Tech. Rep. 599, Computer Science Dept., Univ. of Rochester

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 ← SAFEREAD(&Freelist)
3       if p = NULL
4           error out of memory
5       if CAS(&Freelist, p, p^.next) = TRUE
6           p^.claim ← 0
7           return p
8       else
9           RELEASE(p)

RECLAIM(p)
1   repeat
2       q ← Freelist
3       p^.next ← q
4   until CAS(&Freelist, q, p) = TRUE

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

RELEASE(p)
1   if p = NULL
2       return
3   if FETCHANDADD(&p^.refct, −1) > 1
4       return
5   if TESTANDSET(&p^.claim) = 1
6       return
7   for all link fields q in p^
8       RELEASE(q)
9   RECLAIM(p)
```

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

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

Figure 2: Dequeue operation with race condition.

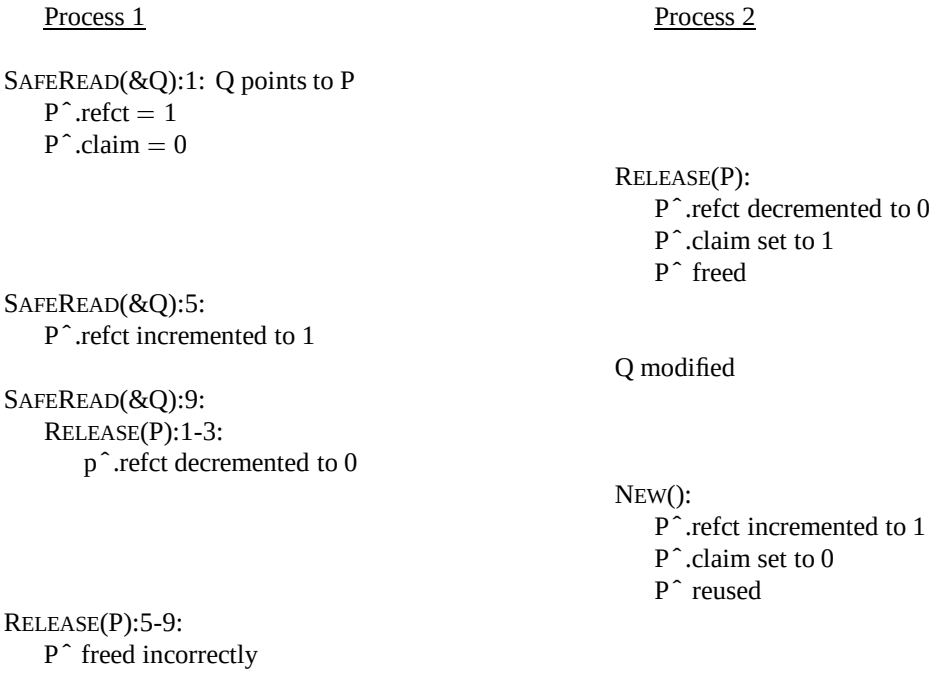

Figure 3: Race condition in RELEASE.

Process 1	Process 2
DEQUEUE:6:	
CAS succeeds Head points to P^	
P^.refct = 2	
	DEQUEUE:6:
	CAS succeeds Head points to P^.next^
	DEQUEUE:12-13:
	P^.refct decremented to 0
	P^.claim set to 1
	P^ freed incorrectly

Figure 4: Race condition in DEQUEUE.

```
DEQUEUE()
 1  repeat
 2      p ← SAFEREAD(&Head)
 3      next ← p^.next
 4      if next = NULL
 5          RELEASE(p)
 6          return empty queue
 7      ATOMICADD(&next^.refct_claim, 2)
 8      r ← CAS(&Head, p, next)
 9      if r = FALSE
10          RELEASE(next)
11      RELEASE(p)
12  until r = TRUE
13  v ← p^.next^.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.