Concurrent Queues: Practical Fetch-and-Φ Algorithms

John M. Mellor-Crummey

Department of Computer Science
The University of Rochester
Computer Science Department
Rochester, NY 14627

Technical Report 229

November 1987

Abstract

With the growing use of multiprocessors, data structures that support concurrent operations have become increasingly important. In particular, algorithms and data structures for efficient implementation of concurrent queues have generated interest since operating systems heavily use queueing structures. Existing algorithms for managing concurrent queues all have drawbacks for practical implementation and use. This paper presents two practical implementations of a concurrent queue of unbounded length using fetch-and-Φ primitives (a class of atomic read-modify-write operations) and argues their correctness. The first implementation provides *wait-free* enqueues with unbounded concurrency, although dequeues possess an inherently sequential component. The second implementation provides parallelism among a bounded number of enqueue and dequeue operations. This implementation provides greater parallelism than existing algorithms and allows the user to trade space for potential parallelism. Both implementations satisfy the strong correctness criterion of linearizability whereas existing algorithms with similar properties do not.

This research was supported by U.S. Army Engineering Topographic Laboratories research contract no. DACA76-85-C-0001 and NSF research grant no. CCR-8704492.

1 Introduction

With the growing use of multiprocessors, data structures that permit concurrent operations have become increasingly important. In particular, algorithms and data structures for efficient implementation of concurrent queues have generated interest since operating systems heavily use queuing structures.

An important correctness criterion for an implementation of a concurrent data structure is that of *linearizability* [6]. The real-time relationships among a set of operations on a concurrent data structure (hereafter referred to as an *operation history* for the data structure) define a partial order. For each pair (x,y) of non-overlapping operations in the set where x precedes y in the execution, the partial order includes a constraint that orders x before y; concurrent operations have no ordering constraints between them. Operations on a concurrent data structure are said to be linearizable *iff* the partial order of operations in any possible operation history for the data structure can be extended to a total order that corresponds to a legal sequential history for that data structure. In other words, a linearizable operation history has the same effect on the data structure as some sequence of non-overlapping operations. Satisfying the linearizability criterion insures that introducing concurrency in an implementation of operations on a data structure does not alter its semantics.

Constructing an implementation of a concurrent queue that permits only linearizable operation histories is difficult. Without direct hardware support, enqueue and dequeue operations on the queue are not atomic; rather, they are implemented as a sequence of primitive atomic operations supported by an underlying machine. Operations on the queue must coordinate using the primitives of the underlying machine to insure that all operation histories for the data structure are linearizable.

Fetch-and-Φ primitives, proposed by Gottlieb and Kruskal [5], are a class of atomic read-modify-write operations that have gained acceptance as an expressive set of primitives for coordinating parallel computation. Φ denotes a binary associative atomic operation that is applied to the value of a location, and possibly a set of additional values, given as arguments to the fetch-and-Φ operation. Examples of fetch-and-Φ primitives include *fetch-and-add* and *fetch-and-store*. This class also includes operations such as *atomic-write* and *atomic-read* which are degenerate cases of *fetch-and-store*. Several architectures [1], [4] provide fetch-and-Φ primitives for coordination of parallel processes.

There are several existing algorithms for managing concurrent queues, but each has drawbacks for efficient, practical implementation and use.

Lamport [8] presents an example of a simple concurrent queue of bounded length that permits only a single enqueue and dequeue to be simultaneously active at any time; however, the implementation permits full concurrency between those two operations.

Dimitrovsky [2] presents an implementation of a concurrent queue using fetch-and-Φ primitives that collects enqueues and dequeues into separate phases. All operations within a phase execute concurrently. The primary limitation is that the queue may hold only a bounded number of elements. This implementation limits the number of concurrent operations on the queue to the size of the queue.

Herlihy and Wing [6] present a design for a concurrent queue using *fetch-and-add*, *atomic-write*, and *fetch-and-store* primitives. Their implementation provides *wait-free* [6] enqueues.[1] A dequeue, however, may take arbitrarily long: it continues scanning the queue until it finds an item. The main drawback to their solution is that it requires a dedicated buffer of unbounded length to store the queue. Also, augmenting the semantics of their dequeue operation to recognize when it has been invoked on an empty queue calls for non-trivial changes to their algorithm.

Practical implementations of queues of unbounded length use some form of a linked-list data structure. Rudolph [9] presents a design for an unbounded concurrent queue using fetch-and-Φ primitives which represents a single queue as an array of linked lists. *Fetch-and-add* operations on indices direct enqueue and dequeue operations to the next appropriate list. Each list is protected by a critical section which forces operations on that list to be sequential. The maximum concurrency in this design is bounded by the number of lists. Rudolph's implementation fails to satisfy the linearizability correctness criterion as it admits operation histories that are non-linearizable and therefore by our standard, are not correct. Appendix B addresses this point in greater depth.

This paper presents practical algorithms using fetch-and-Φ primitives to implement enqueue and dequeue operations for an unbounded concurrent queue and argues that they satisfy the linearizability correctness criterion providing first-in-first-out (FIFO) semantics. A limitation of the algorithms presented in this paper is that they do not tolerate failure of any operation. Section 2 presents algorithms for concurrent enqueue and dequeue operations for a simple linked-list queue. These algorithms provide wait-free enqueue operations with potential for unbounded concurrency, but dequeues have an inherently sequential component. Section 3 extends these algorithms to manipulate a queue implemented as an array of linked lists. This solution balances potential parallelism between enqueue and dequeue operations better than the first solution. The maximum number of concurrent operations in this design is twice the number of linked-lists that form the queue. The number of lists can be fixed at arbitrarily high level trading space for potential parallelism. Section 4 offers some insight into designing algorithms that provide only linearizable operation histories on a concurrent data structure and some implications the algorithms in this paper have for the design of multiprocessors.

2 A Simple Concurrent Queue

This section presents enqueue and dequeue algorithms for a simple linked-list implementation of an unbounded FIFO queue and argues their correctness. These algorithms rely on several built-in atomic primitives: *atomic-read*, *atomic-write*, *fetch-and-store*, *fetch-and-add*, and *compare-and-swap* (the semantics of these operations are given in appendix A). Each of the assumed primitives belongs to the class of fetch-and-Φ primitives described by Gottlieb and Kruskal [5], [3].

[1] An operation is said to be *wait-free* if it takes only a finite number of instructions to complete, regardless of the execution speeds of other processes.

A linked-list queue implementation permits a potentially unbounded number of elements to simultaneously reside in the queue.[2] However, linked structures suffer from the disadvantage that removing an element requires indirection through a pointer followed by a write to that pointer; this process is inherently sequential. Thus, there must be a sequential component in a dequeue operation on a linked queue.

In our implementation, the queue data structure is represented by two pointers: *L.head*, a pointer to the first element in the list, and *L.tail*, a pointer to the last element in the list. The list is initialized by calling initList, which sets *L.head* and *L.tail* to `nil`.

```
type
    element = record
        --pointer to the next element on the list
        next:  pointer to element;
        ...; --the user defined portion of the element structure
    end;

    --data structure for a FIFO list
    list = record
        --pointers to the head and tail elements in the list, respectively
        head, tail:  pointer to element;
    end;

--initialize the list data structure
procedure initList(var L: list);
begin
    L.head := nil; L.tail := nil;
end;

--enqueue an element on a FIFO list
procedure enqueue(var L: list; var e:  element);
var last:  pointer to element;
begin
    e.next = nil; --new element will be last on list
    --replace the tail of the list with e, returning the previous tail
    last := fetch-and-store(L.tail,e);
    if last = nil then --list was previously empty: update the head pointer
        atomic-write(L.head,e);
    else
        atomic-write(last^.next,e); --link e behind last
    end;
end;
```

[2] Actually, a bound is present in this implementation as well, but it is effectively the cardinality of the set of possible addresses that a pointer may assume. In a practical sense, this is the maximum realizable queue size.

The enqueue operation takes two arguments: L, a reference to a list, and e, a reference to an element. To enqueue e on L, the enqueue routine sets e's next field to `nil`, preparing e to become the tail of the list. The *fetch-and-store* on *L.tail* returns the previous value of *L.tail* (the current last element on the list) and replaces it with a pointer to e. This operation establishes e as the tail of the list and provides a pointer to *last*, the previous tail of the list. If *last* is `nil`, then the queue was previously empty. In this case, the enqueue operation updates *L.head* to point to e, the first (and only) element in the queue. Otherwise, if *last* is non-`nil`, the enqueue completes by linking e after *last*.[3]

A dequeue of an element from a list occurs in two stages. The first stage removes an element from the head of the list. The second stage handles the special case in which the element removed was the only element in a singleton list.

```
--dequeue an element from a FIFO list, return nil if list is empty
procedure dequeue(var L: list):  pointer to element;
var firstEl, secondEl:  pointer to element;
begin
    loop --until an element is found or the list is empty
        --get the element at the head and lock the head with nil
        firstEl := fetch-and-store(L.head,nil);
        if not (firstEl = nil) then exit; --got an element
        elsif atomic-read(L.tail) = nil then
            return nil; --the list is empty
        end;
        --a concurrent operation has made the list state inconsistent, try again
    end;
    --get a pointer to the next element in the list
    secondEl := atomic-read(firstEl^.next);

    if secondEl = nil then
        --firstEl was the only element in the list, need to update L.tail
        if compare-and-swap(L.tail,firstEl,nil) = FALSE then
            --firstEl is no longer at the tail: a concurrent enqueue is detected
            --loop until the concurrent enqueue completes
            repeat until not (firstEl^.next = nil);
            atomic-write(L.head,firstEl^.next); --update head pointer
        end;
    else
        atomic-write(L.head,secondEl); --establish secondEl as the head of the list
    end;
    return firstEl; --return the element dequeued
end;
```

[3]By using the address of the list data structure L as the value `nil` and requiring that *head* be at the same offset in L as *next* is in an element, the body of the enqueue operation simplifies to: `e.next := nil; atomic-write(fetch-and-store(L.tail,e)^.next,e)`.

The initial loop in dequeue terminates either by successfully removing an element from the head of the list, or by returning `nil` if the list is empty. To provide linearizability for the queue, a dequeue must never return `nil` when invoked on a queue that is non-empty for the duration of the dequeue operation. For now, assume the queue invariant that *L.tail* is non-`nil` if the queue is non-empty.

While *L.tail* is non-`nil` (and thus, the queue is non-empty), a dequeue polls *L.head* with a *fetch-and-store* until an element is found. This use of *fetch-and-store* prevents two dequeue operations from seeing the same element at the head of the queue. Upon exit from the loop, *firstEl* points to the first element in the list, and *L.head* is `nil`. The successor of *firstEl* in the list is assigned to *secondEl*.

After removing *firstEl* from the front of the list, it is necessary to test if *firstEl* was the only element on the list. If *secondEl* is not `nil`, then its value is copied into *L.head* establishing *firstEl*'s successor as the head of the list and the dequeue is complete. Otherwise, if *secondEl* is `nil`, the tail pointer may need to be updated. In this case, *firstEl* was the only element on the list at the time *secondEl* was assigned the value of *firstEl*'s next pointer. This may no longer be true as a concurrent enqueue operation may have linked a new element at the end of the list. The protocol that follows prevents a concurrent enqueue from linking a new element behind *firstEl* unobserved by the dequeue. A *compare-and-swap* with *L.tail* is performed, replacing its value with `nil` if it still points to *firstEl*. If this *compare-and-swap* succeeds, then no concurrent enqueue operation has a pointer to *firstEl*, the list has been marked empty, and the dequeue can safely complete.[4] Otherwise, failure of the *compare-and-swap* indicates the presence of a concurrent enqueue operation that may still have a pointer to *firstEl*. In this case, the dequeue busy-waits until the enqueue has updated the *next* field of *firstEl*. When the update is complete, the dequeue copies the value of *firstEl*'s *next* pointer into *L.head*, establishing the new element as the first in the queue, and returns successfully.

In the description of the dequeue operation, we assumed above that the queue possesses the following property: *L.tail* is `nil` *iff* the queue is empty. This property is shown to be invariant by examining the cases in which *L.tail* is written.

1. *L.tail* and *L.head* are initialized to `nil` by initList. The list is initially empty and the property holds.

2. As its first operation on the queue data structure, each enqueue updates *L.tail* by replacing its value with a non-`nil` value using a *fetch-and-store*. Thus, the property is preserved by enqueue operations.

3. A dequeue operation may set *L.tail* to `nil` *iff* the element dequeued was the final one in the queue and no enqueue operations have begun insertion (by executing a *fetch-and-store* on *L.tail*) since *firstEl* was entered into the queue. Thus, a dequeue will not set *L.tail* to `nil` unless the queue is really empty maintaining the property.

[4]Since enqueues access the tail pointer only through *fetch-and-store*, if the value of *firstEl* is still in *L.tail*, then no enqueue has executed a fetch-and-store after the one that enqueued *firstEl*. Thus, *firstEl* is the last element on the list and *L.tail* is correctly set to `nil` by the *compare-and-swap*.

Since the property initially holds and none of the updates to *L.tail* cause it to be violated, we may conclude that the property is invariant for the queue. A consequence of this invariant is that *L.tail* = `nil` implies *L.head* = `nil` since *L.tail* = *L.head* = `nil` when the queue is empty.

2.1 Restrictions

Elements are passed by reference as arguments to the enqueue routine. If an element already linked in a queue is passed by as an argument to the enqueue routine, the queue structure can be disrupted since the element's *next* pointer will be overwritten. This may cause a set of elements to dissappear from the queue. Therefore, an element that is already linked in the queue may not be provided as an argument to the enqueue routine. Furthermore, the element provided to the enqueue routine must be non-*nil*. Code that prevents misuse of the element argument can be added easily to the enqueue routine; however, it was neglected here to simplify the presentation.

2.2 Properties

To complete the description of the enqueue and dequeue routines presented above, we present arguments to show that they correctly implement a linearizable FIFO queue. First, we show sequential correctness. Specifically:

- An enqueue correctly adds an element at the rear of the queue.
- A dequeue correctly removes an element from the front of the queue.

Next, we argue interference freedom for the queue operations. Specifically:

- Concurrent enqueues do not interfere with one another.
- Concurrent dequeues do not interfere with one another.
- Concurrent enqueues and dequeues do not interfere with one another.

We conclude by showing that operation histories on the queue are linearizable by demonstrating a mapping that transforms an operation history into a linear order, and showing that this order corresponds to a legal sequential history for a queue that maintains the FIFO property.

Sequential Correctness

- An enqueue correctly adds an element at the rear of the queue.

  The *next* field of a new element e is set to `nil`, preparing e to be the tail of the list. A *fetch-and-store* on *L.tail* returns *last*, the element formerly at the tail of the list, and establishes e as the new tail of the list. If *last* is `nil`, then the list is empty and e is established at the head of the list; otherwise, e is linked behind *last*.

- A dequeue correctly removes an element from the front of the queue.

  If the queue is empty ($L.tail = L.head =$ `nil`) then *firstEl* gets `nil` from $L.head$ (leaving $L.head$ unchanged) and the dequeue returns `nil`.

  For a non-empty queue (neither $L.tail$ nor $L.head$ is `nil`), *firstEl* gets a pointer to the first element in the list from $L.head$ and *secondEl* gets a pointer to *firstEl*'s successor. If *secondEl* is `nil`, then $L.tail$ is set to `nil` if it still points to *firstEl* (in the absence of a concurrent enqueue, this always succeeds); $L.head$ is already `nil` from the *fetch-and-store* that returned *firstEl*; and the dequeue returns *firstEl* leaving the queue empty. Otherwise, if *secondEl* is non-`nil` its value is written into $L.head$ to re-establish $L.head$ as a pointer to the head of the list.

Interference Freedom

Consider each enqueue or dequeue operation *active* during the closed temporal interval defined by execution of its first and last atomic operations on the shared list structure. For example, an enqueue is active during the interval beginning with its *fetch-and-store* and ending with its *atomic-write*. We refer to two queue operations as *concurrent* if their active intervals overlap.

Above we showed that enqueues and dequeues perform properly if executed in mutual exclusion. Here, we show that enqueues and dequeues perform correctly even in the presence of concurrent operations. Specifically, we show that concurrent operations do not interfere with one another. To understand the arguments for interference freedom presented below, it is important to keep in mind that concurrent operations cannot directly affect the state of each other's local variables; they can only indirectly affect one another by modifying the shared list structure.

- Concurrent enqueues do not interfere with one another.

  Using a *fetch-and-store* on $L.tail$ insures that each enqueue never issues a *dead write* to $L.tail$.[5] Since the *fetch-and-store* atomically reads and overwrites the value of $L.tail$, it insures that each value of $L.tail$ is seen by at most one enqueue; thus, concurrent enqueues each obtain a different pointer from $L.tail$. Since each enqueue gets a different pointer from $L.tail$, no two concurrent enqueues write to the same location with their *atomic-write* statement. Therefore, since the atomic operations by concurrent enqueues on the shared list structure do not interfere, concurrent enqueues do not interfere with one another.

- Concurrent dequeues do not interfere with one another.

  A dequeue D claims an element at the head of the queue by atomically obtaining its pointer from $L.head$ and overwriting $L.head$ with `nil` so no other dequeue sees the same element. If *firstEl*, the value from $L.head$, and $L.tail$ are `nil`, then the list is empty and the dequeue returns `nil` leaving the list unchanged. If *firstEl* is non-`nil`, then the dequeue exits its *loop* statement and continues.

[5] A *dead write* to a variable is one that is followed by another *write* to that variable without an intervening *read*.

From the invariant shown earlier, *L.head* $\neq$ `nil` implies *L.tail* $\neq$ `nil`. When *D* obtains a non-`nil` value from *L.head* with a *fetch-and-store* setting *L.head* to `nil`, *L.tail* remains non-`nil`. No enqueues update *L.head* while *L.tail* is non-`nil` and no dequeues other than *D* can affect the state of the shared list until *L.head* becomes non-`nil`. *D* does not set *L.tail* to `nil` or *L.head* to a non-`nil` value until its final operation on the shared list structure; therefore, the rest of *D*'s operations execute without interference from other dequeues.

- Concurrent enqueues and dequeues do not interfere with one another.

  A dequeue can change the value of *L.head* with a *fetch-and-store* only if *L.head* is non-`nil`. An enqueue can only update *L.head* if it obtains a `nil` from *L.tail*. As shown earlier, *L.tail* = `nil` only if *L.head* = `nil`; therefore, updates to *L.head* by enqueues and *fetch-and-store*s to *L.head* by dequeues do not interfere. A dequeue only issues an *atomic-write* to *L.head* when another element is in the list (*i.e.*, the list is non-empty and *L.tail* $\neq$ `nil`); therefore, *atomic-write*s to *L.head* by dequeues do not interfere with updates to *L.head* by enqueues either.

  Concurrent dequeues do not interfere with enqueues by overwriting values of *L.tail*. Dequeues update *L.tail* using *compare-and-swap*. A dequeue overwrites a value of *L.tail* using *compare-and-swap* only if its value matches the pointer obtained from *L.head*. An element is visible from the head of the queue only after its enqueue is complete. Therefore, a dequeue will not write `nil` over an element pointer in *L.tail* until the enqueue of the element is complete.

  When an enqueue is linking an element at the end of the queue, a concurrent dequeue will not cause the result of the enqueue to be lost. If the element being removed by a dequeue is not the only element in the list, then the dequeue does not interact with an enqueue. If the dequeue is removing the last element in a list, we must consider the interaction between the dequeue and an enqueue updating the element's *next* field. Since a dequeue atomically reads the *next* field of the element being dequeued and the update of the element's *next* field by an enqueue is also atomic, the enqueue updates the field either before or after the read. If the update occurs before the read, the enqueue is complete and the successor element is correctly in place. Otherwise, the dequeue sees a `nil` in the *next* field and executes a *compare-and-swap* on *L.tail* to finish unlinking its element. With a concurrent enqueue is in progress, the *compare-and-swap* fails signalling the enqueue's presence and the dequeue waits for the enqueue to finish linking the new element behind the singleton. The dequeue then copies a pointer to the new element into *L.head*, correctly installing it at the head of the list.

Linearizability

To demonstrate linearizability, we present a mapping that transforms an operation history for the concurrent queue into a linear order that is consistent with the real-time ordering of the operations and argue that this operation ordering provides FIFO semantics for the queue.

- Map each enqueue to the completion of its *fetch-and-store* operation on *L.tail*. If two simultaneous enqueues map to the same point as a result of true concurrency available with a *combining network* [7] (a switching network that combines operations directed at the same memory location), the mapping can be perturbed slightly to serialize the mapping of the operations in a manner consistent with the apparent serialization provided by the combining network.

- Map each dequeue returning a non-`nil` value to the completion of its last *fetch-and-store* on *L.head* and each dequeue returning `nil` to its last *atomic-read* of *L.tail*. Successful dequeues each map to a unique point since only one of a set of simultaneous *fetch-and-store* operations receives a non-`nil` value. Unsuccessful dequeues that map to the same point due to combination of simultaneous *atomic-read*s of *L.tail* can be serialized in an arbitrary order. A successful dequeue and an unsuccessful dequeue never map to the same point because an unsuccessful dequeue requires $L.tail = \texttt{nil}$, but a successful dequeue only occurs when the queue is non-empty and $L.tail \neq \texttt{nil}$.

To resolve a conflict when an enqueue and a dequeue are simultaneous, perturb the mapping as follows:

- If the dequeue is unsuccessful, map it before the enqueue. This insures that the queue is empty for the dequeue.

- If the dequeue is successful, map it after the enqueue. Perturbing the mapping in this manner will not violate the FIFO property as the presence of additional elements in the queue will not alter the result of a dequeue.

Mapping enqueues to their *fetch-and-store*s insures that they map in the order that they actually link the elements into the queue. Mapping successful dequeues to their last *fetch-and-store* on *L.head* serializes the dequeues in the order they remove elements from the queue. Clearly, a dequeue cannot obtain a pointer to an element using a *fetch-and-store* on *L.head* until after that element has been enqueued – enqueueing an element is the only way to (eventually) have a pointer to it appear in *L.head*.

An unsuccessful dequeue may occur only when *L.tail* is `nil`; therefore, *L.head* must also be `nil` (from the invariant shown earlier) and the queue indeed empty. This policy prevents a dequeue from missing an element enqueued by a concurrent enqueue.

Using these mappings for operations, any operation history permitted by the implementation can be turned into a linear ordering. Since each queue operation maps to a statement inside its active interval, the mapping satisfies the partial order induced by the real-time relationships of the intervals. Furthermore, the above arguments show that the mapping is consistent with a legal sequential history for the queue. Therefore all operation histories of the concurrent queue presented in this section satisfy the linearizability criterion.

2.3 Discussion

The important feature of the above algorithms for enqueueing and dequeueing items from a linked list is the lack of a critical section in the enqueue routine. This enables unbounded

concurrency among enqueue operations. The only limit on the potential parallelism among enqueues is the number of processors available, provided that the underlying hardware combines simultaneous *fetch-and-store* operations on *L.tail* using a combining network. Also, by substantially decoupling the enqueue and dequeue operations, a dequeue need only be aware of concurrent enqueues if it is removing the only element in a singleton queue. The algorithms designed by Rudolph for manipulating FIFO linked lists (as part of a larger queue data structure) use a critical section in both the enqueue and dequeue routines [9, p. 76]. This approach is inherently sequential, permitting only a single operation on each list at a time.

In the absence of a combining network, it is unclear that the implementation of an unbounded queue described herein is less efficient than Rudolph's implementation of a bounded queue [9, p.73]. In Rudolph's implementation, enqueues and dequeues update mutually shared counters with *fetch-and-add*s to maintain upper and lower bounds on the number of elements in the queue. Without a combining network, updates to these shared variables are sequential and limit the parallelism achievable. In the approach presented here, enqueue and dequeue operations are decoupled – enqueues operate on the tail pointer and dequeues operate on the head pointer – except in the boundary cases arising with singleton and empty queues. If the head and tail pointers are located in different memory banks, the operations can occur in parallel with no interaction. On the other hand, the short critical section in the dequeue algorithm could prove the limiting factor on the queue throughput. Empirical studies are necessary to reliably determine which implementation provides greater throughput.

Additionally, consider the performance of the algorithms presented here against the classic approach that uses a critical section guarded by a *test-and-set* to protect the integrity of a queue. In the presence of contention, our algorithms would certainly outperform any *test-and-set* style solution since we decouple enqueues and dequeues, whereas the classic critical section solution couples enqueues and dequeues. In the absence of contention, the algorithms presented here would again outperform a solution using *test-and-set* because they are so simple. Just executing a *test-and-set* on a lock and a remote write to clear the lock when finished with the critical section uses as many remote operations as the entire enqueue routine presented here; operating on the queue in the critical section requires additional operations. This observation also applies to the dequeue operation.

3 Balancing Parallelism Among Enqueues And Dequeues

Although the algorithms presented in section 2 are simple and efficient, the sequential nature of the dequeue operation may cause inadequate performance in the presence of contention since only one dequeue can execute its sequential stage at a time. To increase the potential parallelism available, this section introduces an alternative data structure for a concurrent queue implemented by an array of lists. (A similar approach was used by Rudolph [9].) The increase in parallelism provided by this data structure is orthogonal to those described in section 2 for a single linked list. The algorithms presented in this section satisfy the linearizability property and provide more parallelism than Rudolph's queue.

It is important to note that the improvements suggested in this section over the algorithms in section 2 would probably only be effective on a machine possessing a combining network. Without a combining network, algorithms that use *fetch-and-add* on shared variables to distribute operations through a concurrent data structure degrade due to contention for the shared variables. In such situations, simple solutions that use shared variables infrequently and have short critical sections, probably will perform better than complex solutions which admit greater potential concurrency.

```
const
    --NumLists must be defined as a power of 2
type
    guardedList = record --a linked-list with synchronization flags
        L: list;
        --flags for serializing enqueues and dequeues, respectively, on L
        eguard,dguard:  integer;
    end;

    --data structure for a highly parallel queue formed from an array of linked-lists
    queue = record
        glarray:  array [0..(NumLists - 1)] of guardedList; --an array of guarded lists
        --counters that cycle enqueues and dequeues through the list array
        enqCounter,deqCounter:  integer;
        --upper and lower bounds on the number of elements in the queue
        UBcount,LBcount:  integer;
    end;

procedure initQ(var Q: queue); --initialize the queue data structure
var i,prev_guard:  integer;
begin
    with Q do
        --initialize the operation and entry counts
        enqCounter := 0; deqCounter := 0; LBcount := 0; UBcount := 0;
        --initialize each list as empty and initialize the guards
        for i := 0 to (NumLists - 1) do
            initList(glarray[i].L);
            prev_guard := i - NumLists;
            glarray[i].eguard := prev_guard;
            glarray[i].dguard := prev_guard;
        end;
    end;
end;
```

The enqueue and dequeue operations presented in this section operate on a queue data structure represented as an array of guarded lists. Each guarded list contains two guards used to coordinate concurrent enqueues and dequeues, respectively, on that list. The guard

eguard (respectively, *dguard*) is used by FIFOenter and FIFOexit to provide mutual exclusion between enqueue (dequeue) operations on the same list and to insure that enqueue (dequeue) operations on a particular list enter their critical section in FIFO order of their assignment to that list.

```
--wait until the previous operation using this guard has completed
procedure FIFOenter(var g:  integer; counter:  integer);
var lcount:  integer;
begin
    lcount := counter - NumLists;
    repeat until atomic-read(g) = lcount;
end;

--signal completion of a guarded operation
procedure FIFOexit(var g:  integer; counter:  integer);
begin
    atomic-write(g,counter);
end;
```

Concurrent enqueues can be accomplished using *fetch-and-add* to update *Q.enqCounter*. Each enqueue receives a unique value from *Q.enqCounter* which is used modulo *NumLists* to direct the enqueue to the appropriate guarded list. Dequeues similarly use *Q.deqCounter*. Enqueues (respectively, dequeues) directed at a particular list are applied sequentially, but operations on different lists can proceed in parallel. Enqueues and dequeues operating on the same list do not interfere unless the list is empty or a singleton.

Maintenance of *Q.deqCounter*, although similar to that of *Q.enqCounter*, is nevertheless more difficult: special care must be taken when a dequeue operation is initiated on an empty queue. *Q.deqCounter* must never overtake *Q.enqCounter*; otherwise, some number of subsequent enqueues would appear lost, only to reappear later. This would violate both the FIFO and linearizability properties of the queue. For this reason, each dequeue operation cannot increment *Q.deqCounter* unless an element is available in the queue.

```
--return TRUE if the specified variable was decremented successfully
--from a positive value, else FALSE
procedure testDecrementRetest(var value:  integer):  boolean;
var result:  boolean;
begin
    result := (atomic-read(value) > 0);
    if result then
        if fetch-and-add(value,-1) <= 0 then
            fetch-and-add(value,1);
            result := FALSE;
        end;
    end;
    return result;
end;
```

To prevent premature increments of *Q.deqCounter*, the implementation maintains the variable *Q.LBcount*, which is a lower bound on the number of elements in the queue.[6] Each enqueue operation, after incrementing *Q.enqCounter*, increments *Q.LBcount* to signal the presence of another element in the queue. Before incrementing *Q.deqCounter*, each dequeue operation checks the value of *Q.LBcount* to make sure there is a free element in the queue. A dequeue secures the right to a free element in the queue by atomically decrementing *Q.LBcount* from a positive value using *testDecrementRetest*.

To provide linearizability for the queue, we must prevent a dequeue invoked on a non-empty queue from returning `nil`. Use of *Q.LBcount* alone is not sufficient to prevent this from occurring. In his implementation of an unbounded queue, Rudolph uses a single variable to keep track of the number of elements available [9, p. 76] and fails to provide linearizability. For an analysis of this point in the context of Rudolph's implementation, see appendix B, especially figure B.2 and its discussion.

To prevent a dequeue invoked on a non-empty queue from returning `nil`, we introduce an additional variable *Q.UBcount*, an upper bound on the number of elements in the queue. We maintain the invariant that $(Q.UBcount > 0)$ whenever an element is present in the queue. Before incrementing *Q.enqCounter*, each enqueue operation increments *Q.UBcount* to signal the pending addition of another element to the queue. After incrementing *Q.deqCounter*, each dequeue operation decrements the value of *Q.UBcount* to indicate another element in the queue has been sucessfully claimed.

To preserve linearizability, we insure that all elements in the queue have been claimed successfully before a dequeue returns `nil`; this is true if $(Q.UBcount = 0)$. If a dequeue does not wait until all elements in the queue have been claimed before returning `nil`, the following non-linearizable sequence can occur on a singleton queue containing *e0*:

1. process *p1* reserves an element in the queue by decrementing *Q.LBcount* to zero.
2. process *p2* issues a dequeue that finds $(Q.LBcount = 0)$ and returns `nil`,
3. process *p2* enqueues an element *e1* in the queue, incrementing *Q.LBcount*,
4. process *p2* issues a dequeue, decrements *Q.LBcount* and finds *e0* still in the queue as it had not yet been claimed successfully by *p1*.

Although *p1* had already decremented *Q.LBcount* to zero (indicating that no more free elements were available in the queue), it had not claimed *e0*, which *p2* then claimed and removed from the queue. Any attempt to linearize this set of operations would need operation (4) after operation (2) due to real-time precedence constraints, but (4) needs to serialize before (2) for this set of operations to be a legal sequential history.

The boolean function *queueEmpty* implements a protocol used by the dequeue operation which avoids the non-linearizable behavior described above. The *queueEmpty* function tests *Q.LBcount* with *testDecrementRetest* to see if an element in the queue is free; if not, it tests *Q.UBcount* to make sure all elements in the queue have been claimed successfully

[6] More accurately, *Q.LBcount* is a lower bound on the number of elements whose position in the queue has been secured, but have not yet been claimed by a dequeue.

before indicating that the queue is empty. Use of this protocol may cause a dequeue that will eventually return `nil` to temporarily delay; however, the dequeue will only have to wait until all other dequeues that received a positive value from *Q.LBcount* to claim their elements from the queue and to decrement *Q.UBcount*, bringing its value to zero.

```
--returns TRUE iff the queue is empty (Q.UBcount = 0), otherwise return
--FALSE when Q.LBcount has been decremented successfully from a positive value
--INVARIANT: the value of Q.UBcount is always non-negative
procedure queueEmpty(var Q: queue):  boolean;
begin
    repeat --until queue is empty, or Q.LBcount has been decremented successfully
        if testDecrementRetest(Q.LBcount) then return FALSE; end;
    until (atomic-read(Q.UBcount) = 0);
    return TRUE;
end;

--enqueue an element on the highly parallel FIFO queue
procedure enqueue(var Q: queue; var e:  element);
var last:  pointer to element;
    index,ecount:  integer;
begin
    --signal the presence of another element in the queue
    fetch-and-add(Q.UBcount,1);
    --get the list for the current enqueue, send the next enqueue to the next list
    ecount = fetch-and-add(Q.enqCounter,1);
    --increment Q.LBcount to permit another dequeue
    fetch-and-add(Q.LBcount,1);
    --compute the list index from the counter value
    index := ecount MOD NumLists;
    --enqueue the element on the specified list
    --insuring that the previous enqueue on this list has completed its critical section
    FIFOenter(Q.glarray[index].eguard,ecount);
        --replace the tail of the list with e, returning the previous tail
        last := fetch-and-store(Q.glarray[index].L.tail,e);
    FIFOexit(Q.glarray[index].eguard,ecount);
    if last = nil then --list was previously empty: update the head pointer
        atomic-write(Q.glarray[index].L.head,e);
    else
        atomic-write(last^.next,e); --link e behind last
    end;
end;
```

For the queue described in this section, the enqueue operation takes two arguments: Q, a reference to a queue structure, and e, a reference to an element. The counter *Q.UBcount* is incremented to indicate the pending addition of a new element to the queue. Next, the

enqueue retrieves a unique value from *Q.enqCounter* with a *fetch-and-add* and increments *Q.LBcount* to permit another dequeue operation to begin. The value (*Q.enqCounter* modulo *NumLists*) specifies the list in the array into which the element will be inserted. FIFOenter and FIFOexit use the *eguard* of the selected list to serialize concurrent enqueues upon it. Using these two routines insures that elements are placed on the list in the order which the enqueue operations were assigned to the list (based on the value the value each retrieved from *Q.enqCounter*). This restriction is necessary to maintain the FIFO and linearizability properties. The algorithm used to enqueue *e* on the list is identical to the one presented in section 2.

```
--dequeue an element from the highly parallel FIFO queue
procedure dequeue(var Q: queue):  element
var firstEl, secondEl:  pointer to element;
    dcount, index:  integer;
begin
    if queueEmpty(Q) then return nil; else
        --get the list for the current dequeue, send the next dequeue to the next list
        dcount := fetch-and-add(Q.deqCounter,1);
        --decrement Q.UBcount to show one less element available
        fetch-and-add(Q.UBcount,-1);
        index := dcount MOD NumLists;
        --insure previous dequeue on this list has completed its critical section
        FIFOenter(Q.glarray[index].dguard,dcount);
            repeat --until a non-nil element has been found
                firstEl := atomic-read(Q.glarray[index].L.head);
            until not (firstEl = nil);
            --get a pointer to the next element in the list
            secondEl := atomic-read(firstEl^.next);
            --splice the first element off the head of the list
            Q.glarray[index].L.head := secondEl;
            if secondEl = nil then
                --firstEl was the only element in the list, need to update the list tail
                if compare-and-swap(Q.glarray[index].L.tail,firstEl,nil)
                    = FALSE then
                    --firstEl is no longer tail: a concurrent enqueue is detected
                    --loop until the concurrent enqueue completes
                    repeat until not (firstEl^.next = nil);
                    --update head pointer
                    Q.glarray[index].L.head := firstEl^.next;
                end;
            end;
        FIFOexit(Q.glarray[index].dguard,dcount); --exit the critical section
    end;
    return firstEl; --return the element dequeued
end;
```

The dequeue operation for the queue described in this section takes a single argument Q, a reference to a queue structure. The dequeue routine checks the queue for emptiness using the boolean function *queueEmpty*. If the queue is empty, dequeue returns `nil`. Otherwise, the dequeue retrieves a unique value from *Q.deqCounter* with a *fetch-and-add* and decrements *Q.UBcount* to indicate that it has claimed its element from the queue. The value (*Q.deqCounter* modulo *NumLists*) specifies the list in the array from which the element will be removed. FIFOenter and FIFOexit use the *dguard* for the selected list to serialize concurrent dequeues upon it. Using these two routines insures that dequeues on a list proceed in the order which they were assigned to the list (based on the value the value each retrieved from *Q.deqCounter*). Thus, retrieving a unique value from *Q.deqCounter* claims a particular element in the queue for the dequeue operation. As with enqueues, dequeues must be serialized on the list to maintain the FIFO and linearizability properties.

Once exclusive access to the target list has been insured with FIFOenter, the dequeue operation polls the head of the list until an element is found. Polling is necessary since the dequeue may need to wait for a concurrent enqueue to complete. The loop will terminate since access is granted to this list only if it already contains an element, or a concurrent enqueue was in the process of adding an element to the list. Inside the critical section, the dequeue splices the first element from the head of the list and handles a dequeue from a singleton list as in the algorithm presented in section 2. The code to handle a singleton list can be moved outside the critical region; however, it provides only very limited parallelism and complicates the analysis of the dequeue algorithm.

3.1 Restrictions

The restrictions presented in section 2.1 on arguments to the enqueue routine of section 2 apply here as well.

The algorithms presented in section 3 make no effort to bound the indices *Q.enqCounter* and *Q.deqCounter*; however, this implementation does not require infinite counters. By restricting *NumLists* to a power of 2, modular arithmetic correctly selects the appropriate list in the array despite wrapping of these counter values. In practice, restricting *NumLists* to a power of 2 is not a problem and maintaining the counters with a simple protocol insures good performance.

Several other limitations need to be imposed to insure the correctness of the algorithms in section 3. First, $\mathbf{P}$, the number of processes must be restricted such that:

$$\mathbf{P} < \frac{|range\ of\ integer|}{NumLists}$$

This insures that no two processes will be active on the same list in the queue with the same counter value. Second, the number of entries in the queue at any particular time must be less than or equal to the $|range\ of\ positive\ integer|$. This is necessary to insure that neither *Q.LBcount* nor *Q.UBcount* wraps negative as a result of increments by the enqueue routine. Finally, the number of concurrent dequeues pending must not exceed the number of completed enqueues by $|range\ of\ negative\ integer|$, otherwise *Q.LBcount* could wrap from a negative to a positive value.

Note that none of these restrictions effects the practicality of these algorithms for use on real architectures.

3.2 Properties

To complete the description of the enqueue and dequeue routines in this section, as in section 2, we show that the routines correctly implement a linearizable FIFO queue. We argue sequential correctness and interference freedom for the queue operations and conclude by showing that each operation history for the queue can be mapped to a linear order that corresponds to a legal sequential history for a FIFO queue.

Sequential Correctness

- An enqueue correctly adds an element at the rear of the queue.

  Each enqueue obtains a value from *Q.enqCounter* using *fetch-and-add* which specifies the list into which the element will be enqueued. Use of this counter cycles enqueues through the lists. Dequeues cycle in an identical fashion. Therefore, appending an element at the end of the list specified by the counter adds an element to the end of the queue. FIFOenter and FIFOexit use the *eguard* of the selected list used to permit one enqueue operation per list at a time; this will not cause a single enqueue to wait. Enqueueing an element on the selected list uses the same sequence of operations as the enqueue operation presented in section 2 and the same arguments for sequential corrrectness apply.

- A dequeue correctly removes an element from the front of the queue.

  A dequeue secures rights to an element in the queue by decrementing *Q.LBcount* from a positive value. Since each enqueue increments *Q.LBcount* after its *fetch-and-add* on *Q.enqCounter*, this prevents a dequeue from executing a *fetch-and-add* on *Q.deqCounter* that would cause it to overtake *Q.enqCounter*. This protocol insures that a dequeue will never wait at a list for an an element whose enqueue has not yet begun. In the absence of concurrent operations on the queue, the value of *Q.LBcount* is identical to that of *Q.UBcount* and always non-negative. If a dequeue sees the value of *Q.LBcount* as zero, *Q.UBcount* will also be zero; the number of successful dequeues that occurred equals the number of enqueues that occurred; thus, the queue is empty and the dequeue returns `nil`.

  If a positive value of *Q.LBcount* is seen, the value obtained from *Q.deqCounter* with a *fetch-and-add* specifies the list from which the dequeue gets its element. Since enqueues and dequeues cycle through the lists in the same order and *Q.enqCounter* never overtakes *Q.deqCounter*, the specified list contains the oldest element in the queue at its head. FIFOenter and FIFOexit use the *dguard* of the selected list to permit one dequeue operation per list at a time; this will not cause a single dequeue to wait. The algorithm used to dequeue an element from the specified list is essentially the same as that presented in section 2. The dequeue polls the head of the list to get its element. In the absence of concurrent operations on the queue, this poll succeeds

the first time. Next, the dequeue splices its element out of the list. If *secondEl* is non-`nil`, then *L.head* points to the next element in the list when the dequeue finishes. Otherwise, *L.head* gets `nil`; the compare-and-swap with *L.tail* succeeds (since there is no concurrent enqueue) and *L.tail* also gets `nil`. Both of these operation sequences leave the list in a consistent state.

Interference Freedom

Above we showed that enqueues and dequeues perform properly if executed in mutual exclusion. Here, we argue that concurrent enqueues and dequeues perform correctly even in the presence of concurrent operations.

- Concurrent enqueues do not interfere with one another.

  All operations by an enqueue on the counters in the shared queue structure are atomic. Regardless of how the operations are interleaved, each enqueue results in a single increment to *Q.LBcount*, *Q.UBcount* and *Q.enqCounter*. Concurrent enqueues each obtain a different value from *Q.enqCounter* which indicates the appropriate list for the current enqueue. FIFOenter and FIFOexit restrict the order in which enqueues are applied to the selected list. Specifically, they insure that the ith enqueue on a list secures a position for its element with its *fetch-and-store* on the list tail before the $i+1$st enqueue on that list may proceed. By regulating mutual exclusion around the *fetch-and-store* operation using the value from *Q.enqCounter*, we insure that all elements are put into the queue in proper order. Interference between enqueue operations adding elements to the same list is ruled out by the same arguments presented for the algorithms in section 2.

- Concurrent dequeues do not interfere with one another.

  A dequeue secures rights to an element in the list by decrementing *Q.LBcount* from a positive value. The atomicity of this operation insures the number of dequeues permitted to proceed will not exceed the number of enqueues initiated on the queue. The atomicity of *fetch-and-add* insures *Q.UBcount* is correct in spite of interleavings and *Q.deqCounter* always indicates the appropriate list for the current dequeue. As with enqueues, FIFOenter and FIFOexit restrict the order in which dequeues are applied to the selected list. Mutual exclusion among dequeues around the operation sequence that splices an element out of the list insures that no two dequeues interfere in their operations on the list.

- Concurrent enqueues and dequeues do not interfere with one another.

  Q.enqcounter is used exclusively by enqueues and *Q.deqCounter* is used used exclusively by dequeues. Therefore, interactions through these counters are impossible.

  Consider the interactions through *Q.LBcount* and *Q.UBcount*. All operations on the counters *Q.LBcount* and *Q.UBcount* queue are atomic; therefore, individual operations will not interfere. Since enqueues do do not use the values returned by *fetch-and-add*s on these counters, enqueues are not affected by operations on these counters

by concurrent dequeues. Dequeues use the values of *Q.LBcount* and *Q.UBcount* to test the queue for emptiness. If more enqueues have been initiated than successful dequeues, then a dequeue may proceed; concurrency among enqueue and dequeue operations does not invalidate this use.

If the concurrent enqueues and dequeues are directed to separate lists in the queue, there is no further opportunity for interference. To complete the arguments for interference freedom, we need only show that concurrent enqueues and dequeues on the same list, also do not interfere. Consider concurrent operations on a particular list L.

- Updates to *L.head* by enqueues and writes to *L.head* by dequeues do not interfere. From the invariant shown in section 2, *L.head* $\neq$ `nil` implies *L.tail* $\neq$ `nil`. A dequeue, *D*, can write *L.head* only after it completes its repeat loop in which it finds *L.head* $\neq$ `nil`; thus, *L.tail* is also non-`nil` after the loop. Since only a dequeue can set *L.tail* to `nil` and dequeues execute their operations on the shared list structure in mutual exclusion, *L.tail* cannot become `nil` while *D* executes unless *D* sets it to `nil`. The two statements in which *D* may write *L.head* execute only when *L.tail* is non-`nil`. An enqueue can only update *L.head* if it obtains a `nil` from *L.tail*. Therefore, the property holds.
- Concurrent dequeues do not interfere with enqueues through interactions with *L.tail* and elements' *next* fields. Since enqueues and dequeues interact with one another through these fields list as in the algorithms presented in section 2, the non-interference arguments in section 2 apply.

Linearizability

To demonstrate linearizability, we present a mapping that transforms an operation history for the concurrent queue into a linear order that is consistent with the real-time ordering of the operations and argue that this operation ordering provides FIFO semantics for the queue.

- Map each enqueue operation to the completion of its *fetch-and-add* on the counter *Q.enqCounter*. If enqueues map to the same point as a result of some combination of simultaneous *fetch-and-add*s to *Q.enqCounter*, the mapping can be perturbed slightly to serialize the mapping of the operations in a manner consistent with the apparent serialization provided by the combining network.
- Map each dequeue returning `nil` to the completion of its final *atomic-read* of *Q.UBcount* (which finds *Q.UBcount* = 0). At this point the queue is guaranteed to be empty. If two such dequeues map to the same point due to combination of simultaneous *atomic-read*s of *Q.UBcount*, they can be serialized in an arbitrary order. For dequeues returning a non-`nil` value, map the dequeue to the completion of its *fetch-and-add* on *Q.deqCounter*. For each pair of successful dequeues that maps to the same point in time, perturb the mapping slightly to serialize the mapping of the operations in a manner consistent with the apparent serialization provided by the combining network. An unsuccessful dequeue will never map to the same point as a successful dequeue:

Q.UBcount is always greater than zero while a dequeue is executing its *fetch-and-add* on *Q.deqCounter*; therefore, unsuccessful dequeue cannot find $Q.UBcount = 0$ at the same time.

If an enqueue and a dequeue map to the same point in time, perturb the mapping so that the dequeue occurs after the enqueue. Modifying the mapping in this manner will never violate the FIFO property.

Each enqueue of an element maps before its corresponding dequeue. Since an enqueue increments *Q.LBcount* following its *fetch-and-add* on *Q.enqCounter*, and a dequeue must decrement *Q.LBcount* from a positive value before it can execute its *fetch-and-add* on *Q.deqCounter*, then each dequeue's *fetch-and-add* must occur after the corresponding enqueue's. Mapping enqueues to their *fetch-and-add*s insures that they are mapped in the order that the elements assume in the queue. Mapping successful dequeues to their last *fetch-and-add* serializes dequeues in the order in which they claim elements from the queue.

Checking that $Q.UBcount = 0$ before returning `nil` from a dequeue D insures that all elements in the queue have been already claimed by some dequeue. Specifically, this protocol insures that dequeues for all elements enqueued before D occurs serialize before D.

Finally, incrementing *Q.UBcount* prior to an enqueue and decrementing it following a dequeue insures that *Q.UBcount* will always be greater than zero if the queue is non-empty. Since a dequeue will never return `nil` unless $Q.UBcount = 0$, a dequeue will never return `nil` when invoked on a non-empty queue. This policy prevents a dequeue from missing an element enqueued by a concurrent enqueue.

Using these mappings for operations, any operation history permitted by the implementation can be turned into a linear ordering. Since each queue operation maps to a statement inside its active interval, the mapping satisfies the partial order induced by the real-time ordering of the intervals. Furthermore, the above arguments show that the mapping is consistent with a legal sequential history for the queue. Therefore all operation histories of the concurrent queue presented in this section satisfy the linearizability criterion.

4 Conclusions

Developing algorithms for operations on a concurrent queue that permit only linearizable operation histories proved surprisingly difficult. The primary difficulty centered on insuring that all operations have a consistent view of when each class of operation takes effect.

Insuring that all operations have a consistent view of when each class of operation takes effect is crucial to constructing algorithms for linearizable operations. For example, consider Rudolph's algorithms in appendix B. Inserts are serialized on their *fetch-and-add* to *Tail*; thus, to inserts, each insert appears to take effect at the completion of its *fetch-and-add* to *Tail*. Elements inserted, however, are not available to a remove operation until the insert executes its *fetch-and-add* to *#Ql*. Therefore, from the point of view of remove operations, inserts appear to take effect at the *fetch-and-add* to *#Ql*. This difference between the two classes of operations in their viewpoint as to when inserts take effect admits anomalous behavior that prevents linearizability. In figure B.1, an operation history is shown that

forces the effect of an insert operation to map to a point long after the insert completes; this mapping, which violates the real-time precedence constraint of linearizability, results from the difference in viewpoint.

In the algorithms presented in section 3, the protocol using *Q.LBcount* and *Q.UBcount* insures that both enqueue and dequeue operations maintain consistent views as to when enqueues take effect. In particular, an enqueue operation is seen by other enqueues to map to its *fetch-and-add* on *Q.enqCounter*. Dequeues view an enqueue operation as occurring in the interval between the enqueue's *fetch-and-add*s on *Q.UBcount* and *Q.LBcount*: a dequeue cannot return `nil` while *Q.UBcount* is non-zero, and also cannot try to grab an element from the queue until *Q.LBcount* is greater than zero. Since dequeues view an enqueue as occurring in the interval between these two operations, it is consistent with their view to map the enqueue to its *fetch-and-add* on *Q.enqCounter*. The consistency of these viewpoints prevents concurrent operations from exhibiting anomalous behavior.

Finally, the algorithms in this paper have implications for designing hardware for parallel machines. Machines have traditionally provided *test-and-set* primitives for process synchronization; few parallel machines provide more complex hardware primitives for process synchronization and interprocess communication. In section 2 of this paper, a queue implementation is presented that would provide better performance than any *test-and-set* style implementation. The primitives required for this implementation, while more costly to implement than *test-and-set*, are nevertheless reasonably simple and general purpose. Therefore, they are strong candidates for hardware implementation.

Acknowledgements

Lawrence Crowl and Peter Dibble helped uncover errors in early versions of the queue algorithms. Also, their detailed comments on draft versions, along with those of Tom LeBlanc, significantly improved the presentation of this paper.

Appendix A: Fetch-and-Φ Primitives

This appendix contains functional specifications for the fetch-and-Φ primitives used by the queue algorithms in this paper. Each of these operations is atomic, namely, it executes indivisibly with respect to other operations on the target memory location. Note that the semantics for the *fetch-and-store* primitive have often been referred to as *swap*; however, we use the *fetch-and-store* name to emphasize the asymmetry of this operation (as opposed to a symmetric *memory-to-memory swap*).

All of the operations described, with the exception of *compare-and-swap*, could take effective advantage of a combining switch to provide parallelism among a set of operations simultaneously directed at the same memory location . Two *compare-and-swap* operations are only combinable if `val1` of the first equals `val2` of the second. Note however that the algorithms presented in the body of this paper never execute in more than one *compare-and-swap* operation on a memory location at a time. A single *compare-and-swap* can be effectively combined with the other operations.

```
--atomically read the value of x and return it
atomic-read(var x:  word):  word;
begin atomic
    return x;
end atomic;

--atomically replace the value of x with val
atomic-write(var x:  word; val:  word);
begin atomic
    x := val;
end atomic;

--atomically add the value val to x returning x's old value
fetch-and-add(var x:  integer; val:  integer):  integer;
var temp:  integer;
begin atomic
    temp := x;
    x := x + val;
    return temp;
end atomic;

--atomically replace the value of x with val returning x's old value
fetch-and-store(var x:  word; val:  word):  word;
var temp:  word;
begin atomic
    temp := x;
    x := val;
    return temp;
end atomic;
```

```
--atomically replace the value of x with val2 if the value
--of x is equal to val1 and return a boolean success code
compare-and-swap(var x:  word; val1,val2:  word):  boolean;
var result:  boolean;
begin atomic
    result := (x = val1);
    if result then x := val2; end;
    return result;
end atomic;
```

Appendix B: Analysis of Rudolph's Concurrent Queue

Rudolph's dissertation presents an implementation for managing an unbounded concurrent queue represented as an array of QSIZE lists [9, p. 76]. The variables *Head* and *Tail* are counters used to direct operations on the queue to a particular list. The variable *#Ql* is a lower bound on the number of elements in the queue. His algorithms for managing an unbounded queue are of the following form:

```
procedure insert(p:  pointer to element;var Q: queue);
var myloc:  integer;
begin
    myloc := fetch-and-add(Tail,1) mod QSIZE;
             ⋮
        insert p into list myloc of Q
             ⋮
    fetch-and-add(#Ql,1);
end;

procedure remove(var p:  pointer to element; var Q: queue);
var myloc:  integer;
begin
    if testDecrementRetest(#Ql,1) then
        myloc := fetch-and-add(Head,1) mod QSIZE;
             ⋮
        remove an element from list myloc of Q assigning it to p
             ⋮
    else
             ⋮
        indicate that queue was empty
             ⋮
    end;
end;
```

Insert operations serialize addition of elements to the queue by the order of their *fetch-and-add* on *Tail*. Remove operations serialize removal of elements from the queue by the order of their *fetch-and-add* on *Head*. These algorithms permit non-linearizable operation histories for the queue data structure. Two classes of non-linearizable operation histories are shown below.

Consider executing the operation history shown in figure B.1 on an empty queue. *Insert1* is serialized before *insert2* since its *fetch-and-add* on *Tail* occurs first. The element added to the queue by *insert1* becomes available for removal after the *fetch-and-add* on *#Ql* in *insert2*. *Remove1* retrieves the element placed in the queue by *insert1*. However, the queue appears

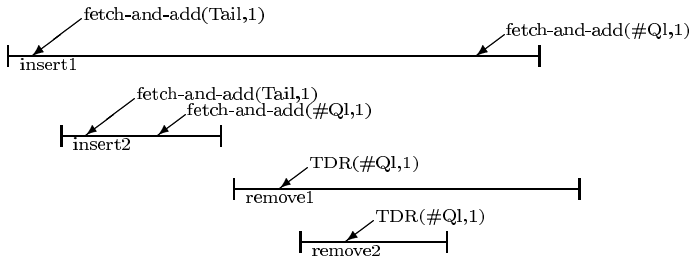

Figure B.1

empty to *remove2* since *#Ql* is again zero following a decrement in the *testDecrementRetest* (TDR) operation of *remove1*. The element added to the queue by *insert2* becomes available for removal after the *fetch-and-add* on *#Ql* by *insert1*. Since *insert2* strictly precedes *remove2* and the element enqueued by *insert2* has not been dequeued by any intervening remove, the element should be visible to *remove2*. This is not the case. Any mapping that transforms the operation history to a legal sequential history must map *insert2* after *remove2*, violating a real-time precedence constraint of the operation history. Therefore, this operation history is not linearizable.

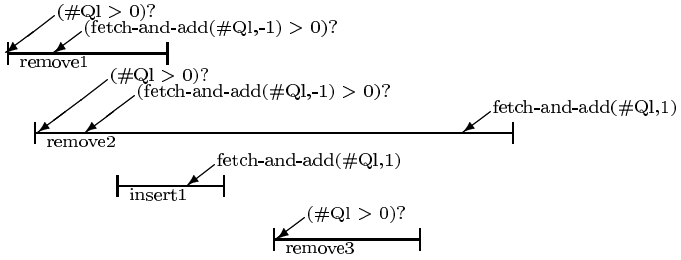

Figure B.2

Non-linearizable histories can arise using Rudolph's queue implementation in a second way as well. Consider executing the operation history in figure B.2 on a queue containing one element. In this figure, the constituent parts of the *testDecrementRetest* operation are shown where appropriate. *Remove1* and *remove2* find *#Ql* equal to 1 when they execute their test. *Remove1* atomically decrements *#Ql* from 1 to 0 and the second test is successful. *Remove2* atomically decrements *#Ql* from 0 to -1 and fails the second test, so it must increment *#Ql* to restore its state. Before *remove2* increments *#Ql*, *insert1* occurs, which atomically increments *#Ql* from -1 to 0. Next, *remove3* finds $\#Ql = 0$ and returns, failing to find an element in the queue. Finally, *remove2* atomically increments *#Ql* from 0 to 1, making the element added by *insert1* visible. Having the queue appear empty to *remove3* violates the real-time precedence constraint necessary for linearizability since *insert1* strictly precedes *remove3* and the element enqueued by *insert1* has not been dequeued by any intervening

remove. Therefore, this operation sequence is also not linearizable.

References

[1] BBN Laboratories. *Butterfly Parallel Processor Overview.* BBN Laboratories, Cambridge, Massachusetts, June 1985.

[2] Isaac Dimitrovsky. A group lock algorithm with applications. Ultracomputer Note 112, Courant Institute, New York University, November 1986.

[3] Jan Edler, Allan Gottlieb, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, Marc Snir, Patricia J. Teller, and James Wilson. Issues related to MIMD shared-memory computers: The NYU Ultracomputer approach. In *Proc. of the 12th Annual International Symposium on Computer Architecture*, pages 126–135, June 1985.

[4] Jan Edler, Allan Gottlieb, and Jim Lipkis. Operating systems considerations for large-scale MIMD machines. Ultracomputer Note 92, Courant Institute, New York University, December 1985.

[5] Allan Gottlieb and Clyde P. Kruskal. Coordinating parallel processors: A partial unification. *Computer Architecture News*, 9(6):16–24, October 1981.

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

[7] Clyde P. Kruskal, Larry Rudolph, and Mark Snir. Efficient synchronization on multiprocessors with shared memory. In *Proc. of the 5th Annual ACM Symposium on Principles of Distributed Computing*, pages 218–228, 1986.

[8] Leslie Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, April 1983.

[9] Lawrence Rudolph. *Software Structures for Ultraparallel Computing.* PhD thesis, New York University, February 1982.