks on subsequent generations of hardware. One way to view scalable locks is as a way to relax the time-criticality of applying more fundamental scaling improvements to the kernel.

The main contribution of this paper is amplifying the conclusion from previous work that non-scalable locks have risks: not only do they have poor performance, but

they can cause collapse of overall system performance. More specifically, this paper makes three contributions. First, we demonstrate that the poor performance of non-scalable locks can cause performance collapse for real workloads, even if the spin lock is protecting a very short critical section in the kernel. Second, we propose a single comprehensive model for the behavior of non-scalable spin locks that fully captures all regimes of operation, unlike previous models [6]. Third, we confirm on modern x86-based multicore processors that MCS locks can improve maximum scalability without decreasing performance, and conclude that the scaling and performance benefits of the different types of scalable locks is small.

The rest of the paper is organized as follows. Section 2 demonstrates that non-scalable locks can cause performance collapse for real workloads. Section 3 introduces a Markov-based model that explains why non-scalable locks can cause this collapse to happen, even for short critical sections. Section 4 evaluates several scalable locks on modern x86-based multicore processors to decide which scalable lock to use to replace the offending non-scalable locks. Section 5 reports on the results of using MCS locks to replace the non-scalable locks in Linux that caused performance collapse. Section 6 relates our findings and modeling to previous work. Section 7 summarizes our conclusions.

2 Are non-scalable locks a problem?

This section demonstrates that non-scalable spin locks cause performance collapse for some kernel-intensive workloads. We present performance results from four benchmarks demonstrating that critical sections that consume less than 1% of the CPU cycles on one core can cause performance to collapse on a 48-core x86 machine.

2.1 How non-scalable locks work

For concreteness we discuss the ticket lock used in the Linux kernel, but any type of non-scalable lock will exhibit the problems shown in this section. Figure 1 presents simplified C code from Linux. The ticket lock is the default lock since kernel version 2.6.25 (released in April 2008).

An acquiring core obtains a ticket and spins until its turn is up. The lock has two fields: the number of the ticket that is holding the lock (`current_ticket`) and

```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}

void spin_lock(spinlock_t *lock)
{
    int t =
        atomic_fetch_and_inc(&lock->next_ticket);
    while (t != lock->current_ticket)
        ; /* spin */
}

void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

Figure 1: Pseudocode for ticket locks in Linux.

the number of the next unused ticket (`next_ticket`). To obtain a ticket number, a core uses an atomic increment instruction on `next_ticket`. The core then spins until its ticket number is current. To release the lock, a core increments `current_ticket`, which causes the lock to be handed to the core that is waiting for the next ticket number.

If many cores are waiting for a lock, they will all have the lock variables cached. An unlock will invalidate those cache entries. All of the cores will then read the cache line. In most architectures, the reads are serialized (either by a shared bus or at the cache line’s home or directory node), and thus completing them all takes time proportional to the number of cores. The core that is next in line for the lock can expect to receive its copy of the cache line midway through this process. Thus the cost of each lock handoff increases in proportion to the number of waiting cores. Each inter-core operation takes on the order of a hundred cycles, so a single release can take many thousands of cycles if dozens of cores are waiting. Simple test-and-set spin locks incur a similar $O(N)$ cost per release.

2.2 Benchmarks

We exercised spin locks in the Linux kernel with four benchmarks: FOPS, MEMPOP, PFIND, and EXIM. Two are microbenchmarks and two represent application workloads. None of the benchmarks involve disk I/O (the file-system cache is pre-warmed). We ran the benchmarks on a 48-core machine (eight 6-core 2.4 GHz AMD

Opteron chips) running Linux kernel 2.6.39 (released in May 2011).

FOPS creates a single file and starts one process on each core. Each thread repeatedly opens and closes the file.

MEMPOP creates one process per core. Each process repeatedly `mmaps` 64 kB of memory with the `MAP_POPULATE` flag, then `munmaps` the memory. `MAP_POPULATE` instructs the kernel to allocate pages and populate the process page table immediately, instead of doing so on demand when the process accesses the page.

PFIND searches for a file by executing several instances of the GNU find utility. PFIND takes a directory and filename as input, evenly divides the directories in the first level of input directory into per-core inputs, and executes one instance of find per core, passing in the input directories. Before we execute the PFIND, we create a balanced directory tree so that each instance of find searches the same number of directories.

EXIM is a mail server. A single master process listens for incoming SMTP connections via TCP and forks a new process for each connection, which accepts the incoming message. We use the version of EXIM from MOSBENCH [3].

2.3 Results

Figure 2 shows the results for all benchmarks. One might expect total throughput to rise in proportion to the number of cores for a while, then level off to a flat line due to some serial section. Throughput does increase with more cores for a while, but instead of leveling off, the throughput *decreases* after some number of cores. The decreases are sudden; good performance with N cores is often followed by dramatically lower performance with one or two more cores.

FOPS. Figure 2(a) shows the total throughput of FOPS as a function of the number of cores concurrently running the benchmark. The performance peaks with two cores. With 48 cores, the total throughput is about 3% of the throughput on one core.

The performance collapse in Figure 2(a) is caused by a per-entry lock in the file system name/inode cache. The kernel acquires the lock when a file is closed in order to decrement a reference count and possibly perform cleanup actions. On average, the code protected by the lock executes in only 92 cycles.

MEMPOP. Figure 2(b) shows the throughput of MEMPOP. Throughput peaks at nine cores, at which point it is $4.7\times$ higher than with one core. Throughput decreases rapidly at first with more than nine cores, then more gradually. At 48 cores throughput is about 35% of the throughput achieved on one core. The performance collapse in Figure 2(b) is caused by a non-scalable lock that protects the data structure mapping physical pages to virtual memory regions.

PFIND. Figure 2(c) shows the throughput of PFIND, measured as the number of find processes that complete every second. The throughput peaks with 14 cores, then declines rapidly. The throughput with 48 cores is approximately equal to the throughput on one core. A non-scalable lock protecting the block buffer cache causes PFIND’s performance collapse.

EXIM. Figure 2(d) shows EXIM’s performance as a function of the number of cores. The performance collapse is caused by locks that protect the data structure mapping physical pages to virtual memory regions. The 3.0.0 kernel (released in Aug 2011) fixes this collapse by acquiring the locks involved in the bottlenecked operation together, and then running with a larger critical section.

Figure 3 shows measurements related to the most contended lock for each benchmark, taken on one core. The “Operation time” column indicates the total number of cycles required to complete one benchmark operation (opening a file, delivering a message, etc). The “Acquires per operation” column shows how many times the most contended lock was acquired per operation. The “Average critical section time” column shows how long the lock was held each time it was acquired. The “% of operation in critical section” reflects the ratio of total time per operation spent in the critical section to the total time for each operation.

The last column of Figure 3 helps explain the point in each graph at which collapse starts. For example, MEMPOP spends 7% of its time in the bottleneck critical section. Once 14 (i.e., $1.0/0.07$) cores are active, one would expect that critical section’s lock to be held by some core at all times, and thus that cores would start to contend for the lock. In fact MEMPOP starts to collapse somewhat before that point, a phenomenon explained in the next section.

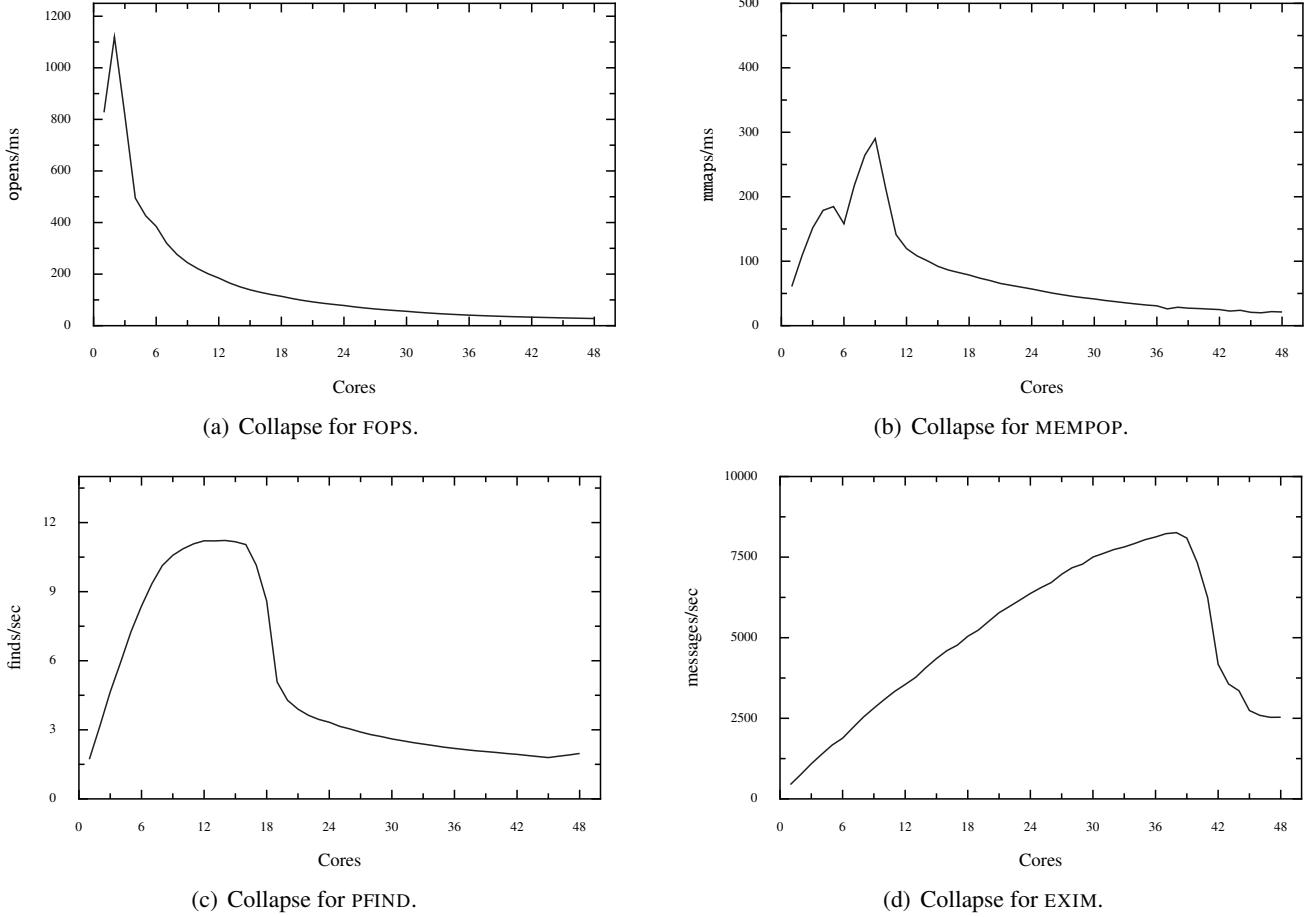


Figure 2: Sudden performance collapse with ticket locks.

Benchmark	Operation time (cycles)	Top lock instance name	Acquires per operation	Average critical section time (cycles)	% of operation in critical section
FOPS	503	d_entry	4	92	73%
MEMPOP	6852	anon_vma	4	121	7%
PFIND	2099 M	address_space	70 K	350	7%
EXIM	1156 K	anon_vma	58	165	0.8%

Figure 3: The most contended critical sections for each Linux microbenchmark, on a single core.

As an example of a critical section that causes non-scalability, Figure 4 shows the most contended critical sections for EXIM. They involve adding and deleting an element from a list, and consist of a handful of inlined instructions.

2.4 Questions

The four graphs have common features which raise some questions.

- Why does collapse start as early as it does? One would expect collapse to start when there is a significant chance that many cores need the same lock at the same time. Thus one might expect MEMPOP to start to see decline at around 14 cores ($1.0/0.07$). But the onset occurs earlier, at nine cores.
- Why does performance ultimately fall so far?
- Why does performance collapse so rapidly? One might expect a gradual decrease with added cores, since each new core should cause each release of the bottleneck lock to take a little more time. Instead, adding just a few more cores causes a sharp drop in total throughput. This is worrisome; it suggests that a system that has been tested to perform well with N cores might perform far worse with $N + 2$ cores.

3 Model

This section presents a performance model for non-scalable locks that answers the questions raised in the previous section. It first describes the hardware cache coherence protocol at a high level, which is representative of a typical *x86* system, and then builds on the basic properties of this protocol to construct a model for understanding performance of ticket spin locks. The model closely predicts the observed collapse behavior.

3.1 Hardware cache coherence

Our model assumes a directory-based cache coherence protocol. All directories are directly connected by an inter-directory network. The cache coherence protocol is a simplification of, but similar to, the implementation in AMD Opteron [4] and Intel Xeon CPUs.

```

static void
anon_vma_chain_link(
    struct anon_vma_chain *avc,
    struct anon_vma *anon_vma)
{
    spin_lock(&anon_vma->lock);
    list_add_tail(&avc->same_anon_vma,
                  &anon_vma->head);
    spin_unlock(&anon_vma->lock);
}

static void
anon_vma_unlink(
    struct anon_vma_chain *avc,
    struct anon_vma *anon_vma)
{
    spin_lock(&anon_vma->lock);
    list_del(&avc->same_anon_vma);
    spin_unlock(&anon_vma->lock);
}

```

Figure 4: The most contended critical sections from EXIM. This compiler inlines the code for the list manipulations, each of which are less than 10 instructions.

3.1.1 The directory

Each core has a cache directory. The hardware (e.g., the BIOS) assigns evenly sized regions of DRAM to each directory. Each directory maintains an entry for each cache line in its local DRAM:

[tag | state | core ID]

The possible states are:

1. invalid (I) – the cache line is not cached;
2. shared (S) – the cache line is held in one or more caches and matches DRAM;
3. modified (M) – the cache line is held in one cache and does not match DRAM.

For modified cache lines the directory records the cache that holds the dirty cache line.

Figure 5 presents the directory state transitions for loads and stores. For example, when a core issues a load request for a cache line in the invalid state, the directory sets the cache line state to shared.

	I	S	M
Load	S	S	S
Store	M	M	M

Figure 5: Directory transitions for loads and stores.

	I	S	M
Load	—	—	DP
Store	—	BI	DI

Figure 6: Probe messages for loads and stores.

3.1.2 Network messages

When a core begins accessing an uncached cache line, it will send a load or store request to the cache line’s home directory. Depending on the type of request and the state of the cache line in the home directory, the home directory may need to send probe messages to all directories that hold the cache line.

Figure 6 shows the probe messages a directory sends based on request type and state of the cache line. “BI” stands for broadcast invalidate. “DP” stands for direct probe. “DI” stands for direct invalidate. For example, when a source cache issues a load request for a modified cache line, the home directory sends a directed probe to the cache holding the modified cache line. That cache responds to the source cache with the contents of the modified cache line.

3.2 Performance model for ticket locks

To understand the collapse observed in ticket-based spin locks, we construct a model. One of the challenging aspects of constructing an accurate model of spin lock behavior is that there are two regimes of operation: when not contended, the spin lock can be acquired quickly, but when many cores try to acquire the lock at the same time, the time taken to transfer lock ownership increases significantly. Moreover, the exact point at which the behavior of the lock changes is dependent on the lock usage pattern, and the length of the critical section, among other parameters. Recent work [6] attempts to model this behavior by combining two models—one for contention and one for uncontended locks—into a single model, by simply taking the max of the two models’ predictions. However, this fails to precisely model the point of collapse, and does not explain the phenomenon causing the collapse.

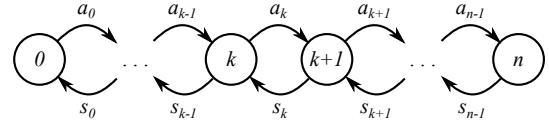


Figure 7: Markov model for a ticket spin lock for n cores. State i represents i cores holding or waiting for the lock. a_i is the arrival rate of new cores when there are already i cores contending for the lock. s_i is the service rate when $i + 1$ cores are contending.

To build a precise model of ticket lock behavior, we build on queueing theory to model the ticket lock as a Markov chain. Different states in the chain represent different numbers of cores queued up waiting for a lock, as shown in Figure 7. There are $n + 1$ states in our model, representing the fact that our system has a fixed number of cores (n).

Arrival and service rates between different states represent lock acquisition and lock release. These rates are different for each pair of states, modeling the non-scalable performance of the ticket lock, as well as the fact that our system is closed (only a finite number of cores exist). In particular, the arrival rate from k to $k + 1$ waiters, a_k , should be proportional to the number of remaining cores that are not already waiting for the lock (i.e., $n - k$). Conversely, the service rate from $k + 1$ to k , s_k , should be inversely proportional to k , reflecting the fact that transferring ownership of a ticket lock to the next core takes linear time in the number of waiters.

To compute the arrival rate, we define a to be the average time between consecutive lock acquisitions on a single core. The rate at which a single core will try to acquire the lock, in the absence of contention, is $1/a$. Thus, if k cores are already waiting for the lock, the arrival rate of new contenders is $a_k = (n - k)/a$, since we need not consider any cores that are already waiting for the lock.

To compute the service rate, we define two more parameters: s , the time spent in the serial section, and c , the time taken by the home directory to respond to a cache line request. In the cache coherence protocol, the home directory of a cache line responds to each cache line request in turn. Thus, if there are k requests from different cores to fetch the lock’s cache line, the time until the winner (pre-determined by ticket numbers) receives the cache line will be on average $c \cdot k/2$. As a result, processing the serial section and transferring the lock to the next

holder when k cores are contending takes $s + ck/2$, and the service rate is $s_k = \frac{1}{s+ck/2}$.

Unfortunately, while this Markov model accurately represents the behavior of a ticket lock, it does not match any of the standard queueing theory that provides a simple formula for the behavior of the queueing model. In particular, the system is closed (unlike most open-system queueing models), and the service times vary with the size of the queue.

To compute a formula, we derive it from first principles. Let P_0, \dots, P_n be the steady-state probabilities of the lock being in states 0 through n respectively. Steady state means that the transition rates balance: $P_k \cdot a_k = P_{k+1} \cdot s_k$. From this, we derive that $P_k = P_0 \cdot \frac{n!}{a^k(n-k)!} \cdot \prod_{i=1}^k (s + ic)$. Since $\sum_{i=0}^n P_i = 1$, we get $P_0 = 1 / \left(\sum_{i=0}^n \left(\frac{n!}{a^i(n-i)!} \prod_{j=1}^i (s + jc) \right) \right)$, and thus:

$$P_k = \frac{\frac{1}{a^k(n-k)!} \cdot \prod_{i=1}^k (s + ic)}{\sum_{i=0}^n \left(\frac{1}{a^i(n-i)!} \prod_{j=1}^i (s + jc) \right)} \quad (1)$$

Given the steady-state probability for each number of cores contending for the lock, we can compute the average number of waiting (idle) cores as the expected value of that distribution, $w = \sum_{i=0}^n i \cdot P_i$. The speedup achieved in the presence of this lock and serial section can be computed as $n - w$, since on average that many cores are doing useful work, while w cores are spinning.

3.3 Validating the model

To validate our model, Figures 8 and 9 show the predicted and actual speedup of a microbenchmark with a single lock, which spends a fixed number of cycles inside of a serial section protected by the lock, and a fixed number of cycles outside of that serial section. Figure 8 shows the predicted and actual speedup when the serial section always takes 400 cycles to execute, but the non-serial section varies from 12.5k to 200k cycles. As we can see, the model closely matches the real hardware speedup for all configurations.

In Figure 9, we also present the predicted and actual speedup of the microbenchmark when the serial section is always 2% of the overall execution time (on one core),

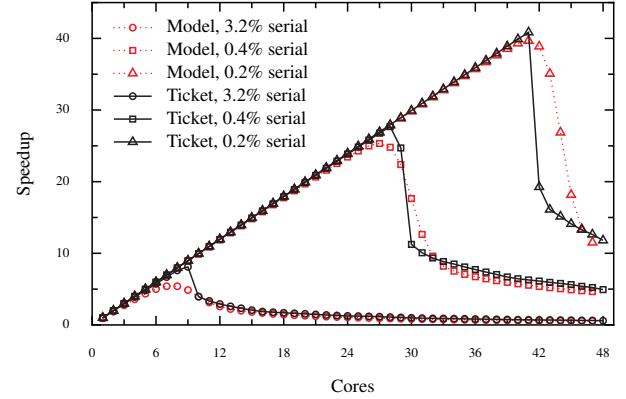


Figure 8: Predicted and actual performance of ticket spin locks with a 400-cycle serial section, for a microbenchmark where the serial section accounts for a range of fractions of the overall execution time on one core.

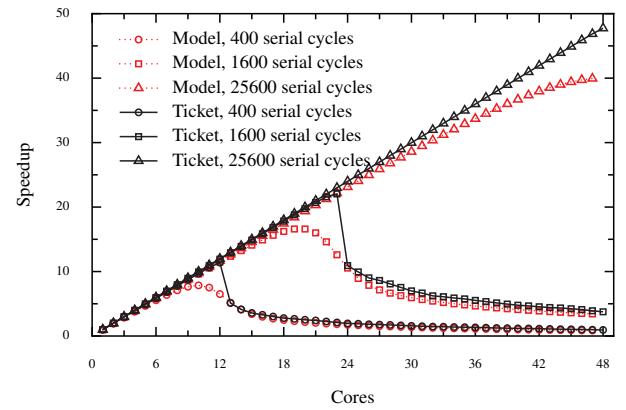


Figure 9: Predicted and actual performance for a microbenchmark where the critical section accounts for 2% of the execution time on one core, but with varying execution time for each invocation of the serial section.

but the time spent in the serial section varies from 400 to 25,600 cycles. Again, the model closely matches the measured speedup. This gives us confidence that our model accurately captures the relevant factors leading to the performance collapse of ticket locks.

One difference between the predicted and measured speedup is that the predicted collapse is slightly more gradual than the collapse observed on real hardware. This is because the ticket lock’s performance is unstable near the collapse point, and the model predicts the average steady-state behavior. Our measured speedup reports the throughput for a relatively short-running microbenchmark, which has not had the time to “catch” the instability.

3.4 Implications of model results

The behavior predicted by our model has several important implications. First, the rapid collapse of ticket locks is an inherent property of their design, rather than a performance problem with our experimental hardware. Any cache-coherent system that matches our basic hardware model will experience similarly sharp performance degradation. The reason behind the rapid collapse can be understood by considering the transition rates in the Markov model from Figure 7. If a lock ever accumulates a large number of waiters (e.g., reaches state n in the Markov model), it will take a long time for the lock to go back down to a small number of waiters, because the service rate s_k rapidly decays as k grows, for short serial sections. Thus, once the lock enters a contended state, it becomes much more likely that more waiters arrive than that the current waiters will make progress in shrinking the lock’s wait queue.

A more direct way to understand the collapse is that the time taken to transfer the lock from one core to another increases linearly with the number of contending cores. However, this time effectively increases the length of the serial section. Thus, as more cores are contending for the lock, the serial section grows, increasing the probability that yet another core will start contending for this lock.

The second implication is that the collapse of the ticket lock only occurs for short serial sections, as can be seen from Figure 9. This can be understood by considering how the service rate s_i decays for different lengths of the serial section. For a short serial section time s , $s_k = \frac{1}{s+ck/2}$ is strongly influenced by k , but for large s , s_k is largely unaffected by k . Another way to understand this result is that, with fewer acquire and release operations, the ticket lock’s performance contributes less to the overall application throughput.

The third implication is that the collapse of the ticket lock prevents the application from reaching the maximum performance predicted by Amdahl’s law (for short serial sections). In particular, Figure 9 shows that a microbenchmark with a 2% serial section, which may be able to scale to 50 cores under Amdahl’s law, is able to attain less than 10 \times scalability when the serial section is 400 cycles long.

4 Which scalable lock?

The literature has many proposals for scalable locks, which avoid the collapse that ticket locks exhibit. Which one should we use to replace contended ticket locks? This section evaluates several scalable locks on modern x86-based multicore processors.

4.1 Scalable locks

A common way of making the ticket lock more scalable is to adjust its implementation to use proportional back-off when the lock is contended. The challenge with this approach is what constant to choose to multiply the ticket number with. From our model we can conclude that choosing the constant well is important only for short critical section, because for large critical sections collapse does not occur. For our experiments below, we choose the best value by trying a range of values, and selecting the one that gives the best performance. This choice provides the best result that the proportional lock could achieve.

Another approach is to replace the ticket lock with a truly scalable lock. A scalable lock is one that generates a constant number of cache misses per acquisition and therefore avoids the collapse that non-scalable locks exhibit. All of these locks maintain a queue of waiters and each waiter spins on its own queue entry. The differences in these locks are how the queue is maintained and the changes necessary to the lock and unlock API.

MCS lock. The MCS lock [9] maintains an explicit queue of qnode structures. A core acquiring the lock adds itself with an atomic instruction to the end of the list of waiters by having the lock point to its qnode, and then sets the next pointer of the qnode of its predecessor to point to its qnode. If the core is not at the head of the queue, then it spins on its qnode. To avoid dynamically allocating memory on each lock invocation, the qnode is an argument to lock and unlock.

K42 lock. A potential downside of the MCS lock is that it involves an API change. The K42 lock [2] is a variant of the MCS lock that requires fewer API changes.

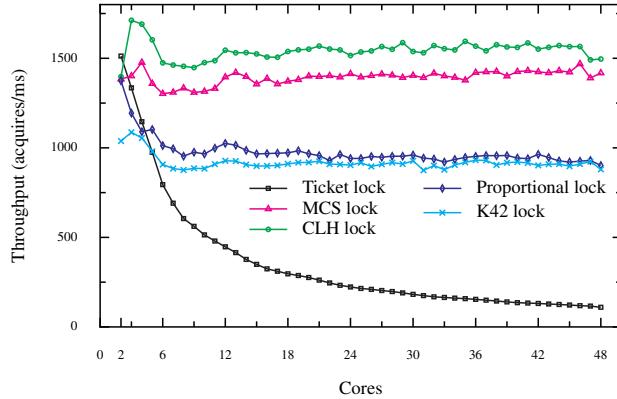


Figure 10: Throughput for cores acquiring and releasing a shared lock. Results start with two cores.

CLH lock. The CLH lock [5] is a variant of an MCS lock where the waiter spins on its predecessor `qnode`, which allows the queue of waiters to be implicit (i.e., the `qnode` next pointer and its updates are unnecessary).

HCLH lock. The HCLH lock [8] is a hierarchical variant of the CLH lock, intended for NUMA machines. The way we use it is to favor lock acquisitions from cores that share an L3 cache with the core that currently holds the lock, with the goal to reduce the cost of cache line transfers between remote cores.

4.2 Results

Figure 10 shows the performance of the ticket lock, proportional lock, MCS lock, K42 lock, and CLH lock on our 48-core AMD machine. The benchmark uses one shared lock. Each core loops, acquires the shared lock, updates 4 shared cache lines, and releases the lock. The time to update the 4 shared cache lines is similar between runs using different locks, and increases gradually from about 800 cycles on 2 cores to 1000 cycles in 48 cores. On our *x86* multicore machine, the HCLH lock improves performance of the CLH lock by only 2%, and is not shown.

All scalable locks scale better than ticket lock on this benchmark because they avoid collapse. Using the CLH lock results in slightly higher throughput over the MCS lock, but not by much. The K42 lock achieves lower

Lock type	Single acquire	Single release	Shared acquire
MCS lock	25.6	27.4	53
CLH lock	28.8	3.9	517
Ticket lock	21.1	2.4	30
Proportional lock	22.0	2.8	30.2
K42 lock	47.0	23.8	74.9

Figure 11: Performance of acquiring and releasing an MCS lock, a CLH lock, and a ticket lock. Single acquire and release are measurements for one core. Shared acquire is the time for a core to acquire a lock recently released by another core. Numbers are in cycles.

throughput than the MCS lock because it incurs an additional cache miss on acquire. These results indicate that for our *x86* multicore machine, it does not matter much which scalable lock to choose.

We also ran the benchmarks on a multicore machine with Intel CPUs and measured performance trends similar to those shown in Figure 10.

Another concern about different locks is the cost of `lock` and `unlock`. Figure 11 shows the cost for each lock in the uncontended and contended case. All locks are relatively inexpensive to acquire on a single core with no sharing. MCS lock and K42 lock are more expensive to release on a single core, because, unlike the other locks, they use atomic instructions to release the lock. Acquiring a shared but uncontended lock is under 100 cycles for all locks, except the CLH lock. Acquiring the CLH lock is expensive due to the overhead introduced by the `qnode` recycling scheme for multiple cores.

5 Using MCS locks in Linux

Based on the result of the previous section, we replaced the offending ticket locks with MCS locks. We first describe the kernel changes to use MCS locks, and then measure the resulting scalability for the 4 benchmarks from Section 2.

5.1 Using MCS Locks

We replaced the three ticket spin locks that limited benchmark performance with MCS locks. We modified about 1,000 lines of the Linux kernel (700 lines for `d_entry`, 150 lines for `anon_vma`, and 150 lines for `address_space`).

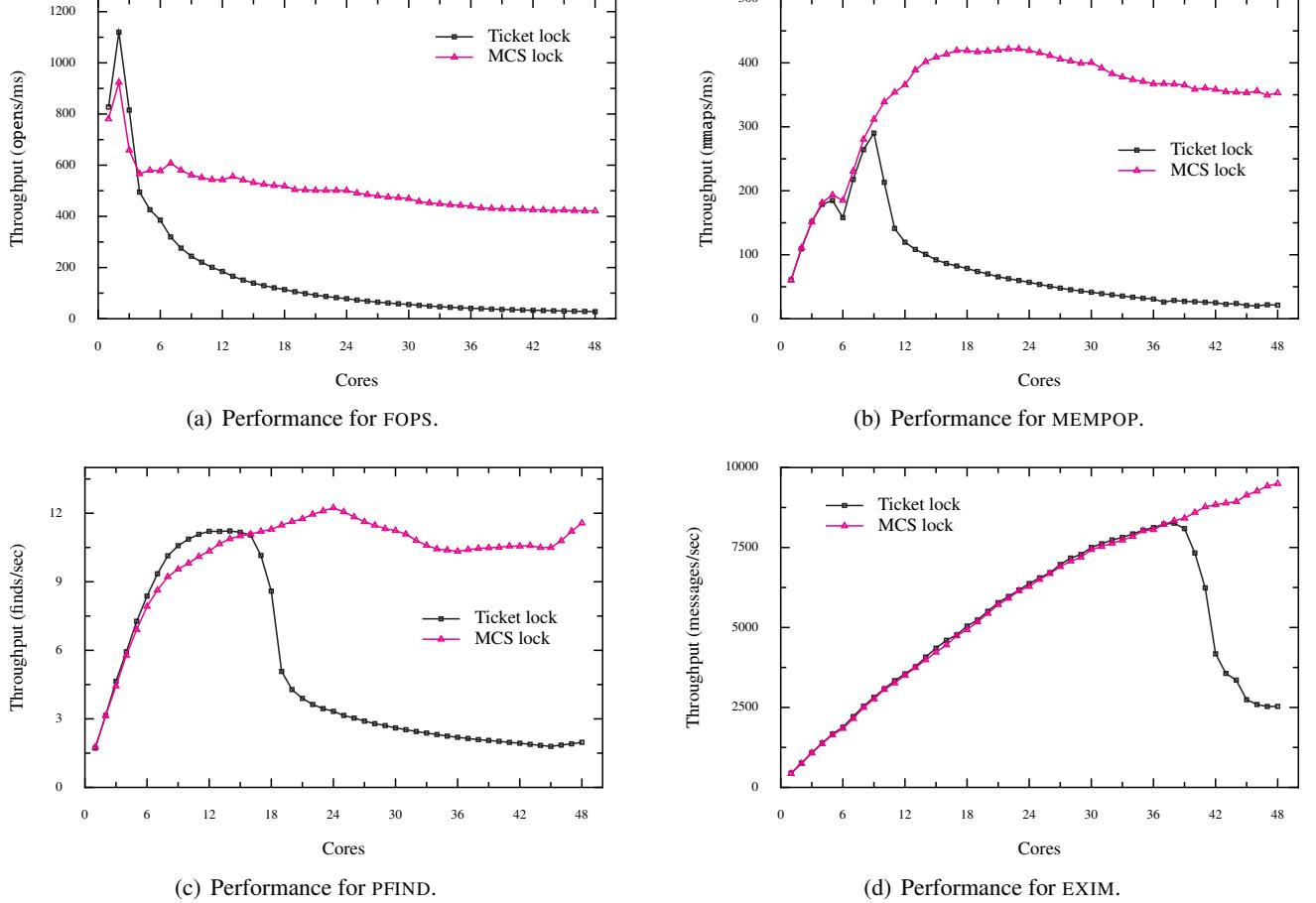


Figure 12: Performance of benchmarks using ticket locks and MCS locks.

As noted earlier, MCS locks have a different API than the Linux ticket spin lock implementation. When acquiring an MCS lock, a core must pass a `qnode` variable into `mcs_lock`, and when releasing that lock the core must pass the same `qnode` variable to `mcs_unlock`. For each lock a core holds, the core must use a unique `qnode`, but it is acceptable to use the same `qnode` for locks held at different times.

Many of our kernel modifications are straightforward. We allocate an MCS `qnode` on the stack, replace `spin_lock` and `spin_unlock` with `mcs_lock` and `mcs_unlock`, and pass the `qnode` to the MCS acquire and release functions.

In some cases, the Linux kernel acquires a lock in one function and releases it in another. For this situation, we stack-allocate a `qnode` on the call frame that is an ancestor of both the call frame that calls `mcs_lock` and the one that calls `mcs_release`. This pattern is common

in the directory cache, and partially explains why we made so many modifications for the `d_entry` lock.

Another pattern, which we encountered only in the directory cache code that implements moving directory entries, is changing the value of lock variables. When the kernel moves a `d_entry` between two directories, it acquires the lock of the `d_entry->d_parent` (which is also a `d_entry`) and the target directory `d_entry`, and then sets the value `d_entry->d_parent` to be the target `d_entry`. With MCS, we must make sure to unlock `d_entry->d_parent` with the `qnode` originally used to lock the target `d_entry`, instead the `qnode` originally used to lock `d_entry->d_parent`.

5.2 Results

The graphs in Figure 12 show the benchmark results from replacing contended ticket locks with MCS locks. For a

large number of cores, using MCS improves performance by at least $3.5\times$ and in one case by more than $16\times$.

Figure 12(a) shows the performance of FOPS with MCS locks. Going from one to two cores, performance with both ticket locks and MCS locks increases. For more than two cores, performance with the ticket spin lock decreases continuously. Performance with MCS locks initially also decreases from two to four cores, then remains relatively stable. The reason for this decrease in performance is that the time spent executing the critical section increases from 450 cycles on two cores to 852 cycles on four cores. The critical section is executed multiple times per-operation and modifies shared data, which incurs costly cache misses. As more cores are added, it is less likely that a core will have been the last core to execute the critical section, and therefore it will incur more cache misses, which will increase the length of the critical section.

Figure 12(b) shows the performance of MEMPOP with MCS locks. Performance with MCS and ticket locks increases up to 9 cores, at which point the performance with ticket locks collapses and continuously decreases as more cores are used. The length of the critical section is relatively short. It increases from 141 cycles on one core to about 350 cycles on 10 cores. MCS avoids the dramatic collapse from the short critical section and increases maximum performance by $16.6\times$.

Figure 12(c) shows the performance of PFIND with MCS. The `address_space` ticket spin lock performs well up to 15 cores, then causes a performance drop that continues until 48 cores. The serial section updates some shared data which increases the time spent in the critical section from 350 cycles on one core to about 1100 cycles on more than 44 cores. MCS provides a small increase in maximum performance, but not as much as with MEMPOP since the critical section is much longer.

Figure 12(d) shows the performance of EXIM with MCS. Ticket spin locks perform well up to 39 cores, then cause a performance collapse. The critical section is relatively short (165 cycles on one core), so MCS improves maximum performance by $3.7\times$ on 48 cores.

The results show that using scalable locks is not that much work (e.g., less work than using RCU in the kernel) and avoids the performance collapse that results from non-scalable spin locks. The latter benefit is important because institutions often use the same kernels for several

years, even as they upgrade to hardware with more cores and the likelihood of performance cliffs increases.

6 Related Work

The literature on scalable and non-scalable locks is vast, many practitioners are well familiar with the issues, and it is well known that non-scalable locks behave poorly under contention. The main contribution that this paper adds is the observation that non-scalable locks can cause system performance to collapse, as well as a model that nails down why the performance collapse is so dramatic, even for short critical sections.

Anderson [1] observes that the behavior of spin locks can be “degenerative”. The MCS paper shows that acquisition of a test-and-set lock increases linearly with processors on the BBN Butterfly and that on the Symmetry this cost manifests itself even with a small number of processors [9]. Many researchers and practitioners have written microbenchmarks that show that non-scalable spin locks exhibit poor performance under contention on modern hardware. This paper shows that non-scalable locks can cause the collapse of system performance under plausible workloads; that is, the locking costs for short critical sections can be very large on the scale of kernel execution time.

The Linux scaling study reports on the performance collapse that ticket locks introduce on the EXIM benchmark, but doesn’t explain the collapse [3]. This paper shows the collapse and the suddenness with several workloads, and provides a model that explains the acuteness.

Eyerman and Eeckhout [6] provide closed formulas to reason about the speedup of parallel applications that involve critical sections, pointing out that there is regime in which applications achieve better speedup than Amdahl’s law predicts. Unfortunately, their model makes a distinction between the contended and uncontended regime and proposes formula for each regime. In addition, the formulas do not model the insides of locks; instead, they assume scalable locks. This paper contributes a comprehensive model that accurately predicts performance across the whole spectrum from uncontended to contended, and applies it to modeling the inside of locks.

7 Conclusion

This paper raises another warning about non-scalable locks. Although it is well understood that non-scalable

spin locks have poor performance, it is less well appreciated that their poor performance can have dramatic effect on overall system performance. This paper shows that non-scalable locks can cause system performance to collapse under real workloads and that the collapse is sudden, and presents a Markov model that explains the sudden collapse. We conclude that using non-scalable locks is dangerous because a system that has been tested with N cores may experience performance collapse at a few more cores—or, put differently, if one upgrades a machine in hopes of achieving higher performance, one might run the risk of ending up with a performance collapse. Scalable locking is a stop-gap solution that avoids collapse, but achieving higher performance with additional cores can require new designs using lock-free data structures.

Acknowledgments

We thank Abdul Kabbani for helping us analyze our Markov model. This work was supported by Quanta Computer, by Google, and by NSF awards 0834415 and 0915164.

References

- [1] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 1:6–16, January 1990.
- [2] M. A. Auslander, D. J. Edelsohn, O. Y. Krieger, B. S. Rosenberg, and R. W. Wisniewski. Enhancement to the MCS lock for increased functionality and improved programmability. U.S. patent application 10/128,745, 2003.
- [3] S. Boyd-Wickizer, A. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of Linux scalability to many cores. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI 2010)*, Vancouver, Canada, October 2010.
- [4] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes. Cache hierarchy and memory subsystem of the AMD Opteron processor. *Micro, IEEE*, 30(2):16–29, March–April 2010.
- [5] T. S. Craig. Building FIFO and priority-queuing spin locks from atomic swap. Technical Report UW-CSE-93-02-02, University of Washington, Department of Computer Science and Engineering, 1993.
- [6] S. Eyerman and L. Eeckhout. Modeling critical sections in Amdahl’s law and its implications for multicore design. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA)*, pages 262–370, Saint-Malo, France, June 2010.
- [7] M. Herlihy and N. Shavit. *The art of multiprocessor programming*. Morgan Kaufman, 2008.
- [8] V. Luchangco, D. Nussbaum, and N. Shavit. A hierarchical CLH queue lock. In *Proceedings of the European Conference on Parallel Computing*, pages 801–810, Dresden, Germany, August–September 2006.
- [9] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

Lock-free programming is a challenge, not just because of the complexity of the task itself, but because of how difficult it can be to penetrate the subject in the first place.

--preshing.com

