

A practical nonblocking queue algorithm using compare-and-swap *

Chien-Hua Shann

Ting-Lu Huang

Cheng Chen

Dept. of Computer Science and Information Engineering

National Chiao Tung University

1001 Ta-Hsueh Road, Hsin-Chu, Taiwan 30010

E-mail: shann@ieee.org, {tlhuang, cchen}@csie.nctu.edu.tw

Abstract

Many nonblocking algorithms have been proposed for shared queues. Previous studies indicate that link-based algorithms perform best. However, these algorithms have a memory management problem: a dequeued node can not be freed or reused without proper handling. The problem is usually overlooked; one just assumes the existence of a lower level mechanism, which takes care of all the details of handling the problem. Employing such a mechanism incurs significant overheads, and consequently the link-based queues may not perform as well as claimed. A new nonblocking queue algorithm based on a finite array is proposed in this paper. Comparing with the link-based algorithms, the new algorithm provides the same degree of concurrency without being subject to the memory problem, hence suggests a good performance.

Keywords: concurrent queue, nonblocking, compare-and-swap, linearizability

1. Introduction

Algorithms for concurrent data structures can be classified as either blocking ones or nonblocking ones. Blocking algorithms are those in which a process trying to read or modify the data structure isolates or locks part or all of the data structure to prevent interference from other processes. A deadlock may occur in such algorithms if a process fails or is halted. Such algorithms also suffer from significant performance degradation when a process is delayed at an inopportune moment [6][7][9]. Possible sources of delay include process preemption, page faults, remote memory access, and cache misses. Nonblocking algorithms, on the other hand, ensure that the data structure is always accessible to all processes. Such algorithms guarantee that some

active process can finish its operation in a finite number of steps, no matter whether or not there are other processes being halted or delayed. Nonblocking algorithms are thus more robust in the presence of process failures. Nonblocking algorithms outperform blocking ones when process delay cannot be ignored.

Concurrent queues are widely used in a variety of parallel applications and operating system implementations. Many nonblocking queue algorithms have been proposed. They can be classified into three categories: queues resulted from universal constructions, array-based queues, and link-based queues. A universal construction is a transformation that constructing a concurrent nonblocking data structure from a sequential one. Herlihy [2] first presented such a transformation. It requires the entire data structure to be copied on every update, and the concurrency of the data structure is also restricted by the transformation, resulting in poor performance. Herlihy proposes an optimization by which the programmer can avoid some fraction of the copying for certain data structures [2]. Alemany and Felten [1] and LaMarca [5] also proposed techniques to reduce unnecessary copying associated with Herlihy's methodologies. However, even with the optimizations applied, nonblocking queues resulted from universal constructions perform worse than the link-based queues.

The second category is the nonblocking queues based on finite or infinite arrays. Herlihy and Wing [3] gave a nonblocking queue algorithm that requires an infinite array. Wing and Gong [14] proposed a modification that removes the need for an infinite array. Treiber [11] also presented a similar algorithm that does not use an infinite array. However, the dequeue operation incurs an accumulative cost: the running time of dequeue operation is proportional to the number of enqueue operations that have been completed since initialization. Poor performance is expected as a result of the accumulative cost. Valois [13][12] gave an array-based algorithm which requires a special unaligned compare-and-swap which is not implemented on any existing machine.

*This work was supported by the National Science Council, Republic of China under grant NSC89-2213-E-009-027

The third category consists of algorithms based on linking lists. Prakash, Lee, and Johnson [8][9] presented a link-based nonblocking algorithm that requires enqueueing and dequeuing processes to take a snapshot of the queue in order to determine its state prior to an update. Valois [13][12] improved the algorithm by avoiding the snapshot and by allowing more concurrency by keeping a dummy node at the front of the linked-list, thus simplified a special case associated with empty and single item queues. Michael and Scott [6][7] also presented a link-based algorithm similar to the one proposed by Valois. These link-based queue algorithms enjoy a high degree of concurrency, and are efficient.

Performance analyses in [6][9][13] suggest that link-based queues outperform the ones of other categories. Unfortunately, link-based queues are subject to a memory management problem: once a dequeuing process removes a node from the list, the node cannot be freed because there may be some other processes that are still accessing it. Since memory is a limited resource, there must be some mechanism to help freeing or reusing the dequeued node. Suggested solutions to deal with this problem introduce significant overheads, and the concurrency of the queue may also be restricted. The memory management problem will be discussed in detail in section 2.

Our goal is to design a nonblocking concurrent queue algorithm that supports enqueue and dequeue operations. The new algorithm is based on a finite array, and use popular `compare-and-swap` primitives. The features of the algorithm include (1) absence of the memory management problem associated with the link-based algorithms, (2) preserving the concurrency of the queue, and (3) dequeue time without the accumulative cost associated with other array-based algorithms. The rest of the paper is organized as follows. Section 2 describes the memory management problem and strategies to deal with it. Section 3 describes a new array-based concurrent queue algorithm. The correctness of the algorithm is presented in section 4. Section 5 is our conclusions.

2. The Memory Management Problem

In link-based nonblocking queue algorithms, a queue is expressed as a linked-list of nodes; an enqueueer inserts a node into the list, and a dequeuer removes a node from the list; processes can access the list concurrently. The memory management problem occurs when a dequeuer removes a node from the list, and at this point, some other processes may still want to access the node. The dequeued node cannot be freed, or subsequent accesses to the node will fail. Two strategies are suggested to deal with this problem.

The first strategy is to reuse a dequeued node. If this strategy is to be used, the queue algorithm has to maintain a pool of free nodes. Whenever an enqueueer requires a node,

it obtains one from the free pool; whenever a dequeuer removes a node from the queue, the node is added to the free pool for later use. A concurrent stack is usually suggested to implement the free pool [6][9].

Several drawbacks come along with this strategy: (1) maintaining the free pool incurs significant overheads, and (2) the concurrency of the queue may also be restricted by the implementation of the free pool. To see how the concurrency is restricted, suppose a stack is used to implement the free pool as suggested. The insert and remove operations of the pool are realized via the `push` and the `pop` operations of the stack. These two operations access the same data item (the stack top); concurrent execution of a `push` and a `pop` operations must therefore be serialized. Thus the concurrency of the queue, is restricted by the stack.

The second strategy is to actually free a dequeued node; the dequeuer has to ensure that no other process is still accessing the node. Valois [12] presented a mechanism that successfully implements this strategy. 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 counter atomically. The reference counter reflects the number of pointers that point to the node. A node can be freed only if the value of its reference counter is zero.

The mechanism is obviously inefficient; however, it is also impractical, due to the infinite memory requirement discussed below. Suppose a process accesses a pointer to a node and is then delayed; while the process is not running, other processes can enqueue and dequeue arbitrary number of additional nodes. All of the additional nodes are successors of the delayed node in the list; each of them has a reference counter value of at least one. None of them 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 [6].

All link-based nonblocking queue algorithms suffer from the memory management problem; unfortunately, existing solutions are inefficient. The link-based algorithms may not perform as well as claimed.

3. The Algorithm

Our goal is to design a nonblocking concurrent queue algorithm that supports enqueue and dequeue operations. The new algorithm, like most nonblocking queue algorithms, uses the `compare-and-swap` atomic instruction, as depicted in figure 1. The instruction is used in the following manner: *shared* is a shared variable and *old* is the private copy of it made earlier by a process; *new* is the value which the process is attempting to update *shared* by means of the `compare-and-swap` instruction. The attempt succeeds

```

compare-and-swap(shared, old, new): Boolean
  if shared = old then
    shared := new
    return TURE
  elseif
    return FALSE
  endif

```

Figure 1. The compare-and-swap instruction

if *old* is equal to *shared*; in this case, one can expect that *shared* is not modified since the private copy is made and the update is consistent. Concurrent algorithms using the compare-and-swap instruction are based on this expectation, i.e., they assume an update is always consistent if the compare-and-swap instruction succeeds.

Indeed, the update may be inconsistent, as a result of the A-B-A problem [4][10]: if during the time between the private copy is made and the attempt to update *share*, there are some other processes that modify *shared* to another value and then modify it back to the old value again, the attempt will succeed while it is expected not to. The most common solution is to associate a modification counter with each shared data item that is accessed by a compare-and-swap instruction; the counter is always incremented at each successful attempt to update the data item. This solution does not guarantee that the A-B-A problem will not occur, but it makes it extremely unlikely [10]. Our algorithm employs this solution to prevent the A-B-A problem.

The idea behind the new algorithm is to implement a *circular list* based on a finite array. An item is added only to the rear end of the list; an item is removed only from the front end of the list. In this way, the circular list successfully serves as a FIFO queue.

Figure 2 presents the data structure and the pseudo-code of the algorithm for the queue. The data structure consists of a finite array and two counters. Each slot in the array, namely *Q*, comprises a *val* part that stores a queue item, and a *ref* part that serves as a modification counter to prevent the A-B-A problem. Two counters, namely *FRONT* and *REAR*, are used to locate the front end and the rear end of the list. We say that a slot is empty if its *val* part is NULL. We also say that the queue is empty if *FRONT* equals *REAR*; it is full if *FRONT*+*L* equals *REAR*. In addition, $Q[FRONT \bmod L]$ is referred to as the front of the list; $Q[REAR \bmod L]$ is referred to as the rear of the list.

The enqueue operation comprises two steps: an enqueueer first stores an item into the rear of the list, it then increments *REAR* by one. A compare-and-swap in-

struction is used to obtain a consistent update for each step. The enqueue operation is nonblocking. An enqueueer can proceed with its operation only if the queue is not full and the rear of the list is empty. If the queue is full, the enqueueer waits for some dequeueer to remove an item from the list. If the rear of the list is not empty, there must be some other process that is at the midst of its steps. In this case, the enqueueer tries to help the other process complete its steps.

The dequeue operation works in a similar manner. It comprises two steps: a dequeueer first empties the front of the list, it then increments *FRONT* by one. A compare-and-swap instruction is used to obtain a consistent update for each step. The dequeue operation is nonblocking. A dequeueer can proceed with its operation only if the queue is not empty and the front of the list is not empty. If the queue is empty, the dequeueer waits for some enqueueer to add an item to the list. If the front of the list is empty, there must be some other process that is at the midst of its steps. In this case, the dequeueer tries to help the other process complete its steps.

The algorithm is based on a finite array; the memory management problem is thus avoided. There is no accumulative cost associated with the algorithm; the execution time of a dequeue operation is independent on the number of enqueue operations that have been completed since initialization. In addition, provided the queue is neither full nor empty, an enqueue operation is never forced to wait for a dequeue operation to complete; a dequeue operation is never forced to wait for an enqueue operation to complete. Thus, enqueue and dequeue operations can proceed independently and concurrently. This is enough to say that the algorithm preserves the concurrency of the queue data type.

Our algorithm is practical and efficient. Indeed, it is the first array-based nonblock queue algorithm which is both practical and efficient. It is practical because it is based on a finite array and uses the popular compare-and-swap instruction. It is efficient because it is subject to neither the memory management problem nor the accumulative cost of the dequeue operation, and it preserves the concurrency of the queue data type. The efficiency of the algorithm suggests a good performance. The correctness of the algorithm is discussed in the next section.

4. Correctness

We show that the algorithm is correct by showing that it satisfies certain safety and liveness properties. We also show that the queue is linearizable. The proof is done under the assumption that the A-B-A problem does not occur.

CAS denotes the compare-and-swap instruction.
$\langle x \parallel y \rangle$ denotes the concatenation operation of x and y .
Variables in uppercase letters are shared; those in lowercase letters are local.

Q : **array** $[0..L-1]$ of **structure** $\{val: \mathbf{qitem_t}; ref: \mathbf{counter_t}\}$
FRONT, REAR: **counter_t**

enqueue($X: \mathbf{qitem_t}$)

```
e1  enq_try_again:
e2    rear := REAR                                # read REAR
e3    x := Q[rear mod L]                          # read the rear of the list
e4    if rear ≠ REAR then goto enq_try_again endif # are rear and x consistent?
e5    if rear = FRONT+L then goto enq_try_again endif # is queue full?
e6    if x.val = NULL then                        # is the rear of the list empty?
e7      if CAS(Q[rear mod L], x, <X || x.ref+1>) then # try to store an item
e8        CAS(REAR, rear, rear+1)                 # try to increment REAR
e9        return                                  # enqueue is done
e10   endif
e11  elseif                                       # the slot is not empty
e12    CAS(REAR, rear, rear+1)                   # help others increment REAR
e13  endif
e14  goto enq_try_again                          # enqueue failed, try again
```

dequeue() : $\mathbf{qitem_t}$

```
d1  deq_try_again:
d2    front := FRONT                              # read FRONT
d3    x := Q[front mod L]                        # read the front of the list
d4    if front ≠ FRONT then goto deq_try_again endif # are front and x consistent?
d5    if front = REAR then goto deq_try_again endif # is queue empty?
d6    if x.val ≠ NULL then                       # is the front of the list nonempty?
d7      if CAS(Q[front mod L], x, <NULL || x.ref+1>) then # try to remove an item
d8        CAS(FRONT, front, front+1)             # try to increment FRONT
d9        return(x.val)                          # dequeue is done
d10   endif
d11  elseif
d12    CAS(FRONT, front, front+1)                # help others increment FRONT
d13  endif
d14  goto deq_try_again                          # dequeue failed, try again
```

Figure 2. Data structure and pseudo code of the algorithm

4.1. Safety

We begin the proof with defining four events. The first two events are for the enqueue procedure. A *STORE* event occurs whenever an enqueueer successfully stores an item into a slot (line e7), and an *INC_REAR* event occurs whenever an enqueueer successfully increments *REAR* by one (line e8, e12). Similarly, we define two events for the dequeue procedure. An *EMPTY* event occurs whenever a dequeueer successfully empties a slot (line d7), and an *INC_FRONT* event occurs whenever a dequeueer success-

fully increments *FRONT* by one (line d8, d12). The following lemmas state some basic properties of the algorithm.

Lemma 1 *An INC_REAR event can occur, only if all of the following statements are true: 1. REAR is not modified since it was last read at line e2. 2. Q[REAR mod L] is nonempty.*

Proof. The compare-and-swap instruction and the assumption of the absence of the A-B-A problem ensures the first statement is true. We now show the second statement is also true. By the first statement, we know the read at

line e3 indeed reads $Q[REAR \bmod L]$, and by the check at line e6, the read value is nonempty. Since only the dequeuer can empty the slot, we need to show no such dequeuer exists. Suppose there is a dequeuer that empties the slot $Q[REAR \bmod L]$, this implies $FRONT$ equals $REAR$. However, this is a contradiction to the check at line d5, which simply prohibits any dequeuer to proceed when the queue is empty. \square

Lemma 2 *A STORE event can occur, only if all of the following statements are true: 1. $Q[rear \bmod L]$ is empty. 2. $Q[rear \bmod L]$ is not modified since it was read at line e3. 3. $REAR$ is not modified since it was last read at line e2.*

Proof. To show that the first statement is true, suppose $Q[rear \bmod L]$ is nonempty. However, by line e6, x must be empty. This implies that the compare-and-swap instruction must fail, a contradiction. To show the second statement is true, suppose $Q[rear \bmod L]$ is modified after the read at line e3, the ref part of it must have changed. Therefore the compare-and-swap instruction must fail, a contradiction.

It remains to show that the third statement is also true. Suppose $REAR$ is modified since it was last read at line e2. This implies that there must be some other enqueueer that successfully performs either one of the two compare-and-swap instructions in line e8 and line e12. However, Lemma 1 implies that in either case, $Q[rear \bmod L]$ must be nonempty, again, a contradiction. \square

Lemma 3 *The STORE events and the INC_REAR events occur alternatively.*

Proof. Lemma 3 can be easily proved by the previous lemmas and by induction on the number of enqueue operations performed. The detail are skipped here. \square

Lemma 1, Lemma 2 and Lemma 3 are associated with the enqueue procedure. We can have similar lemmas for the dequeue procedure. The proofs are also similar to those of Lemma 1, Lemma 2 and Lemma 3. They are thus omitted.

Lemma 4 *An INC_FRONT event can occur, only if all of the following statements are true: 1. $FRONT$ is not modified since it was last read at line d2. 2. $Q[FRONT \bmod L]$ is nonempty.*

Lemma 5 *An EMPTY event can succeed, only if all of the following statements are true: 1. $Q[front \bmod L]$ is empty. 2. $Q[front \bmod L]$ is not modified since it was read at line d3. 3. $FRONT$ is not modified since it was last read at line d2.*

Lemma 6 *The EMPTY events and the INC_FRONT event occur alternatively.*

The safety properties of the algorithm is as follows. To show that the algorithm satisfies these safety properties is straightforward by these lemmas.

Property 1 *A dequeuer removes an item only from the front of the list.*

Property 2 *A dequeuer removes an item only from the front of the list.*

Property 3 *Each array slot in the list, except the front and the rear of the list, is nonempty; Each slot not in the list is empty.*

4.2. Linearizability

Linearizability [3] is a correctness condition for concurrent objects. A concurrent object is linearizable if every (concurrent) computation accepted by it is equivalent to some legal sequential computation, and the order of enqueue and dequeue operations in the concurrent computation is respected by the sequential one. We show our algorithm is linearizable by stating how a legal sequential computation can be constructed according to our concurrent queue algorithm.

We order every operation of the concurrent queue by the time the *STORE* event occurs, if it is an enqueue operation, or the *EMPTY* event occurs, if it is a dequeue operation. In this way we can construct a new computation in which every operations in it are ordered. The new computation is therefore a sequential one. Obviously the order of the concurrent computation is respected by the new sequential one, and it is a legal queue computation. This is because (1) by Property 1 and Property 2, we can show that the queue obeys the FIFO order, and (2) by Property 3, we can show that items do not spontaneously appear, and they do not spontaneously disappear.

4.3. Liveness: nonblockingness

An algorithm is nonblocking if there are non-delayed processes attempting to perform operations, an operation is guaranteed to complete within a finite time. We show that our algorithm is nonblocking by showing that a process loops beyond a finite number of times only if another processes completes an operation on the queue.

Provided the queue is not full, an enqueue operation cannot proceed with its operation only if (1) the conditional test in line e4 fails, (2) the conditional test in line e6 fails, or (3) the compare-and-swap instruction in line e7 fails. We argue that, in each case, an enqueue operation cannot

proceed only if there is some other process that has completed its operation; nonblockingness is thus sustained.

- (1) The conditional test in line e4 fails only if *REAR* is written by some other process. By property 3 and the assumption that the queue is not full, the rear of the list must be empty after *REAR* is written. Therefore some other process completes its operation in a finite number of steps.
- (2) The conditional test in line e6 fails only if the rear of the list is nonempty. After the `compare-and-swap` instruction in line e12 the rear of the list must become empty. Therefore some other process completes its operation in a finite number of steps.
- (3) The `compare-and-swap` instruction in e7 fails only if another process has completed its operation.

We need to show the dequeue procedure is also non-blocking. However, the proof is very much similar to how the nonblockingness of the enqueue procedure is shown. Thus, it is again omitted.

5. Conclusion

Concurrent queue is a frequently used data object for various applications and parallel programs. A practical non-blocking algorithm is thus essential to system availability and performance. In this paper, we have presented a practical nonblocking queue algorithm based on a finite array and the popular `compare-and-swap` atomic instruction. Our algorithm is more efficient than other array-based algorithms, which are subject to an accumulative cost in the dequeue operation. Comparing with the link-based algorithms, our algorithm provides the same degree of concurrency without being subject to the memory management problem, hence suggested a good performance. In particular, the proposed algorithm is the first array-based algorithm that promises such performance.

We also have discussed the memory management problem, which is usually overlooked. The problem is fundamental; all link-based nonblocking algorithms appear to suffer from it. Nevertheless, there is no practical and efficient solution that has been proposed; the performance degradation owing to this problem is not carefully studied either. We think solving this problem would be an interesting research direction.

Because of lack of time, we cannot provide a performance evaluation of the proposed algorithm in this paper. We will still working on it and will present the results in the next version of this article.

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*, pp. 125–134, 1992.
- [2] M. Herlihy, "A methodology for implementing highly concurrent data structures," in *Proceedings of the Second ACM Symposium on Principles and Practice of Parallel Programming*, pp. 197–206, 1990.
- [3] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 3, pp. 463–492, 1990.
- [4] International Business Machines Corp., *System/370 Principles of Operation*, 1983.
- [5] A. LaMarca, "A performance evaluation of lock-free synchronization protocols," in *Proceedings of the Thirteenth Symposium on Principles of Distributed Computing*, pp. 130–140, 1994.
- [6] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *Proceedings of the Fifteenth Symposium on Principles of Distributed Computing*, pp. 267–275, 1996.
- [7] M. M. Michael and M. L. Scott, "Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors," *Journal of Parallel and Distributed Computing*, pp. 1–26, 1998.
- [8] 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*, pp. 68–75, 1991.
- [9] S. Prakash, Y. H. Lee, and T. Johnson, "A nonblocking algorithm for shared queues using compare-and-swap," *IEEE Transactions on Computers*, vol. 43, pp. 548–559, May 1994.
- [10] J. M. Stone, "A simple and correct shared-queue algorithm using Compare-and-Swap," in *Proceedings of Supercomputing '90*, pp. 495–504, 1990.
- [11] R. K. Treiber, "Systems programming: Coping with parallelism," Tech. Rep. RJ 5118, IBM Almaden Research Center, Apr. 1986.
- [12] J. D. Valois, *Lock-Free Data Structures*. PhD thesis, Rensselaer Polytechnic Institute, Department of Computer Science, 1995.
- [13] J. D. Valois, "Lock-free linked lists using compare-and-swap," in *Proceedings of the Fourteenth Symposium on Principles of Distributed Computing*, pp. 214–222, 1995.
- [14] J. M. Wing and C. Gong, "A library of concurrent objects and their proofs of correctness," Technical Report CMU-CS-90-151, Carnegie-Mellon University, 1990.