IEEE TRANSACTIONS ON COMPUTERS, VOL. 43, NO. 5, MAY 1994

A Nonblocking Algorithm for Shared Queues Using Compare-and-Swap

Sundeep Prakash, Yann Hang Lee, and Theodore Johnson

***Abstract*—Nonblocking algorithms for concurrent objects guarantee that an object is always accessible, in contrast to blocking algorithms in which a slow or halted process can render part or all of the data structure inaccessible to other processes. A number of algorithms have been proposed for shared FIFO queues, but nonblocking implementations are few and either limit the concurrency or provide inefficient solutions. In this paper we present a simple and efficient nonblocking shared FIFO queue algorithm with *O(n)* system latency, no additional memory requirements, and enqueuing and dequeuing times independent of the size of the queue. We use the *compare&swap* operation as the basic synchronization primitive. We model our algorithm analytically and with a simulation, and compare its performance with that of a blocking FIFO queue. We find that the nonblocking queue has better performance if processors are occasionally slow, but worse performance if some processors are always slower than others.**

I. Introduction

ALGORITHMS for concurrent access to data structures fall in two broad categories: blocking and nonblocking. 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. The problem with the blocking approach is that in an asynchronous system with processes having different speeds, a slower process might prevent faster processes from accessing the data structure. Non-blocking algorithms, on the other hand, ensure that the data structure is always accessible to all processes and a process that's inactive (whether temporarily or permanently) cannot render the data structure inaccessible. Such an algorithm guarantees that some active process will be able to complete an operation in a finite number of steps [3], making the algorithm robust with respect to process failures.

Shared queues are required in a number of multiprocessing applications. A nonblocking implementation is desirable due to its robustness and consistent performance in the presence of processes with varying speeds. A number of shared queue implementations exist. Lamport [7] gives a wait free implementation but restricts the concurrency to a single enqueuer and a single dequeuer (an algorithm is wait-free if every process can complete an operation in a finite number of steps [3]). Gottlieb *et al.* [2] present a blocking algorithm for enqueuing and dequeuing using the *replace-add* and *swap* operations for synchronization. This implementation allows a for high degree of parallelism, limited only by the predefined maximum queue size. However, it is possible for an enqueuer or dequeuer to block other dequeuers and enqueuers. Stone [11] proposes a "nondelaying" implementation using the *compare&swap* operation for synchronization, which allows an arbitrary number of enqueuers and dequeuers. However, a faulty or slow enqueuer can block all the dequeuers. Herlihy and Wing [4] give a nonblocking algorithm using the *compare&swap* operation. This also permits an arbitrary number of enqueuers and dequeuers but is impractical, as it requires an infinite array size for continued operation. Treiber [13] gives a nonblocking algorithm for concurrent FIFO access to a shared queue. The enqueue operation requires only one step but the time taken for the dequeue operation is proportional to the number of objects in the queue, so the algorithm is inefficient for large queue lengths and many simultaneous dequeue attempts.

Herlihy [3] presents a general methodology for automatically transforming any sequential data structure implementation into a concurrent nonblocking or wait-free implementation, but the memory requirements for each concurrent process grow in proportion to the total number of concurrent processes. Prakash [9] and Turek, Shasha, and Prakash [15] propose a transformation for creating nonblocking algorithms from locking algorithms, and Turek [14] proposes a transformation that creates wait-free algorithms. These transformations require only $O(1)$ space overhead per processor. The algorithm presented here is considerable simpler and more efficient than those provided in [9], [14], [15], and in fact a previous version of this work is used in [14].

In this paper, we present an efficient nonblocking algorithm for a shared FIFO queue. This implementation uses the *compare&swap* operation for synchronization. As proved elsewhere [5], it is impossible to design nonblocking or wait-free implementations of many simple data structures using other well-known synchronization primitives (i.e., *read, write, test&set, fetch&add* and *swap*). However, the *compare&swap* operation in its simple form has a standard difficulty (the A-B-A problem [16]). The solution is to use the modified *compare&swap* operation (also described in [16]), which is what we have done. The nonblocking property is achieved by processes cooperating to complete an operation started by one of the processes. Both the enqueue and dequeue operations are independent of the size of the queue.

Manuscript received July 15, 1992; revised January 15, 1993. This work was supported in part by a grant from Florida High Technology and Industry Council.

The authors are with the Department of Computer and Information Sciences, University of Florida, Gainesville, FL 32611.

IEEE Log Number 9214076.

The rest of the paper is organized as follows: In Section II we briefly discuss the *compare&swap* operation and the A-B-A problem. Section III describes the data structures we use for implementing the queue and Section IV contains the algorithm, the proof of its correctness and its analysis. We model the performance of the algorithm in Section V. In Section VI, we conclude by summing up the results.

II. The A-B-A Problem

We use the implementation of the *compare&swap* operation found in the IBM/370 architecture (also used in the Cedar supercomputer at the University of Illinois [17], the BBN [1], and the Motorola 68000 family [8]). It is a three-operand atomic instruction of the form **CS(A, B, C). A, B** and and **C** are one-word variables.[1] The instruction does the following:

If **A** equals **C**
then put **B** into **C**, return condition code 0
else put **C** into **A**, return condition code-1.

The compare-and-swap instruction is used in the following manner: **C** is a shared variable and **A** is the private copy of it made sometime earlier by a process. **B** is the value which the operation is attempting to put in **C.** The operation is allowed to do so only if **C** has not been modified by some other process since this process made a copy of it. If the attempt fails, the current value of the shared variable is returned. We do not use this last feature in our algorithm.

The A-B-A problem occurs when **C** is the same as **A** even though **C** has been modified a number of times since the process made a copy of it. A *compare&swap* operation performed by the process now will succeed, which can cause errors to occur in many implementations of concurrent objects. To prevent this error a counter is appended to **C** and is always incremented when a modification is made to **C.** This leads to the *compare&swap double* operation (implemented in the Motorola 68000 family [8]), a five-operand atomic instruction of the form **CSDBL(A1, A2, B1, B2, C)** that does the following:

If **A1** $||^2$ **A2** equals **C**
then put **B1** || **B2** into **C**, return condition code 0
else put **C** into **A1** || **A2**, return condition code-1.

Now, the shared variable **C** is twice the size of the other variables since one half of **C** is used as a counter (we shall assume it to be the second half). A process first reads the value of **C** and puts it in **A1** and **A2**. It puts the desired new value of **C** into **B1,** assigns to **B2** the value **A2** +1, and then executes the **CSDBL** instruction. Although the A-B-A problem can still occur, the probability is much lower [11] and is acceptable for a large class of applications.

III. Data Structures for the Queue

The data structure we use for the shared FIFO queue is a singly linked list. An object in the list consists of a pointer to the next object and whatever other structure may be needed for the particular application. The pointer is a double-size variable, that is, one half is the actual pointer and the other half is the counter (to be used in the *compare&swap double* operation). One bit of the counter, however, is used to indicate whether the object is going to be dequeued. We will discuss the use of this bit in Section 4. When the counter is used in this way this bit shall be referred to as **mark** and the remaining bits as **counter.** The head and tail of the queue are pointed to by shared pointers of the same type as those in objects of the list. The counter need not be subdivided further for these variables.

[1] We do not give the exact location of the variables at the time of instruction execution as it is not important here.

Structure Pointer		
	ptr	**Pointer_to_object**
	count	**Integer**
Structure Object		
	data	**Structure_of_data**
	nextobject	**Pointer**
Shared Variables		
	Shared_head	**Pointer**
	Shared_tail	**Pointer**
Private Variables		
	Private_tail	**Pointer**
	Private_head	**Pointer**
	Next	**Pointer**
	objectptr	**Pointer_to_object**

Fig. 1. Data structure of the queue.

The data structures and variables are shown in Fig. 1. An important assumption for the shared queue is that objects cannot be destroyed. A dequeued object must be either returned to a shared pool or kept for later use in enqueuing (the reason for this shall become evident in Section IV). A shared pool, if used, can be maintained as a stack. The implementation of a nonblocking concurrent stack using the *compare&swap* operation has been discussed elsewhere [13].

IV. The Algorithm

Objects are dequeued at the head and enqueued at the tail of the linked list. For the algorithm to be nonblocking it is necessary that

- either the enqueue and dequeue operations are atomic,
- or when they comprise more than a single step, the subsequent steps can be performed by any process.

Our algorithm successfully implements the second option. When the enqueue or dequeue operation consists of two or more steps, the queue goes into a unique intermediate state after the first step of the enqueue or dequeue operation. Any process can now identify and correct the state of the queue thus completing the previously incomplete enqueue or dequeue operation and enabling it to proceed with its own operation.

A. Reading the State of the Queue

The queue can be in one of the eight states shown in Fig. 2.

These states are generated on the basis of steps taken in the enqueue and dequeue operations which are discussed in the following subsections. The state of the queue can be uniquely identified by getting a *simultaneous* picture of the variables **Shared_head, Shared_tail** and the **nextobject** pointer of the object pointed to by **Shared_tail,** storing these values in **Private_head, Private_tail** and **Next.** This can be obtained by the Snapshot procedure shown in Fig. 3. When the procedure succeeds, it gives the simultaneous value of **Shared_head, Shared_tail** and the **nextobject** pointer at the instant the

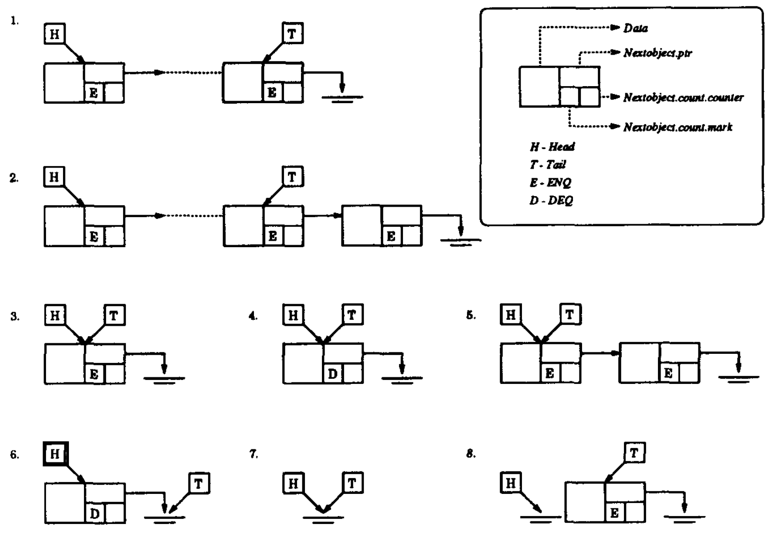


Fig. 2. States of the Queue.

nextobject pointer is last read. This procedure returns to the caller (in the caller's own private variables) the value in each of these variables.

B. The Enqueue Procedure

An enqueuer can proceed with its operation from only three states, namely states 1, 3, and 7 (the states are shown in Fig. 2). If the queue is in some other state, the enqueuer must first bring the queue to one of these states. Two cases can occur when a process enqueues an object starting from one of these "correct" states. The steps taken in each of these cases are described below and the state transitions they cause are shown in Fig. 4.

- When the queue has one or more elements (state 1 or 3) the operations are the following.
  a) (Operation 2E(m):1 in Fig. 4) Append the object to the end of the linked list. This operation will not succeed if a dequeuer has already marked it for deletion, by setting **mark** to DEQ (this case will only occur if there is only a single object in the queue, since only then may enqueuers and dequeuers be attempting to make changes to the same object) or, if another enqueuer has already appended its own object to the end of the linked list.
  b) (2E(m):2) Shift the tail to point to the new end of the list.
- When the queue is empty (state 7) the operations are:
  a) (2E(0):1) Shift the tail to point to the new object;
  b) (2E(0):2) Shift the head to point to it.

If after a process completes the first step another enqueuing or dequeuing process accesses the queue, the second process may do the second operation itself (since it detects an intermediate state). Nonetheless, the first process will try to do

```
Procedure Snapshot(Private_head, Private_tail, Next)
  repeat
        Read Shared_head into firsthead
        repeat
            Read Shared_tail into firsttail
            if firsttail.ptr ≠ NULL then
              Read firsttail.ptr->nextobject into Next
            Read Shared_tail into Private_tail
        until Private_tail = firsttail
        Read Shared_head into Private_head
  until Private_head = firsthead
end procedure Snapshot
```

Fig. 3. The Snapshot procedure.

the second operation and will be unsuccessful. This is due to the fact that we use the *compare&swap* instruction to do the operation, so once the tail (or head, in the second case) has been changed, no process which read the tail (head) before the change can succeed in modifying it. The remaining part of the enqueue procedure is devoted to detecting intermediate states of the queue and correcting them by completing the unfinished enqueue and dequeue operations. The complete enqueue procedure is shown in Fig. 5.

C. The Dequeue Procedure

A dequeuer can proceed with its operation only from states 1, 2, 3, and 7. Just as with enqueuers, if a dequeuer encounters the queue in some other state, it must first bring the queue to one of these states. Two cases arise when an object is to be dequeued.

- If there are two or more objects in the queue (state 1 or 2), dequeuing consists of a single step: shifting the head of the queue to the next object in the list (Operation 1D(m):1 in Fig. 4).
- If there is only a single object in the queue (state 3), it must be first marked, to prevent enqueuers from appending objects to this object and also, to tell other dequeuers that the object has already been claimed. The object is marked for deletion by changing **mark** of

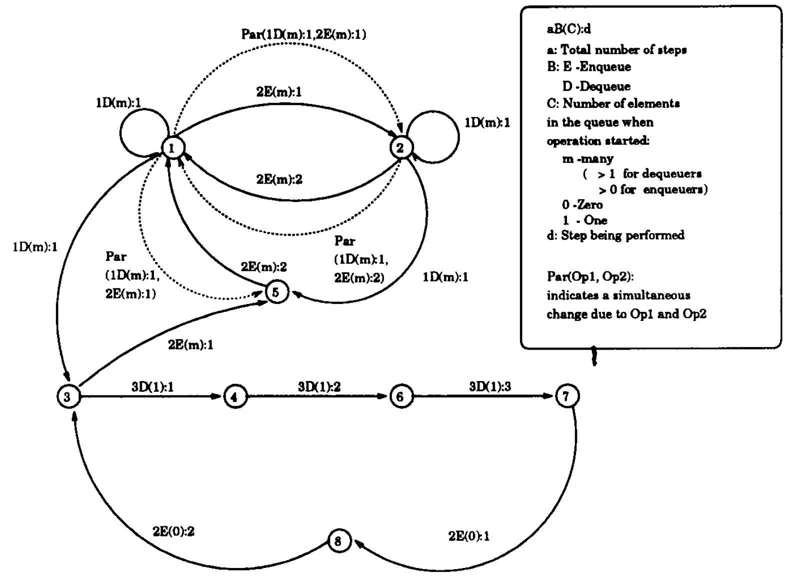


Fig. 4. State Transition Diagram.

the **nextobject** pointer from ENQ to DEQ. If the mark operation succeeds, then the object can be dequeued by shifting the tail and head to NULL (Operations 3D(1):1, 2, and 3, respectively).

In the second case, if other enqueuers and dequeuers access the queue after the first step is complete they will cooperate in completing the dequeue operation before proceeding to do their own operations. If the queue is in state 7, the procedure simply reports an empty queue. Just as in the enqueuing procedure, part of the dequeuing procedure is devoted to correcting the state of the queue.

The complete dequeue procedure is shown in Fig. 6.

D. Proof of Correctness

We show that this algorithm is correct by proving it satisfies certain safety and liveness properties. The safety properties are the following.

- If the head and the tail of the queue are different, serial changes may be made simultaneously at the head and at the tail. If head and tail are the same, all changes occur serially.
- Additions to the queue are done only at the tail of the queue and deletions only at the head.

and the liveness property is:

- if there are enqueue and dequeue attempts being made on the queue, some attempt will succeed in a finite amount of time.

The first safety property ensures that each enqueue (dequeue) operation must be completed before the next enqueue (dequeue) operation is started. Enqueuing and dequeuing may proceed in parallel, however, except for the case in which the head is the same as the tail. Here, the enqueue and dequeue operations are serialized. Since we require enqueuing to be done only at the tail and dequeuing only at the head, it follows that the queue will be accessed in FIFO order.

The liveness property ensures that some enqueuer or dequeuer must succeed in performing its operation in a finite amount of time regardless of the failure or inactivity of other enqueuers and dequeuers. Since the liveness property is unconditional, this algorithm is nonblocking.

The proof of the safety and liveness properties follow from the state transition diagram (Fig. 4). The safety properties are obeyed as follows.

- When the head and the tail of the queue are different, enqueue attempts are serialized by a single decisive operation such as the successful appending of an object to the end of the linked list. (Concurrent actions are *decisive operation serializable* [12] if they are *serializable* [12] and there is an operation $dec(a)$ in each of the actions such that if $dec(a_i)$ occurs before $dec(a_j)$ then a_i comes before a_j in the serial order.) Due to the nature of the *compare&swap* operation, only one enqueuer can succeed in its decisive operation. All other enqueuers then cooperate to complete the enqueue operation. They do so when they detect an intermediate state of the queue which implies an incomplete enqueue operation. Many enqueuers may attempt to correct the state, but only one can succeed. Dequeue attempts are also serialized by a single decisive operation, the shifting of the head to the object after the one being pointed to by the head. Again, it is the *compare&swap* operation which allows only one dequeuer to succeed, because once the head has been shifted, no process which read the head before the change can modify (shift) it.
- When the head and the tail are the same and the queue is not empty, all operations are serialized by the attempt to claim the **nextobject** pointer of the object pointed to by

```
Procedure Enqueue (objectptr)

Initialize object, leave count untouched except for
marking the object to be enqueued

objectptr->nextobject.ptr = NULL
objectptr->nextobject.count.mark = ENQ

success = FALSE

repeat

  take a snapshot of the queue

  Snapshot(Private_head, Private_tail, Next)
  Determine state of queue and assign to State

  case State of:

  1,3: Queue is in a correct state, so enqueue object

    cc = CSDBL (Next.ptr, Next.count, objectptr,
                ENQ || Next.count.counter+1,
                Private_tail.ptr->nextobject)

    if cc = 0 then

      Attempt was successful, so shift the tail to the
      object just added

      CSDBL (Private_tail.ptr, Private_tail.count,
             objectptr, Private_tail.count+1,
             Shared_tail)

      success = TRUE

  2,5: Finish incomplete enqueue operation

    CSDBL (Private_tail.ptr, Private_tail.count,
           Next.ptr, Private_tail.count +1,
           Shared_tail)

  4: Cooperate in dequeuing the object

    Shift the tail to NULL

    CSDBL (Private_tail.ptr, Private_tail.count,
           NULL, Private_tail.count+1,
           Shared_tail)

    Shift the head to NULL

    CSDBL (Private_head.ptr, Private_head.count,
           NULL, Private_head.count+1,
           Shared_head)

  6: Complete dequeuing the object

    CSDBL (Private_head.ptr, Private_head.count,
           NULL, Private_head.count+1,
           Shared_head)

  7: No elements in the queue, do an empty queue
     enqueue by first shifting tail to the object

    cc = CSDBL (Private_tail.ptr, Private_tail.count,
                objectptr, Private_tail.count+1,
                Shared_tail)

    if cc = 0 then

      The object has been added, so shift the head
      to point to it

      CSDBL (Private_head.ptr, Private_head.count,
             objectptr, Private_head.count+1,
             Shared_head)

      success = TRUE

  8: Finish incomplete enqueue operation

    CSDBL (Private_head.ptr, Private_head.count,
           Private_tail.ptr, Private_head.count+1,
           Shared_head)

  end case
until success
end procedure enqueue
```

Fig. 5. The Enqueue Procedure.

the head and the tail (dequeuers do this by changing **mark** and enqueuers by changing the pointer). Again, only one attempt can be successful and all processes cooperate to complete that operation. This happens when the processes detect an intermediate state of the queue which implies an incomplete enqueue or dequeue operation.

- When the queue is empty, enqueue attempts are serialized by the successful shifting of the tail to a process's own object. Again, only one process can be successful in shifting the tail and all processes now attempt to complete the operation with only the first one making the attempt being successful.

The proof of the liveness property follows from the observation of two facts. First, there is no queue state from which an enqueuer or dequeuer cannot take an action. Second, the only way enqueuers and dequeuers can be kept from taking an action is if the state of the queue changes while they are reading or trying to modify the state of the queue. By contradiction, if no one succeeds then the state of the queue cannot change, so some process must succeed. Together, these parts imply that some action must be completed in a finite amount of time.

In this proof we have not taken into consideration the A-B-A problem, which we have assumed to have a very low probability of occurrence. Using one bit of the count in the **nextobject** pointer of an object to mark objects will not significantly affect the chances of occurrence of the A-B-A problem for the queue as a whole. This is because the A-B-A problem is more likely to occur at the head or tail, since a *compare&swap double* operation will be performed on the **nextobject** pointer of an object far less frequently than the head or the tail (as there are many objects in use).

```
Procedure Dequeue(objectptr)

  success = FALSE

  repeat

Snapshot (Private_head, Private_tail, Next)
Determine state of queue and assign it to State

case State of:

1, 2: Do a single step dequeue i.e shift head to the
   next object

  cc = CSDBL(Private_head.ptr, Private_head.count,
             Private_head.ptr->nextobject.ptr,
             Private_head.count+1, Shared_head)

  if cc = 0 then
    objectptr = Private_head
    success = TRUE

3: Single object in queue, so do a three step dequeue

  Mark the object to be dequeued

  cc = CSDBL( Next.ptr, Next.count, Next.ptr,
             DEQ || Next.count.counter+1,
             Private_tail.ptr->nextobject)

  if cc = 0 then
    success = TRUE
    objectptr = Private_head

    Attempt was successful, so make the tail NULL

    CSDBL (Private_tail.ptr, Private_tail.count,
           NULL, Private_tail.count+1,
           Shared_tail)

    Now shift the head to NULL

    CSDBL (Private_head.ptr, Private_head.count,
           NULL, Private_head.count+1,
           Shared_head)

4: Cooperate in dequeuing the object

  Shift the tail to NULL

  CSDBL (Private_tail.ptr, Private_tail.count,
         NULL, Private_tail.count+1,
         Shared_tail)

  Now shift the head to NULL

  CSDBL ( Private_head.ptr, Private_head.count,
          NULL, Private_head.count+1,
          Shared_head)

5: Complete unfinished enqueue operation

  CSDBL (Private_tail.ptr, Private_tail.count,
         Private_tail.ptr->nextobject.ptr,
         Private_tail.count+1, Shared_tail)

6: Complete dequeuing the object

  CSDBL (Private_head.ptr, Private_head.count,
         NULL, Private_head.count+1,
         Shared_head)

7: Report empty queue

  objectptr = NULL

  success = TRUE

8: Finish incomplete enqueue operation

  CSDBL (Private_head.ptr, Private_head.count,
         Private_tail.ptr, Private_head.count+1,
         Shared_head)

  end case
until success

Delink dequeued object from list

if objectptr≠NULL then
  objectptr->nextobject.ptr = NULL

end procedure dequeue
```

Fig. 6. The Dequeue Procedure.

It may seem that the snapshot isn't needed, since all operations are protected by the *compare&swap double* operation. However, an error can result if the snapshot isn't used. Consider the following example: The queue has a single element in it, and it is marked for dequeuing. An enqueuer reads the head and tail of the queue, then blocks. The element is removed, and at some point the DEQ mark is reset. The enqueuer unblocks, reads the element. The queue appears to be in state 3, so the enqueuer attempts to append its element to the dequeued element, and might succeed in appending its element to one not in the queue.

E. Analysis

The algorithm has an $O(n)$ system latency, where n is the number of processes in the system (*system latency* is defined in [3] as the maximum number of steps a system can take without completing an operation). Assuming n enqueuers and dequeuers, a single modification of **Shared_head** or **Shared_tail** by one of them could cause all others to fail in reading the state of the queue and require them to start reading the state all over again. The number of modifications a process can make is a constant, so this sequence of events could be repeated only a constant number of times until a full enqueue or dequeue operation is complete, after which the next operation must succeed.

The number of steps taken for the enqueue and dequeue operations is independent of the queue size, as is evident from the state transition diagram. There are no additional memory requirements for the algorithm (except for some temporary private variables). The only requirement is that objects should not be freed but returned to a shared pool because even after dequeuing there may be some processes reading or attempting to write to the object (the write will fail). As mentioned earlier, this pool can be maintained as a stack. In the event of the failure of an enqueuing or dequeuing process, all that is (possibly) lost is the single object in the process of being enqueued or had been dequeued. The queue is thus fault-tolerant.

As we discuss in the introduction, several authors have written lock-free queue algorithms, but most are not directly comparable because they limit concurrency [7], block processes [2], [11], require infinite space for continued operation [4], or require $O(n)$ time to perform a dequeue, where there are n items in the queue [13]. Several authors have presented transformations with similar complexities. Herlihy's transformation [3] required $O(p^2)$ space overhead for correct execution. Our algorithm requires $O(p)$ space overhead. Prakash and Turek *et al.* [9], [14], [15] present methods for transforming locking algorithms into asymptotically efficient non-blocking algorithms. While these transformations are asymptotically as efficient as our algorithm and are more general, our algorithm is practically implementable.

V. Performance

Our nonblocking queue has some definite advantages over a blocking queue in terms of fault tolerance. The question remains as to whether the nonblocking queue has better performance than a blocking queue. The nonblocking queue lets fast processors perform work for slow processors, but requires processors to restart if the queue is modified. In order to compare the algorithms, we simulated them and developed an analytical model.

We wrote a hypothetical locking algorithm to compare against our nonblocking algorithm. The locking algorithm has the same concurrency as the nonblocking algorithm. There are three locks in the system: an enqueue lock, a dequeue lock, and a common lock. When an enqueuer comes into the system it tests to see if the enqueue lock is set. If so, another enqueuer is active and the blocked enqueuer goes to sleep in a queue at the enqueue lock. If not, it sets the enqueue lock. It then tests to see if there is only one element in the queue (from which elements are to be dequeued and enqueued). If not, it proceeds to do an enqueue. When the enqueuer leaves, it resets the enqueue lock only if there are no enqueuers waiting at the lock, else it wakes the first waiting enqueuer. If there is only a single element in the queue after the enqueuer sets the enqueue lock, it checks to see the common lock and if the lock is not set, it sets the lock and continues enqueuing as before. If the lock is set, then a dequeuer wants that element, so the enqueuer goes to sleep at the enqueue lock, after setting an indicator that all enqueuers are asleep. When the dequeuer leaves, it should wake one of the waiting enqueuers. Whenever an enqueuer leaves, it too checks to see if the indicator that all dequeuers are asleep is set, so that it can wake one as it goes.

A dequeuer proceeds in very much the same way. It first sets the dequeue lock, and then checks to see if there is only a single element in the queue. If so, it checks to see if the enqueue lock is set. If the lock is, enqueuers are in the system, and they get preference, so the dequeuer goes to sleep after setting the all dequeuer asleep indicator. If the enqueue lock was not set, the dequeuer sets the common lock and proceeds to do a dequeue. When it leaves it wakes one of the waiting dequeuers (if there are any), else it resets the dequeue lock. If there was more than one element in the system after the dequeuer set the dequeue lock, it proceeds to do a dequeue without doing any additional checking. If there were no elements in the queue, the dequeuer leaves, and also sends away all dequeuers waiting at the dequeue lock, saying that the queue is empty.

This algorithm also has two streams, an enqueuing stream and a dequeuing stream, which combine to a single stream when there is only a single element in the queue.

A. The Simulator

We wrote a discrete event simulation of the locking and nonblocking algorithms. Enqueue and dequeue operations in both the locking and the nonblocking algorithms required about the same number of instructions. Operations arrived in a Poisson process, and we assumed that operations of a given class always required the same amount of time to complete. We measured the number of times snapshots were missed, and the number of times a process caught the queue in an intermediate state and corrected it (this tells us how much "cooperation" there is among processes). We ran the simulations until 5000 enqueue and dequeue operations completed, both being equally probable.

B. Simulation Results

We found that the locking and nonblocking algorithms show very similar behavior when all processes have the same speed. We next used a mix of processes of two speeds. Here, some processors ran ten times fasters than the others, and 90% of the processors were of the faster variety. Processes take 17 instructions for enqueuing and 13 for dequeuing and one instruction takes a single time unit for fast processes. Since a processor always works at the same rate, we call this model the *constant speed* model. We found that in this model, the blocking queue has lower response times than the nonblocking queue (see Fig. 7), especially as the arrival rate increases. While fast processes have a lower response time in the nonblocking queue (see Fig. 8), this advantage is

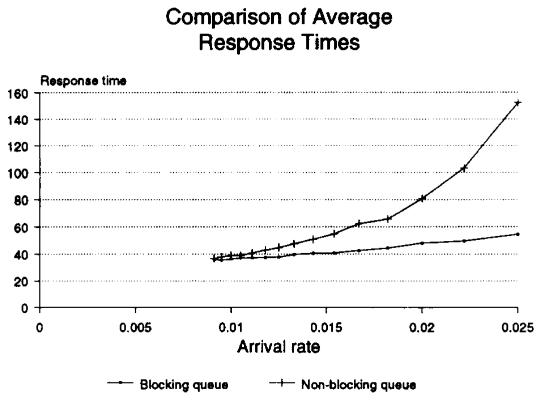


Fig. 7. Simulation results for the constant speed model.

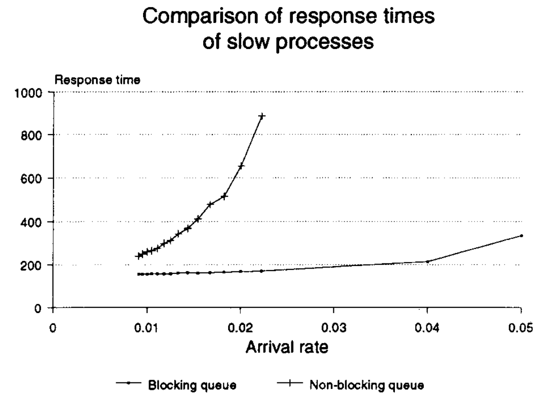


Fig. 9. Simulation results for the constant speed model.

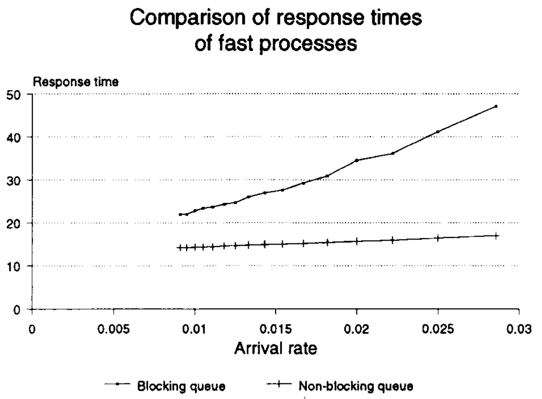


Fig. 8. Simulation results for the constant speed model.

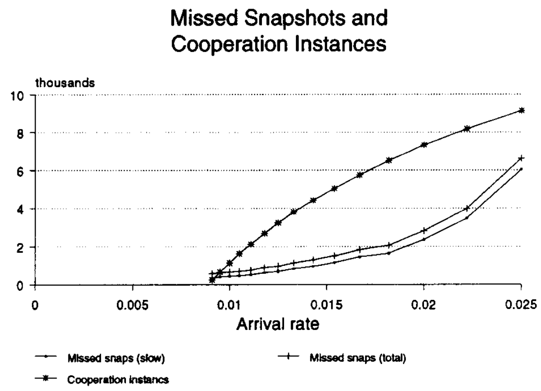


Fig. 10. Simulation results for the constant speed model.

more than offset by the higher response times of the slow processes (see Fig. 9). We find that the snapshots missed by the slow processes increases rapidly (see Fig. 10) with the increasing arrival rate, indicating starvation. The number of times processes cooperate also goes up with arrival rate. This increase is not enough to compensate for the missed snapshots, since the penalty for a missed snapshot is greater than the advantage gained in a single cooperation instance in this algorithm. On the whole, slow processes do not get the chance to commit their operations very often, even after getting valid snapshots, as is indicated by their slow response times. It is this inability of slow processes to complete their work that results in the overall poorer performance of the nonblocking algorithm.

The constant speed model addresses differences in processors, but doesn't account for the possibility that a processor might temporarily become slow due to memory, network, or CPU contention. We also considered the *varying speed* process model, in which we again assume that there are two process speeds, the fast speed being ten times the slow speed. A process is ten times more likely to start out as fast than slow. The amount of time spent at the fast speed is exponentially distributed with the mean as 300 time units. The amount of time spent at the slow speed is exponentially distributed with the mean as 30 time units. As before, one instruction takes a single time unit at the fast speed. In the varying speed model, the nonblocking queue has better performance than the blocking queue (see Fig. 11). The performance of the nonblocking algorithm is better in this situation because of the following.

- When a process is slow, it is very probable that the process will not be able to either get a valid snapshot. Even if the process does get a snapshot, it may not be quick enough to take a decisive action. Thus, the slow process gets delayed, but does not delay others. In the blocking algorithm, since there is no discrimination against slow processes (or processes while they are slow), a process in its slow phase may get a lock and delay all the waiting processes. In the non-blocking algorithms, operations are performed on the queue at the pace of the fast processors.
- Since a process does eventually become fast, it will ultimately be able to perform its operation. This situation is in contrast to the constant speed model, in which slow processes were starved by the fast processes, which led to very poor response times.

We thus see that the performance of the nonblocking algorithm depends on the situation, and is better than a blocking algorithm when we have a system of processes running on

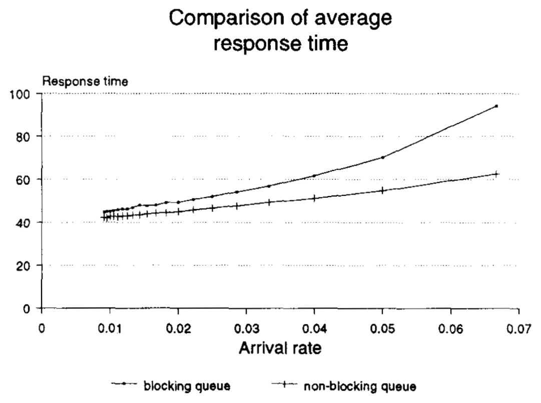


Fig. 11. Simulation results for the varying speed model.

processors which occasionally get slow. It is not better in the case of a mixture of slow and fast processors, because the algorithm favors the faster processes.

C. Analytical Model

To better understand the phenomena that the simulations revealed, we analytically modeled both the blocking and the nonblocking queues. We make the simplifying assumption that the queue never empties, so that the enqueue and the dequeue operations are always independent. In this case, the blocking queue can be modeled as a standard M/G/1 queue. Our nonblocking queue needs different analytical tools, since there is no queuing. The nonblocking algorithm uses methods similar to Optimistic concurrency control [6], since an operation reads the control data, computes a modification, then commits the change only if no other operation has modified the data. We make use of the analytical framework provided by Ryu and Thomasian [10].

Constant Speed Model Ryu and Thomasian model a closed transaction processing system in which V transactions each execute one of C transaction types. When a new transaction enters the system, it is a class c transaction with probability f_c. If a transaction is aborted, it restarts as the same class transaction, so that when a transaction commits, it is a class c transaction with probability f_c, because of conservation of flow. Thus, we can apply Ryu and Thomasian's techniques to the constant speed model.

A class c transaction is assumed to have an execution time of $\beta(V)b_c(x)$, where $b_c(x)$ is the probability density function of the execution time of a class c transaction, and $\beta(V)$ is the increase in execution time due to resource contention. The expected residence time (time required to execute the transaction if there is no abort) is denoted by $R_a^c(V) = \beta(V)b_c$, where b_c is the expected value of b_c. A transaction might be required to restart several times due to data conflicts. The expected time that a transaction spends executing aborted attempts is denoted by $R_d^c(V)$, and the total residence time of a class c transaction is $R^c(V) = R_a^c(V) + R_d^c(V)$. The *efficiency* of a class is the proportion of its expected residence time spent executing a transaction that commits: $U_c = R_A^c(V)/(R_A^c(V) + R_A^d(V))$. The expected residence time a transaction in the system is calculated by taking the expectation of the per-class expected residence times. The system efficiency is calculated by

$$U(V) = R_a(V) \Big/ \left(\sum_{c=1}^{C} f_c R_a^c(V)/U_c(V) \right) \tag{1}$$

In order to calculate the per-class efficiencies, we need to calculate the probability that a transaction aborts due to a data conflict. If Φ_c is the probability that a transaction conflicts with a class c transaction when it commits, then other transactions conflict with a class c transaction at rate

$$\gamma_c = \frac{(V-1)\Phi_c}{b} U(V)$$

where b is the expected execution time of all transactions. If the execution time of a class c transaction is fixed, its residence time has a geometric distribution, based on the number of restarts until it can commit. Ryu and Thomasian assume that the process by which other transactions conflict with a class c transaction can be accurately modeled as a Poisson process.

In our model, a queue operation reads the state variables, computes the necessary changes, then performs the decisive operation to commit the changes. When an operation attempts to commit its change, it fails if another operation has already changed the data structure. We call the interval between the time that the operation first reads a control variable to the time that it attempts to commit its change the execution time of the operation. We assume that the queue never empties, and that we are modeling only the enqueuers (or only the dequeuers), so that every operation conflicts with every other and $\Phi = 1$. The probability of conflict between any two operations is the same, so their conflict rates are the same, and we will denote the conflict rate by γ.

In the simulations, we modeled fast and slow operations. Correspondingly, we use fast and slow transaction types in our analytic models. We define f_s to be the probability that a transaction is a slow transaction (and $1 - f_s$ is the probability that a transaction is fast).

In [10], if a transaction executes for t seconds, then aborts, it will execute for t seconds when it restarts. Our system is better described by assuming that the time required to execute the restarted operation is independent of the time to execute the first transaction, and is a sample from the same distribution.

If an operation requires t seconds, the probability that it will be commit is $e^{-\gamma t}$, since we assume that conflicts form a Poisson process. Therefore, the unconditional probability that the operation commits is:

$$p_c = \int_{t=0}^{\infty} e^{-\gamma t} b_c(t)\,dt. \tag{2}$$

The number of times that the operation executes has a geometric distribution, so an operation will execute $1/p_c$ times. The first $1/p_c - 1$ times the operation executes, it will be unsuccessful. Knowing that the operation is unsuccessful tells us that it probably required somewhat longer than average to execute, since slow operations are more likely to be aborted. Similarly, successful operations are likely to be faster. The distributions of the execution times of the successful and unsuccessful operations is given by

$$b_c^s(t) = K_s e^{-\gamma t} b_c(t)$$
$$b_c^f(t) = K_f(1 - e^{-\gamma t}) b_c(t),$$

where K_s and K_f are normalizing constants computed by

$$K_s = 1 \Big/ \left(\int_0^\infty e^{-\gamma t} b_c(t) dt \right)$$
$$K_f = 1 \Big/ \left(\int_0^\infty (1 - e^{-\gamma t}) b_c(t) dt \right)$$

If b_c^s and b_c^f are the expected values of $b_c^s(t)$ and $b_c^f(t)$, the expected time to complete a class c operation is $b_c^s + (1/p - 1)b_c^f$. Therefore, we find that

$$U_c = \frac{b_c}{(b_c^s + (1/p - 1)b_c^f)}. \tag{3}$$

Equation 1 for the system efficiency depends on the per-class efficiencies, and equation 3 for the per-class efficiency depends on the system efficiency. Ryu and Thomasian show that this system can be rapidly solved by iteration.

If we assume that the operations have an exponentially distributed execution time, $b_c(t) = \mu_c e^{-\mu_c t}$, then the system is simple enough to solve algebraically. If the rate at which a fast transaction commits is μ_f and the rate at which a slow transaction commits is μ_s, then:

$$U_f = \frac{(1 - U_s \mu_s) f_f \mu_s U_s}{\mu_f(\mu_s U_s f_s - f_s - U_s V + U_s)},$$

where U_s is the solution in $[0, 1]$ of

$$0 = (\mu_s^3 f_s + \mu_s^2(f_f \mu_f + 1 - V)U_s + ((\mu_f - \mu_s)(1 - V)$$
$$+ \mu_f \mu_s(2f_s - 1) - 2f_s \mu_s^2)U_s + f_s \mu_s - f_s \mu_s.$$

If the operations have deterministic execution times (the case in the simulation models), the assumption of Poisson conflict arrivals is a poor approximation to the actual conflict arrival process. We use a different model to handle deterministic execution times.

Variable Speed Model The simulations showed that the nonblocking queue has markedly different performance depending on whether slow processors stay slow, or whether they are occasionally slow, then become fast. Ryu and Thomasian's model can be directly applied to the case in which there is a mixture of fast and slow processors in the system, since a transaction stays in its class until it commits. However, we need another model for the case when a processor is usually fast, but occasionally becomes slow. We will assume that the processor executing an operation is either fast or slow, but it can change its speed after trying to commit the operation. In this model, f_s' is the probability that any given transaction is slow (and $1 - f_s'$ is the probability that any given transaction is fast). In Ryu and Thomasian's model, every transaction eventually commits as a transaction of its initial class. Now, it isn't necessarily the case that every slow transaction commits as a slow transaction. We can calculate f_s' to be $f_s b_s / b$. In this model, we will solve for the probability that an operation commits, p_s and p_f.

Since f_s' and $f_f' = 1 - f_s$ are the proportion of fast and slow operations, we need to redefine the conflict rate to be

$$\gamma = (V - 1)(f_s' p_s / b_s + f_f' p_f / b_f)$$
$$= \frac{V - 1}{b}(f_s p_s + f_f p_f).$$

Given a conflict rate γ, we calculate p_c using formulae (2). If b_c has an exponential distribution, then we can calculate p_c to be $\mu_c/(\gamma + \mu_c)$. We have two equations in two variables, which we can solve to find that

$$p_f = \frac{b - V b_s p_s f_s - p_s b}{p_s V b_s f_f} \tag{4}$$

and p_s is the solution in $[0, 1]$ of

$$\left(\frac{V b_f f_s b_s - b_s^2 V f_s}{b} \right) p_s^3$$
$$+ \left(\frac{b_f b - b_s b - V b_s^2 f_f - V b_f b_s f_s}{b} \right) p_s^2$$
$$+ (b_s - 2b_f)p_s + b_f = 0. \tag{5}$$

Deterministic Execution Times The Ryu-Thomasian model can't be applied to the case of deterministic execution times, because the conflict process becomes significantly different than a Poisson process. We provide here a simple model for deterministic execution times.

Under the constant speed execution model, a fast processor will always commit before the slow processors. As a result, slow processors tend to accumulate, so the processor population is either all slow processors, or one fast and all but one slow processor. If a processor commits and is replaced by another fast processor, no time is lost. If a fast processor commits and is replaced by a slow processor, then the first processor to abort will be the one that commits. The time between the commit of the fast processor and the first abort of a slow processor is the wasted time. If we model the commit of the fast processor as occurring uniformly randomly within the slow processor's execution time, then the wasted time is the minimum of $N - 1$ samples of a uniform $[0, b_s]$ random variable, which has mean b_s/N. So the throughput of nonblocking queue under the constant speed processor model with deterministic execution times is:

$$T_{\text{const},D} = \frac{1}{f_f b_f + f_s b_f (N + 1)/N}.$$

In the varying-speed processor model, a fast processor (almost) commits, so the throughput is

$$T_{vary,D} = \frac{N}{f_f b_f (N + 1)}.$$

D. Comparison

We can compare the performance of the blocking and the nonblocking queues by comparing their throughput in closed systems with varying numbers of concurrent operations. The throughput of the blocking queue is $1/(f_s/\mu_s + f_f/\mu_f)$, independent of the number of concurrent operations.

Fig. 12 shows a comparison between throughput of the blocking and the nonblocking queue when the execution times

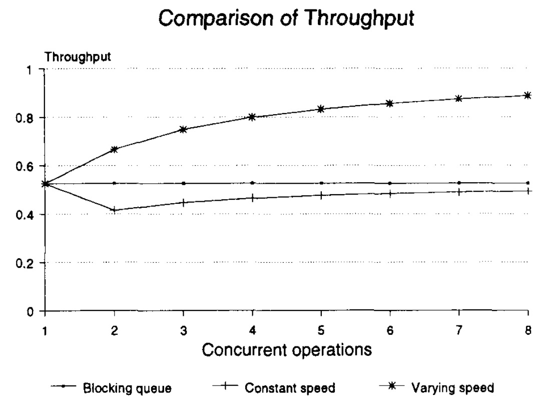


Fig. 12. Analytical results for deterministic execution times.

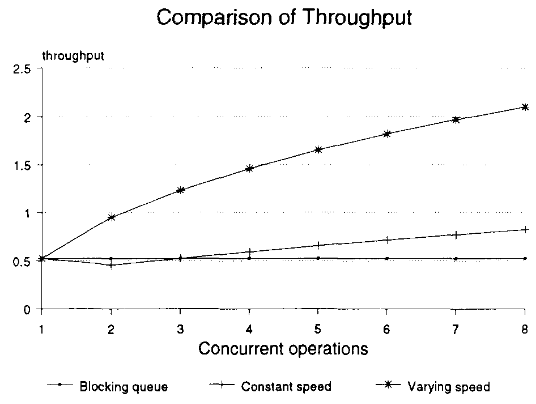


Fig. 13. Analytical results for exponential execution times.

are deterministic. The throughput of both the constant and the varying speed models is shown. The input parameters are similar to those used in the simulation experiments: Fast operations required an expected 1 second to finish, and constituted 90% of the operations. Slow operations required 10 seconds to finish. As expected, the constant speed nonblocking queue has a lower throughput than the blocking queue, which has a lower throughput than the varying-speed nonblocking queue.

We also performed calculations assuming exponentially distributed execution times. Fig. 13 shows a comparison of the throughput of the locking queue and of the nonblocking queue under both processor models. Under the constant speed model, the throughput falls below that of the locking queue for two and three concurrent operations. This drop in throughput reflects the time wasted by invalidated operations. However, throughput increases with increasing concurrency. This is due to the preference shown to fast operations. The average execution time of the successful operations for both nonblocking queue processor models is shown in Fig. 14. Under the varying speed model, the nonblocking queue has a higher throughput than the blocking queue. Again, this difference reflects the preference shown to fast operations by the nonblocking queue.

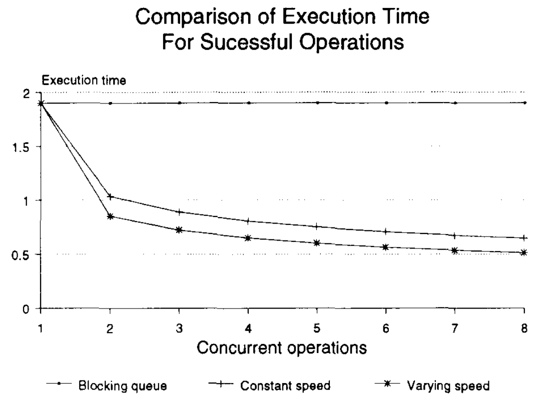


Fig. 14. Analytical results for exponential execution times.

VI. Conclusion

The algorithm we have presented is a simple nonblocking concurrent implementation of a FIFO queue. Modifications to the queue are not always made atomically, as might be expected in a nonblocking implementation. The cooperation between the processes compensates for the non-atomicity and ensures the nonblocking property.

We model the nonblocking queue both analytically and with simulations. Our results show that if a processor takes a varying amount of time to complete an operation (due, perhaps, to network or memory contention), then the nonblocking queue will have better performance because the fastest operations are favored. If, on the other hand, some processors are always fast and some always slow, then the slow processors are starved and the result is a less efficient data structure. If the underlying architecture provides uniform memory access (UMA), then the nonblocking queue will provide better performance than the blocking queue. If the underlying architecture provides non-uniform memory access (NUMA), then processors that are distant from the queue will be starved and the nonblocking queue will be worse than the blocking queue. The alternative in a NUMA architecture is a wait-free implementation [14], but wait-free data structures require significant overhead.

References

[1] "TC2000 Programming Handbook," BBN Advanced Computers, Inc.
[2] A. Gottlieb, B. D. Lubachevsky, and L. Rudolph, "Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors," *ACM Trans. Programming Languages Syst.*, vol. 5, no. 2, pp. 164–189, Apr. 1993.
[3] M. Herlihy, "A methodology for implementing highly concurrent data structures," in *Proc. 2nd ACM SIGPLAN on Principles and Practice of Parallel Programming*, Mar. 1989, pp. 197–206.
[4] M. Herlihy and J. Wing, "Axioms for concurrent objects," in *14th ACM Symp. Principles of Programming Languages*, Jan. 1987, pp. 13–26.
[5] M. P. Herlihy, "Impossibility and universality results for wait-free synchronization," in *Seventh ACM SIGACT-SIGOPS Symp. Principles of Distributed Computing*, Aug. 1988, pp. 276–290.
[6] H. T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Trans. Database Syst.*, vol. 6, no. 2, pp. 213–226, 1981.
[7] L. Lamport, "Specifying concurrent program modules," *ACM Trans. Programming Languages Syst.*, vol. 5, no. 2, pp. 190–222, Apr. 1983.
[8] "M68000 Family Programmer's Reference Manual," Motorola.

[9] S. Prakash, "Non-blocking algorithms for concurrent data structures," Masters thesis, Univ. of Florida, Dept. of Computer Sci., 1991. Available at anonymous ftp site cis.ufl.edu:cis/tech-reports/tr91/tr91-002.ps.Z.

[10] I. K. Ryu and A. Thomasian, "Performance analysis of centralized databased with optimistic concurrency control," Performance Evaluation 7, pp. 195–211, 1987.

[11] J. M. Stone, "A simple and correct shared-queue algorithm using compare-and-swap," in *Proc. IEEE Comput. Society and ACM SIGARCH Supercomputing '90 Conf.*, Nov. 1990, pp. 495–505.

[12] D. Shasha and N. Goodman, "Concurrent search structure algorithms," *ACM Trans. Database Syst.*, vol. 13, no. 1, pp. 53–90, Mar, 1988.

[13] R. Kent Treiber, "Systems programming: Coping with parallelism," RJ 5118, IBM Almaden Res. Ctr., Apr. 1986).

[14] J. Turek, "Resilient computation in the presence of slowdowns," Ph.D. Thesis, Dept. of Comput. Sci., NYU, 1991.

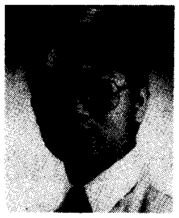
[15] J. Turek, D. Shasha, and S. Prakash, "Locking without blocking: Making lock based concurrent data structure algorithms nonblocking," in *ACM Symp. Principles of Database Syst.*, 1992, pp. 212–222.

[16] IBM T. J. Watson Res. Ctr., System/370 Principles of Operations, 1983, pp. 7.13, 14.

[17] C.-Q. Zhu and P.-C. Yew, "A synchronization scheme and its applications for large multiprocessor systems," in *Proc. 4th Int. Conf. Distrib. Computing Syst.*, May 1984, pp. 486–493.

Sundeep Prakash received the B.Tech degree in electrical engineering from the Indian Institute of Technology, Delhi, India, in 1989, and the M.S. degree in computer science from the University of Florida, Gainesville, in 1991.

Since September 1991, he has been working towards the Ph.D. degree in computer science at UCLA. His research interests lie in the area of parallel and distributed computing.



Yann-Hang Lee received the B.S. degree in engineering science and the M.S. degree in electrical engineering from the National Cheng Jung University, in 1973 adn 1978, respectively, and the Ph.D. degree in computer, information, and control engineering from the University of Michigan, Ann Arbor, MI in 1984.

From December 1984 to August 1988, he was a Research Staff Member at the Architecture Analysis and Design Group, IBM T. J. Watson Research Center, Yorktown Heights, NY. Since August 1988, he has been an Associate Professor with the Computer and Information Sciences Department, University of Florida, Gainesville. His current research interests include distributed computing and parallel processing, real-time systems, computer architecture, performance evaluationm, and fault-tolerant systems.

Dr. Lee is a member of the Association for Computing Machinery and the IEEE Computer Society.

Theodore Johnson received the degree in mathematics from the Johns Hopkins University in 1986, and the Ph.D. degree in computer science from the Courant Institute of New York University in 1990.

Since 1990, he has been an Assistant Professor at the University of Florida, Gainesville. His research interests include the study of parallel, distributed, and concurrent processing, performance modeling, random data structures, and real-time systems.

Dr. Johnson is a member of the Association for Computing Machinery and the IEEE Computer Society.