To appear in *Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems*, Las Vegas, NV, October 1994.

Implementing Lock-Free Queues

John D. Valois
Department of Computer Science
Rensselaer Polytechnic Institute
Troy, NY 12180

Abstract

We study practical techniques for implementing the FIFO queue abstract data type using *lock-free* data structures, which synchronize the operations of concurrent processes without the use of mutual exclusion. Two new algorithms based on linked lists and arrays are presented. We also propose a new solution to the ABA problem associated with the COMPARE&SWAP instruction. The performance of our linked list algorithm is compared several other lock-free queue implementations, as well as more conventional locking techniques.

1 Introduction

Concurrent access to a data structure shared among several processes must be synchronized in order to avoid conflicting updates. Conventionally this is done using mutual exclusion; processes modify the data structure only inside a *critical section* of code, within which the process is guaranteed exclusive access to the data structure. Typically, on a multiprocessor, critical sections are guarded with a *spin-lock*. We will refer to all methods using mutual exclusion as locking or lock-based methods.

More recently, researchers have studied methods of implementing concurrent data structures which make no use of mutual exclusion. In an asynchronous environment such *lock-free* data structures can have several advantages. In particular, slow or stopped processes do not prevent other processes from accessing the data structure.

The FIFO queue is an important abstract data type, lying at the heart of operating system implementation. Queues are also useful in implementing parallel versions of many algorithms, such as quicksort and branch-and-bound, and are generally useful as a means of distributing work to a number of processes [14]. Many authors have proposed algorithms for lock-free queues in the literature [6, 8, 11, 12, 16, 18, 19, 20].

In the remainder of this paper we examine practical implementations of lock-free FIFO queues. Section 2 introduces some essential concepts related to lock-free data structures. Sections 3 surveys previous work and presents a new algorithm for lock-free queues using a linked list data structure, and Section 4 discusses how to handle the ABA problem that can occur with these types of algorithms. Section 5 surveys algorithms based on arrays, and presents our second new algorithm. Section 6 reports some preliminary experimental results comparing our algorithms to other techniques.

2 Background

Our goal is to design a *concurrent* queue that supports the normal ENQUEUE and DEQUEUE operations. In a concurrent data structure, individual processes execute single operations sequentially; however, operations by different processes may be in progress simultaneously. The differs from a *parallel* data structure, in which processes cooperate to perform one or more operations together simultaneously, and a *sequential* data structure which can only be accessed by a single process.

There are two useful properties that a lock-free data structure may have. The *non-blocking* property guarantees that at least one process executing an operation will complete within a finite number of steps, while the *wait-free* property guarantees that every process will complete its operation in a finite number of steps. A non-blocking data structure has the property, men-

tioned in the introduction, that the data structure is always accessible despite processes that may slow down or halt during an operation. A wait-free data structure further ensures that no process will starve. Note that a lock-based data structure cannot have either property, since a process inside the critical section can delay all operations indefinitely.

Other authors have used the terms *lock-free* and *non-blocking* as synonymous, but we find it useful to distinguish between algorithms that do not require mutual exclusion and those that actually provide the non-blocking property. Several of the algorithms we will discuss in this paper fall into the former category.

We use *linearizability* [7] as the correctness condition for our data structures. Linearizability implies that each operation appears to take place instantaneously at some point in time, and that the relative order of non-concurrent operations is preserved. In other words, for operations that are not concurrent, the data structure behaves exactly like its sequential counterpart. Concurrent operations can take place in any relative sequential order.

Universal constructions exist for constructing lock-free data structures from sequential functional algorithms [4, 15], or concurrent lock-based algorithms [17, 21]. In general, for a simple data structure like a queue, these methods have far more overhead than the algorithms we will be considering.

We will assume that the target architecture supports common atomic read-modify-write primitives, such as FETCH&ADD (FAA) and COMPARE&SWAP (CSW). FAA atomically reads the value in a memory location, adds another value to it, and writes the result back into the memory location, returning the original value. CSW takes three values: a memory location, an *old* value, and *new* value. If the current value of the memory location is equal to the *old* value, then the *new* value is written to the memory location. Thus, CSW atomically writes a new value into a memory location only if we know its current contents. CSW returns a condition code indicating if it is successful or not.

We use the following notation in our pseudo-code: if p is a pointer, then $p\hat{}$ represents the object pointed to, and $p\hat{}.field$ refers to a field in the object. We assume that memory allocation and reclamation are provided; memory management is discussed further in Section 4.

3 Linked List Implementations

In this section we review several proposed algorithms for lock-free queues that are based on a linked list data structure, and we propose a new algorithm. The data structure in all of these algorithms is composed of records, each containing two fields: *next*, a pointer to the next record in the list, and *value*, the data value stored in the record. Two global pointers, *head* and *tail*, point to records on the list; these pointers are used to quickly find the correct record when dequeuing and enqueuing, respectively.

All of these algorithms, with the exception of the algorithm of Massalin and Pu described in section 3.3, can be implemented using the CSW atomic primitive.

3.1 Non-Linearizable Methods

The use of CSW to implement queues shared by multiple processors is mentioned in [18], where its original use is attributed to the early 1970s. A brief mention of this method is also in [8].

The method works as follows: For a DEQUEUE operation, CSW is used to advance the *head* pointer forward one node in the linked list; the node originally pointed at is now dequeued. For an ENQUEUE operation, CSW is used to make the *tail* pointer point at the new node being enqueued; the new node is then linked onto the end of the list.

The author omits a discussion of how to handle empty queues; this is not a trivial task, since with an empty queue, concurrent ENQUEUE and DEQUEUE operations can conflict. These algorithms can also result in non-linearizable behavior.

In particular, it is possible for a process performing a DEQUEUE operation to think that the queue is empty when it is not if an enqueuing process is slow in linking the new node onto the end of the list. Furthermore, if the process halts completely, the list structure is broken and cannot be repaired, since the only halted process has any knowledge of what link needs to be made.

3.2 Blocking Methods

Mellor-Crummey [12] and Stone [19] present versions of the above basic algorithm which fix these flaws. In order to ensure that the queue is linearizable, however, these two algorithms detect when a slow enqueuing process has not yet linked its node to the list, and simply wait. Thus, while they do not employ any mutual exclusion, neither of these two algorithms has the non-blocking property.

3.3 Non-blocking Methods

Prakash *et al.* [16] present a queue that is both linearizable and non-blocking. Their approach is to take

a *snapshot* of the current state of the queue; by using this information, a process is able to complete the operation of any stalled process that may be blocking it.

In order to accomplish this, during an ENQUEUE operation, this algorithm first uses CSW to link the new node onto the list, and then uses a second CSW to update *tail*. (This second CSW is not retried if it fails.) This keeps all of the information necessary for another process to complete the ENQUEUE operation (by updating *tail*) globally accessible.

A disadvantage of this algorithm is that because of the need to take the snapshot of the queue, enqueuing and dequeuing processes, which would not normally interfere with each other, can experience contention.

Massalin and Pu have developed lock-free queue algorithms as part of a lock-free multiprocessor operating system [11]. Their algorithms rely on a powerful variant of COMPARE&SWAP that allows two arbitrary words to be modified atomically, found on the Motorola 65030 processor; we do not consider their algorithms in this paper.

3.4 A New Lock-Free Queue

We now describe a new lock-free queue algorithm. Pseudo-code for this algorithm appears in Figure 1.

```
ENQUEUE(x)
  q ← new record
  q^.value ← x
  q^.next ← NULL
  repeat
      p ← tail
      succ ← COMPARE&SWAP(p^.next, NULL, q)
      if succ ≠ TRUE
          COMPARE&SWAP(tail, p, p^.next)
  until succ = TRUE
  COMPARE&SWAP(tail, p, q)
end

DEQUEUE()
  repeat
      p ← head
      if p^.next = NULL
          error queue empty
  until COMPARE&SWAP(head, p, p^.next)
  return p^.next^.value
end
```

Figure 1: ENQUEUE and DEQUEUE operations.

Like the algorithm of Prakash *et al.*, for ENQUEUE operations our algorithm first links the new node to the end of the list, and then updates the *tail* pointer. Our DEQUEUE operation is slightly different, however. Rather than having *head* point to the node currently at the front of the queue, it points at the last node that was dequeued. (Thus, the node at the head of the queue is the node immediately following the one pointed at by *head*.)

This dummy node at the front of the list ensures that both *head* and *tail* always point at a node on the linked list, thus avoiding problems that occur when the queue is empty or contains only a single item. This technique also eliminates contention between enqueuing and dequeuing processes even when there is only a single item in the queue.

We no longer need the snapshot of Prakash *et al.*'s algorithm, since the only intermediate state that the queue can be in is if the *tail* pointer has not been updated. A process performing an ENQUEUE operation will discover this when its first CSW returns unsuccessfully, and it can then attempt to update *tail* itself.

Figure 2 shows a queue implemented as described in this section. Notice that a process is in the midst of enqueuing item C, and that the *tail* pointer has not yet been updated.

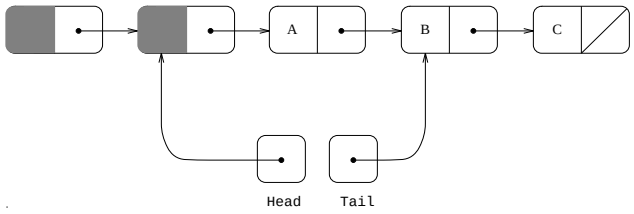


Figure 2: Queue in a linked list.

There a several strategies we can use when retrying operations. In the code above, we employ a strict policy regarding the positioning of the *tail* pointer; it always points to the last node on the list or the one immediately preceding it. This is accomplished by the second CSW instruction, which attempts to update the *tail* pointer if the process fails to enqueue its own node.

An alternative policy is to treat the *tail* as only a "hint" to the location of the last node on the list, pointing to a node that is fairly close to but possibly not exactly at the end of the list. Figure 3 gives pseudo-code implementing this policy.

The following theorem shows that the maximum distance from the end of the list that the *tail* pointer can stray is limited by the number of concurrent ENQUEUE operations.

```
ENQUEUE(x)
  q ← new record
  q^.value ← x
  q^.next ← NULL
  p ← tail
  oldp ← p
  repeat
      while p^.next ≠ NULL
          p ← p^.next
  until COMPARE&SWAP(p^.next, NULL, q)
  COMPARE&SWAP(tail, oldp, q)
end
```

Figure 3: ENQUEUE using alternative policy.

Theorem 1 *If p concurrent processes are performing queue operations, tail points to a node at most $2p-1$ node from the end of the list.*

Proof: We need only consider processes performing ENQUEUE operations. Consider the last operation that succeeded in setting the *tail* pointer. At most $p-1$ other operations could have completed ENQUEUE operations, adding nodes to the end of the list, but failed to update *tail* due to a conflict with the first process. There are also at most p concurrent ENQUEUE operations that have inserted a node onto the list but not yet attempted to change the *tail* pointer. □

Yet a third alternative exists. Experiments with implementations of the above two policies indicated that the second policy resulted in enqueuing processes spending the majority of their time traversing the linked list, and using the first policy resulted in undue contention from the second CSW instruction. (Intuitively, the second CSW instruction was superfluous, since most of the time when a process fails to link an item onto the end of the list, the process that was successful will have already updated *tail*.)

These observations lead to a third policy, in which processes never update the *tail* pointer unless they have just successfully linked a new item onto the list. (This is simply the code in Figure 1 with the **if** ··· COMPARE&SWAP(···) deleted). Our experiments have shown this policy to result in the fastest code under conditions in which processes do not stall or stop.

Unfortunately, this change has the side effect of destroying the non-blocking property, since if a stopped enqueuing process fails to update the *tail* pointer no further enqueues can succeed. However, the non-blocking property can be restored by implementing a hybrid of the second and third policies; if the *tail* pointer is not updated, after a few tries an enqueuing process can simply search for the end of the list and update *tail* itself.

4 The ABA Problem

In the algorithms discussed in the last section, CSW is used in the following way: a pointer in the data structure is read, some computation is done to determine how to change the data structure, and then CSW is used to write a new value to the pointer only if it has not changed in the interim. A subtle problem can arise due to the fact that the CSW instruction does not really ensure that the pointer has not changed, but only that it has a certain value. If the pointer has changed, but by coincidence has the same value that it did when we originally read it, then the CSW instruction will succeed when it should fail.

To see how this problem can occur with our lock-free queue algorithm, consider a process that is attempting to dequeue an item. This process will read the value of *head*, determine the address of the second node on the linked list (by following the *next* of the first node), and then use CSW to make *head* point at the second node. If *head* has changed (due to other processes completing DEQUEUE operations), then the CSW instruction should fail. However, suppose that the block of memory making up the first node on the list is "recycled" and reused as a new node which is enqueued (after our process has already read the *head* pointer, but before it has tried the CSW). If this node happens to work its way up to the front of the list, then when our process performs its CSW, it will succeed, most likely corrupting the linked list structure.

This problem is known as the *ABA problem* [9]. The conventional solution to this problem has been to make use of a variant of CSW that operates on two adjacent words of memory at a time; one word is used to hold the pointer, and the other word is used to hold a tag that is incremented every time the pointer is changed. In this way, even if the pointer changes and then changes back to its original value, the tag will have changed and the CSW operation will not succeed.

4.1 The Safe Read Protocol

The preceding solution, in addition to requiring a stronger version of CSW, only makes it unlikely that the ABA problem will occur. In this section we propose an alternative solution that does not require a

double word version of CSW and which guarantees that the ABA problem will not occur.

We observe that when we are using CSW to manipulate pointers, the root cause of the ABA problem can be attributed to nodes being recycled and reused while some processes are still looking at them. Thus, we view the ABA problem as one of memory management. To solve it, we keep track of when it is safe to recycle a node by assigning each node a reference count, and not reusing a node until its reference count has gone to zero.

It is necessary to ensure that a process, when following a pointer in the data structure, atomically reads the pointer and increments the reference count of the pointed-at node. We call this operation a *safe read*. Pseudo-code for this operation is given in Figure 4. A corresponding RELEASE operation is used to decrement the reference count when a process is done with the pointer. If the count becomes zero, the memory block can be recycled.

```
SAFEREAD(q)
  loop:
      p ← q^.next
      if p = NULL then
          return p
      FETCH&ADD(p^.refct, 1)
      if p = q^.next then
          return p
      else
          RELEASE(p)
  goto loop
end
```

Figure 4: SAFEREAD operation.

The SAFEREAD and RELEASE operations are part of presumed to be supported by the underlying memory management library, which would also provide the usual ALLOC and FREE operations. Further details on how to implement such a library providing lock-free versions of these four operations can be found in the author's PhD thesis [22].

5 Array Implementations

It is common to implement sequential queues using a "circular array" data structure. This type of data structure has the advantage of lower overhead over linked list structures, since there is no need for *next* pointers, and it is unnecessary to allocate and deallocate memory on every operation.

Herlihy and Wing [7] present an array based queue that is non-blocking and linearizable, but which requires an array of infinite length. Wing and Gong [23] propose a modification to this algorithm removing the need for an infinite array; however, for both algorithms the running time of the DEQUEUE operation degrades as more ENQUEUE operations are done. An algorithm proposed by Treiber [20] also suffers from poor performance.

Gottlieb *et al.* [3] present an algorithm that is efficient, but which blocks under certain conditions. Although the probability of blocking occurring can be made smaller by increasing the size of the array used, it is not a true non-blocking algorithm.

5.1 A New Algorithm

We present a new algorithm for a lock-free queue based on an array. The algorithm is both non-blocking and linearizable. Our approach differs from previous algorithms in that it uses the CSW instruction, rather than the FAA instruction. The algorithms in the previous section all use FAA to allocate a position in the array when enqueuing.

The array is set up as a standard circular array. In addition to the data values the user wishes to store in the queue, there are three special values: HEAD, TAIL, and EMPTY. Initially, every location in the array is set to EMPTY, with the exception of two adjacent locations which are set to HEAD and TAIL. This represents the empty queue.

The algorithm works as follows. To enqueue the value x, a process finds the (unique) location containing the special TAIL value. The double-word[1] COMPARE&SWAP operation is then used to change the two adjacent locations from $\langle TAIL, EMPTY\rangle$ to $\langle x, TAIL\rangle$. Note that if the location adjacent to the one containing TAIL is not EMPTY, then the queue is full and the operation aborts.

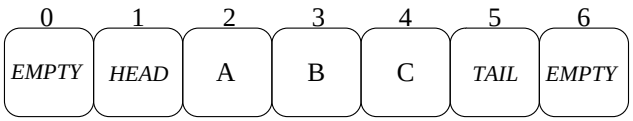
The DEQUEUE operation works in a similar manner, by using the CSW operation to change two adjacent locations from $\langle HEAD, x\rangle$ to $\langle EMPTY, HEAD\rangle$, returning the value x (provided of course that x was not TAIL, in which case the queue was empty).

In order to quickly find the locations in the array containing the values HEAD and TAIL, we keep two counts; the number of ENQUEUE and the number of DEQUEUE operations, modulo the size of the array. The counts are incremented (using FAA) whenever a process completes an operation, and can be used to determine the location of the HEAD or TAIL values

[1] The standard version of CSW could also be used, provided the data values to be stored were half-words.

to within p, where p is the number of concurrent processes. Note that keeping the indices of the ends of the queue in variables using CSW would not work, due to the ABA problem, and since we cannot prevent indices from being reused, the safe read protocol cannot be applied.

Figure 5 shows the same queue as in Figure 2, only implemented as described in this section. Again, notice that a process is in the midst of enqueuing item C, and that the tail count variable has not yet been incremented.




```
Head count = 1

Tail count = 4
```

Figure 5: Queue in a circular array.

This technique can also be used to provide lock-free stack and deque (double ended queue) abstract data types. However, it does have a subtle problem: on real machines, memory operations must be aligned. Our algorithm requires an unaligned CSW for every other operation, and thus would be infeasible on a real machine.

6 Experimental Results

It is difficult to characterize the performance of these types of concurrent algorithms, since their running time depends on the number of concurrent processes. For the algorithms in this paper, a sequence of operations will take time proportional to the product of the number of operations and the number of concurrent processes.

Theorem 2 *A sequence of n queue operations will take $O(np)$ time.*

Proof: The following proof can be generalized to any of the algorithms in this paper. Note that in the case that there is no contention from other processes, an operation will complete in constant time. Contention causes processes to do more work due to two things: looking for the proper place in the data structure to perform the operation (e.g., traversing nodes to find the end of the linked list), and retrying failed CSW operations.

Note that from Theorem 1, the time to find the end of the linked list from the *tail* pointer is at most $O(p)$. For the array implementation, the enqueue and dequeue operation counters will also be within $O(p)$ of their "correct" value, since at most p processes will have performed an operation but not incremented the counter.

New items may be added to the linked list (or the array) while an operation is in progress. The work to traverse over these items will total $O(np)$, since each operation can cause at most $p-1$ concurrent operations to have to traverse its item.

Finally, by a similar argument, the total work to retry failed CSW operations is $O(np)$ as well. The bound follows. □

This bound reflects the worst-case behavior. In order to better compare the performance of the different lock-free algorithms discussed in this paper, as well as equivalent lock-based algorithms, we have implemented several of them using the Proteus parallel architecture simulator [2]. All numerical results are quoted in "cycles" as simulated by Proteus.

To asses the performance of these algorithms, we measured two quantities: the sequential latency of each operation (i.e., the time for an operation to be performed with no contention from other processes), and the latency of each operation under varying amounts of contention from concurrent operations. Contention was modeled by assuming an infinite number of processors which performed queue operations with interarrival times following an exponential distribution.

In addition to the lock-free data structures described in Section 3, we also implemented a concurrent queue using mutual exclusion. We tested the following locking mechanisms: simple test-and-set locks, test and test-and-set locks, and the ticket locks and queue locks of Mellor-Crummey and Scott [13].

In order to measure only the algorithm performance, we did not implement the safe read protocol in our tests; we avoid the ABA problem by not reusing nodes on the linked list. In addition, to remove the overhead of memory allocation from our results, we use a pre-allocated buffer of nodes for the tests.

Some method of managing contention between processes is necessary, typically by "backing off" after a failed CSW or lock acquisition. This type of backoff can be very sensitive to tuning parameters; however, for these initial experiments, we used the same backoff algorithm (a simple exponential backoff) for all algorithms, with the exception of the ticket and queue

locks. Ticket locks use a proportional backoff procedure, while queue locks do not require any backoff.

6.1 Latency Under Contention

Figure 6 graphs the average latency of the ENQUEUE operation. These results represent the mean over 1000 operations, at varying levels of contention (as measured by the average interarrival time of operations). Figure 7 shows the same results for the DEQUEUE operation.

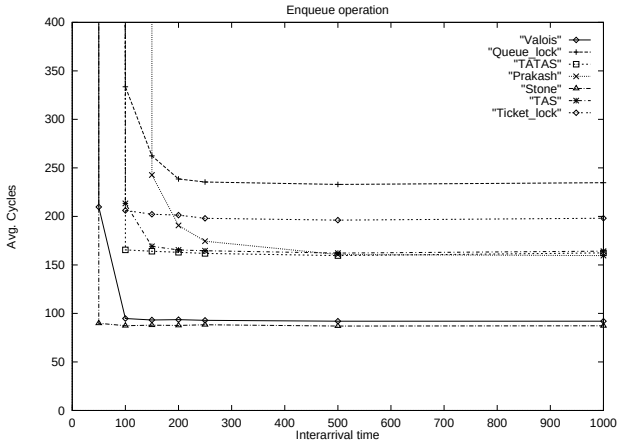


Figure 6: Average latency for ENQUEUE operation.

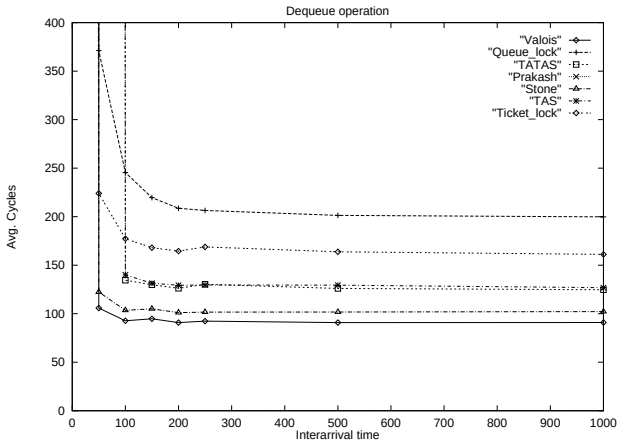


Figure 7: Average latency for DEQUEUE operation.

Our preliminary results indicate that lock-free queues are competitive with data structures using mutual exclusion. The algorithm presented in this paper is not only efficient, but is non-blocking and provides linearizable behavior, making it promising for practical applications.

Algorithm	ENQUEUE	DEQUEUE
Stone	77	85
Valois	81	73
test & test-and-set	129	94
test-and-set	136	100
Prakash *et al.*	145	146
ticket lock	158	122
Queue lock	228	197

Table 1: Sequential latency of queue operations.

6.2 Sequential Latency

Table 1 contains the results of the sequential latency tests.

The algorithm of Stone and the algorithm presented in this paper are the two fastest. This can be attributed to their simplicity; for the sequential case, when there is no contention, the algorithms execute only a few instructions.

7 Summary

We have presented two new data structures and algorithms for implementing a concurrent queue which is non-blocking and linearizable. We have also proposed a new solution to the ABA problem. Initial experiments comparing our first algorithm to other alternatives, including data structures using mutual exclusion, indicate that it is practical.

7.1 Future Research

Further experiments are needed to determine the performance of these algorithms under varying conditions. For example, the lock-free approach is attractive if processes can suffer slow-downs inside of their critical section; experiments are needed to determine under what conditions lock-free data structures outperform their lock-based counterparts.

In this paper we have focused on the queue abstract data type; other data types could benefit from lock-free methods as well. Other researchers have presented lock-free algorithms for a variety of problems, including disjoint-sets [1], garbage collection [5], priority queues [10], and a multiprocessor operating system kernel [11]. We are currently investigating implementations of other lock-free data structures such as linked lists and binary search trees [22].

The universal constructions mentioned in Section 2 can provide lock-free data structures with the wait-free property. While this is a desirable property, it generally requires providing a higher level of coordination among the processes, and introduces a large overhead. We believe a better approach is to ensure fairness through scheduling and backoff policy. More work is needed in determining how best to do this.

The array-based implementation presented in Section 5 is not feasible on real machines due to alignment problems. However, the algorithm is far more efficient that other solutions using arrays. Is there an algorithm that is both realistic and efficient ?

Lock-free data structures provide an alternative method of synchronization which can have advantages over spin-locking. Research is needed to determine the extent of these advantages, and how they can be exploited in applications.

References

[1] R. Anderson and H. Woll. Wait-free parallel algorithms for the union-find problem. In *Proceedings of the 23rd ACM Symposium on Theory of Computation*, pages 370–380, 1991.

[2] E. Brewer, C. Dellarocas, A. Colbrook, and W. Weihl. PROTEUS: A high-performance parallel-architecture simulator. In *Proceedings of the 1992 ACM SIGMETRICS and PERFORMANCE '92 Conference*, June 1992.

[3] A. Gottlieb, B. 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. Herlihy. A methodology for implementing highly concurrent data structures. In *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 197–206, 1990.

[5] M. Herlihy and J. 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.

[6] M. Herlihy and J. Wing. Axioms for concurrent objects. In *14th ACM Symposium on Principles of Programming Languages*, pages 13–26, 1987.

[7] 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.

[8] K. Hwang and F. Briggs. *Computer Architecture and Parallel Processing*, pages 559–562. McGraw-Hill, 1985.

[9] IBM T.J. Watson Research Center. *System/370 Principles of Operation*, 1983.

[10] A. Israeli and L. Rappoport. Efficient wait-free implementation of a concurrent priority queue. In *Proceedings of the 1993 Workshop on Distributed Algorithms*, pages 1–16, 1993.

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

[12] J. Mellor-Crummey. Concurrent queues: Practical fetch-and-ϕ algorithms. Technical Report 229, University of Rochester, November 1987.

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

[14] Peter Moller-Nielsen and Jorgen Staunstrup. Problem-heap: A paradigm for multiprocessor algorithms. *Parallel Computing*, 4:63–74, 1987.

[15] S. Plotkin. Sticky bits and universality of consensus. In *Proceedings 8th ACM Symposium on Principles of Distributed Computing*, pages 159–175, August 1989.

[16] S. Prakash, Y. Lee, and T. Johnson. A non-blocking algorithm for shared queues using compare-and-swap. In *Proccedings 1991 International Conference on Parallel Processing*, volume 2, pages 68–75, 1991.

[17] S. Prakash, Y. Lee, and T. Johnson. Non-blocking algorithms for concurrent data structures. Technical Report TR91–002, University of Florida, 1991.

[18] R. Sites. Operating systems and computer architecture. In H. Stone, editor, *Introduction to Computer Architecture*, chapter 12, pages 594–604. Science Research Associates, 2nd edition, 1980.

[19] J. Stone. A simple and correct shared-queue algorithm using Compare-and-Swap. In *Proceedings of Supercomputing '90*, pages 495–504, 1990.

[20] R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, April 1986.

[21] J. Turek. *Resilient Computation in the Presence of Slowdowns*. PhD thesis, New York University, 1991.

[22] J. Valois. PhD thesis, Rensselaer Polytechnic Institute, Troy, NY, in preparation.

[23] J. Wing and C. Gong. A library of concurrent objects and their proofs of correctness. Technical Report CMU–CS–90–151, Carnegie Mellon University, 1990.