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 enqueuer 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.

[2]A *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-linearizable[3] 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 enqueuing 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

[3] 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 [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, enqueuers 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 enqueued 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:               if next.ptr == NULL                       # Is queue empty?
D8:                  return FALSE                           # 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
                  # Read value before CAS, otherwise another dequeue might free the next node
D12:              *pvalue = next.ptr–>value
D13:              if CAS(&Q–>Head, head, <next.ptr, head.count+1>)     # Try to swing Head to the next node
D14:                 break                                  # Dequeue is done. Exit loop
D15:              endif
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 enqueuing 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 enqueuing 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 enqueuing 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 backoff [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 $\lfloor 10^6/p \rfloor$ or $\lceil 10^6/p \rceil$ times, for a total of one million enqueues and dequeues. The "other work" consists of approximately 6 μs 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 complete 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-

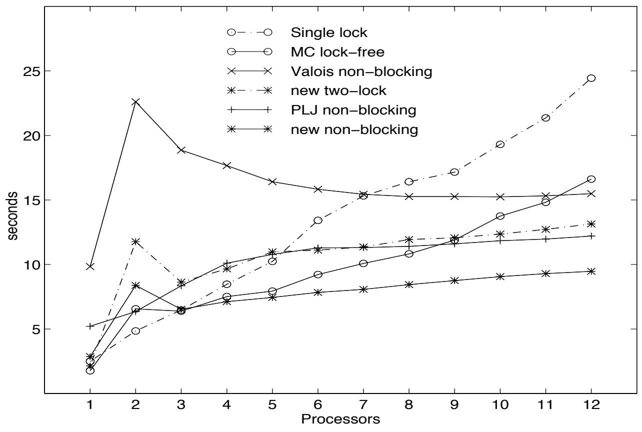

Figure 3: Net execution time for one million enqueue/dequeue pairs on a dedicated multiprocessor.

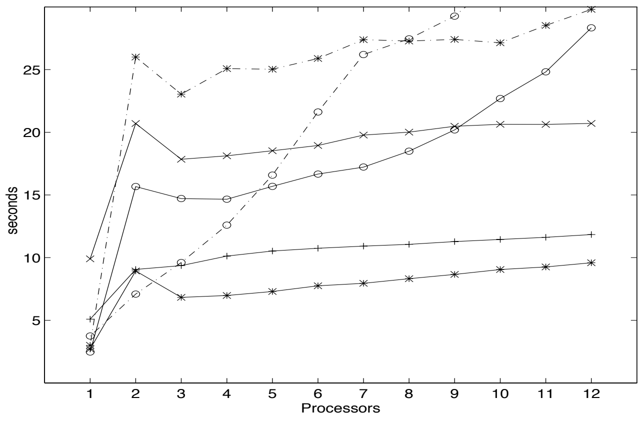

Figure 4: Net execution time for one million enqueue/dequeue pairs on a multiprogrammed system with 2 processes per processor.

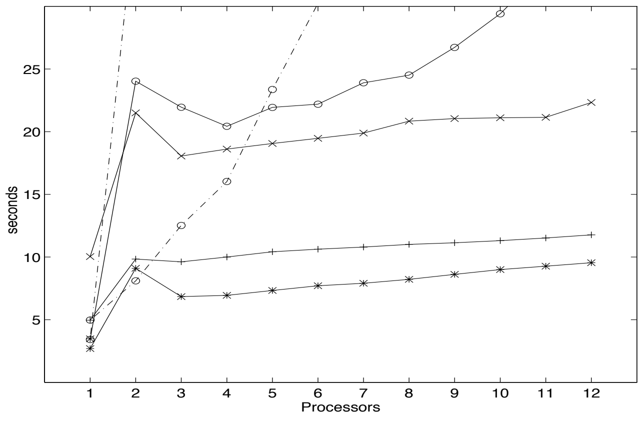

Figure 5: Net execution time for one million enqueue/dequeue pairs on a multiprogrammed system with 3 processes per processor.

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 CUCS-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.