Using Elimination to Implement Scalable and Lock-Free FIFO Queues

Mark Moir[*] Daniel Nussbaum[*] Ori Shalev[*†] Nir Shavit[*]

[*]*Sun Microsystems Laboratories, 1 Network Drive, Burlington 01803, Massachusetts, USA*
[†]*Tel Aviv University, Tel Aviv, Israel*

ABSTRACT

This paper shows for the first time that elimination, a scaling technique formerly applied only to counters and LIFO structures, can be applied to FIFO data structures, specifically, to linearizable FIFO queues. We show how to transform existing nonscalable FIFO queue implementations into scalable implementations using the elimination technique, while preserving lock-freedom and linearizablity.

We apply our transformation to the FIFO queue algorithm of Michael and Scott, which is included in the Java^{TM} Concurrency Package. Empirical evaluation on a state-of-the-art CMT multiprocessor chip shows that by using elimination as a backoff technique for the Michael and Scott queue algorithm, we can achieve comparable performance at low loads, and improved scalability as load increases.

Categories and Subject Descriptors

E.1 [**Data**]: Data Structures

General Terms

Algorithms, Design, Theory, Verification

Keywords

Multiprocessors, nonblocking synchronization, scalability, elimination, lock-free, linearizability, FIFO queues

1. INTRODUCTION

Elimination is a parallelization technique that has shown promise in designing scalable shared counters [2, 20] and Last-In-First-Out (LIFO) structures such as pools and stacks [7, 20]. This paper shows the first example of applying elimination to First-In-First-Out (FIFO) structures, specifically, to one of the most fundamental and widely studied concurrent data structures in the literature: the concurrent FIFO queue [6, 9, 10, 12, 14, 17, 18, 19, 22, 23, 24, 25].

1.1 Scalability of Concurrent Queues

The state of the art in concurrent FIFO queues employs data structures that support lock-free *Enqueue* and *Dequeue* operations with the usual FIFO queue semantics [15]. The most widely known concurrent FIFO queue implementation is the lock-free FIFO queue due to Michael and Scott [15] (henceforth *MS-queue*), which is included as part of the Java^{TM}Concurrency Package [13]. On shared-memory multiprocessors, this queue improves on all previous algorithms and even outperforms lock-based queues [15].[1] Its key feature is that concurrent accesses to the head and tail of the queue do not interfere with each other as long as the queue is non-empty. A recent paper by Ladan-Mozes and Shavit [11] introduced an *optimistic queue* that improves on the performance of the MS-queue in various situations by reducing the number of expensive compare-and-swap (CAS) operations performed. Unfortunately, like all previous FIFO queue algorithms, these state-of-the-art algorithms *do not scale.* In all previous FIFO queue algorithms, all concurrent Enqueue and Dequeue operations synchronize on a small number of memory locations, such as a head or tail variable, and/or a common memory location such as the next empty array element. Such algorithms can only allow one Enqueue and one Dequeue operation to complete in parallel, and therefore cannot scale to large numbers of concurrent operations.

We show how existing nonscalable queue implementations — including both of the above state-of-the-art queues — can be modified to support scalable FIFO elimination; this yields the first scalable non-blocking FIFO queue algorithms.

1.2 Elimination

Elimination is a technique introduced by Shavit and Touitou [20] to achieve scalability in shared pool and counter implementations [2, 20]. A recent paper by Hendler et. al [7] showed how elimination can be used as a backoff technique that achieves scalability for LIFO stacks while preserving linearizability. (Linearizability [9] is a standard correctness condition for shared data structures; it is defined in the next section.) The introduction of elimination into the backoff mechanism serves the dual purpose of allowing operations to complete in parallel and reducing contention for the underlying stack data structure.

[1]Parallel queue algorithms based on blocking combining approaches [5, 4] achieve good throughput for hundreds of processors, but are not competitive when concurrency is low [21]. This paper focuses on algorithms that are competitive with existing algorithms when concurrency is low, but achieve increasing throughput with increasing concurrency.

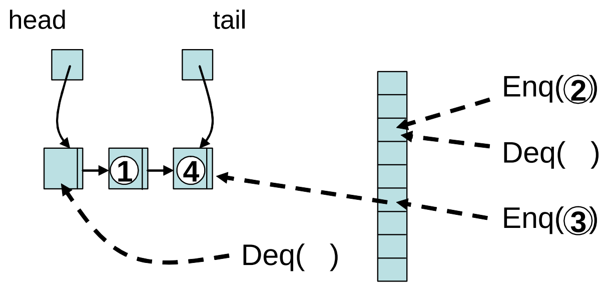


Figure 1: An Elimination Queue consisting of an MS-queue augmented with an elimination array shown during the execution depicted in Figure 2. Four Enqueues are attempted concurrently. The Enqueues of 2 and 3 fail and thus try to eliminate on random locations in the elimination array. Then two concurrent Dequeues are started. One succeeds and returns 1, the other fails, backs-off, and successfully eliminates on the array with the Enqueue of 2, which is now properly "aged."

Elimination works by allowing opposing operations such as pushes and pops to exchange values in a pairwise distributed fashion without synchronizing on a centralized data structure. This technique is straightforward in LIFO ordered structures. As noticed by Shavit and Touitou [20]: a stack's state remains the same after a *push* followed by a *pop* are performed. This means that if pairs of pushes and pops can meet and pair up in separate random locations of an "elimination array", then the threads can exchange values without having to access a centralized stack structure. However, this approach seemingly contradicts the very essence of FIFO ordering in a queue: a Dequeue operation must take the oldest value currently waiting in the queue. It apparently cannot eliminate with a concurrent Enqueue.

We show that, despite this inherent difficulty, any FIFO queue implementation that can support additional NumDeqs and NumEnqs operations (explained later), can be transformed into a scalable elimination queue, while preserving lock-freedom and linearizablity. It is easy to modify MS-queue and optimistic-queue to support these operations.

1.3 FIFO Elimination

How can elimination be achieved with a FIFO queue? Our key observation is that the FIFO order of Enqueue and Dequeue does not prohibit elimination, it only restricts it to particular combinations of operations. Specifically, a Dequeue operation can eliminate an Enqueue *if the values inserted by all Enqueues preceding that Enqueue have already been Dequeued.* Thus, if an Enqueue operation has "aged" to the point where the values of all Enqueues preceding it have been Dequeued, it can eliminate with a concurrent Dequeue operation. In this case, we "pretend" that the eliminated Enqueue completed successfully earlier, and that, because of the aging, its value is now at the head of the queue, and can therefore now be dequeued.

A variety of FIFO queue implementations can be based on this technique. In general, it is preferable to access the underlying (central) queue data structure directly under low load, because elimination partners are harder to find; and to attempt to eliminate under high load, because elimination partners will be easier to find, and excessive contention for the central queue will result in nonscalable performance.

One intuitively appealing way to use elimination is to incorporate it into the backoff mechanism for the central queue. It is well known that backoff techniques are necessary to alleviate the poor scalability of existing FIFO queue algorithms under high load. If an operation uses time that would otherwise be spent backing off to successfully eliminate, then the eliminated operations finish sooner, and also reduce contention on the central queue, because they do not have to retry after backing off. We describe our technique using this approach, and later discuss some potential disadvantages of this approach and some alternatives.

Our implementation uses a single "elimination array" to support a backoff scheme on a shared lock-free queue. We started with MS-queue, and modified the queue so that processes can query it to determine how many Enqueue and Dequeue operations have succeeded in the past, and this information is used to determine when an Enqueue operation has been properly "aged", and can therefore be eliminated. Figure 1 shows an example execution. Enqueue and Dequeue operations that fail to complete on the central queue due to interference from concurrent operations back off to the array to try to achieve elimination between a Dequeue operation and a sufficiently aged Enqueue operation. If the elimination is successful, they exchange values; otherwise, they again attempt to access the central queue. We have proved that this structure is linearizable; we present a detailed overview of our proof in Section 3.

Because our algorithm works as a backoff scheme, it can deliver the same performance as the simple queue at low loads. However, unlike the simple queue, its throughput increases as load increases because (1) the number of successful eliminations grows, allowing many operations to complete in parallel, and (2) contention on the shared queue is reduced beyond levels achievable by the best exponential backoff schemes [1] since many backed off operations are eliminated in the array.

1.4 Performance

Recent initiatives by leading processor manufacturers make it clear that the next generation of high-performance computer chips will be *chip-multi-threaded* (CMT). CMTs have multiple cores with multiple computation strands on a single chip. Effective data structures for multiprocessor chips should achieve scalability through parallelism, while imposing low overhead when concurrency is low.

We compared an elimination-backoff version of the MS-queue with the original MS-queue on a Sun Niagara-based system. Niagara is a CMT multiprocessor chip with 8 computing cores and 4 interleaved strands per core. Our empirical results show that our new elimination-backoff MS-queue performs comparably with MS-queue at low levels of concurrency, and then increasingly outperforms MS-queue as the number of threads increases. We believe that the parallelism afforded by the elimination technique will enable FIFO queues to scale to very large systems, while the MS-queue algorithm clearly will not.

2. THE NEW ALGORITHM

Our scalable FIFO queue algorithm is based on ideas similar to those of Hendler et. al. [7]. However, elimination for

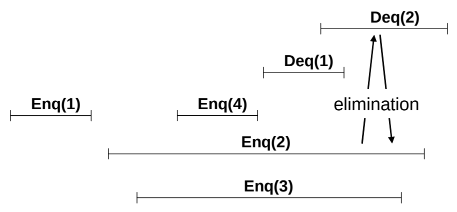


Figure 2: Example execution illustrating linearizability of elimination queue. Time progresses from left to right.

a FIFO queue is substantially more difficult than for a stack, because we cannot simply eliminate any Enqueue-Dequeue pair. The reason is that, while a push followed by a pop on a stack has no net effect on the stack, the same is not true for a FIFO queue. For example, if a queue contains a single value 1, then after an Enqueue of 2 and a Dequeue, the queue contains 2, regardless of the order of these operations. Thus, because the queue changes, we cannot simply eliminate the Enqueue and Dequeue. Note that if the queue were empty, we *could* eliminate an Enqueue-Dequeue pair, because in this case the queue is unchanged by an Enqueue immediately followed by a Dequeue. Our algorithm exploits this observation, but also goes further, allowing elimination of Enqueue-Dequeue pairs even when the queue is not empty.

To understand why it is acceptable in some cases to eliminate Enqueue-Dequeue pairs even when the queue is not empty, one must understand the linearizability correctness condition [9], which requires that we can order all operations in such a way that the operations in this order respect the FIFO queue semantics, but also so that no process can detect that the operations did not actually occur in this order. If one operation completes before another begins, then we must order them in this order. Otherwise, if the two are concurrent, we are free to order them however we wish.

Key to our approach is the observation that we really want to use elimination when the load on the queue is high. In such cases, if an Enqueue operation is unsuccessful in an attempt to access the queue, it will generally backoff before retrying. If in the meantime all values that were in the queue when the Enqueue began are dequeued, then we can "pretend" that the Enqueue did succeed in adding its value to the tail of the queue earlier, and that it now has reached the head and can be dequeued by an eliminating Dequeue. Thus, we use time spent backing off to "age" the unsuccessful Enqueue operations so that they become "ripe" for elimination. Because this time has passed, we ensure that the Enqueue operation is concurrent with Enqueue operations that succeed on the central queue, and this allows us to order the Enqueue before some of them, even though it never succeeds on the central queue. The key is to ensure that Enqueues are eliminated only after sufficient aging.

To understand how the aging process works, consider the execution of Figure 2 on the queue structure shown in Figure 1. Figure 2 describes the time intervals of the operations depicted in Figure 1, starting from an empty queue. As can be seen, in this execution, first 1 is enqueued into an empty queue, then concurrent Enqueue attempts of $2, 3$ and 4 begin. The Enqueue of 4 succeeds in adding 4 to the queue, but causes the attempts to enqueue 2 and 3 to fail, so they back off; the queue now contains 1 followed by 4. At this point two Dequeues are started. The first successfully takes the value 1 from the queue, causing the second to fail, which therefore backs off. After 1 is dequeued, the backed-off Dequeue can eliminate with the Enqueue of 2 even though the head of the queue at this point contains 4. This is because the Enqueue of 2 has waited long enough so that all values enqueued completely before it, namely 1, are no longer in the queue. We can thus order the Enqueue of 2 after the Enqueue of 1 and before the Enqueue of 4, even though no process has yet dequeued 4 from the central queue.

2.1 The transformed central queue

How can a Dequeue detect that an Enqueue has aged sufficiently? One approach is to enhance the central queue with additional operations that allow a Dequeue that wishes to eliminate with an Enqueue to determine that all items inserted by Enqueue operations that completed before the candidate Enqueue operation began have already been dequeued. We now describe an abstract *counting queue*, which supports detection of aging for elimination. Most lock-free queue implementations (including MS-queue) can be easily adapted to implement the required semantics.

A counting queue provides EnqueueAttempt and DequeueAttempt operations, with the same semantics as Enqueue and Dequeue, except that they can return a special value "fail" in case of interference from concurrent operations. It also provides NumDeqs and NumEnqs operations, which report how many Dequeue and Enqueue operations respectively have succeeded so far. A straightforward transformation of the MS-queue to provide these operations is outlined in an appendix.

2.2 The Elimination Queue in Detail

The data structures used in our example code are shown at the top of Figure 3. The node_t type contains a value to be Enqueued and a serial number. In our presentation, nodes serve two purposes. First, they are passed to EnqueueAttempt to communicate the value to be enqueued to the central queue implementation. Depending on the implementation, the central queue may also use the node, and the node may include other fields not shown here. The other purpose of the node is for elimination; the value is the value to be passed from an eliminated Enqueue operation to the corresponding eliminated Dequeue operation, and the serial number is used to determine when it is safe to eliminate an Enqueue-Dequeue pair, as explained in more detail below. A FIFO queue (type queue_t) consists of a counting queue and an elimination array. We assume two "special" values of type pnode_t: "EMPTY" and "DONE", which can be distinguished from "real" node pointers. These values might be values that cannot be node addresses (for example due to alignment assumptions), or two special nodes can be allocated for this purpose.

An Enqueue operation begins by determining how many Enqueue operations have already completed on the central queue. This information is used to determine when the Enqueue has aged sufficiently to allow elimination with a Dequeue operation. The Enqueue operation then allocates a

```
structure node_t {value: valtype, seq: uint}
structure ptrctr_t {node: pointer to node_t, ver: uint}
structure Queue_t {Q: counting_queue_t,
                   Collisions: array of ptrctr_t}

Enqueue(Q: pointer to Queue_t, value: valtype)

uint seen_tail = NumEnqs(Q)
pointer to node_t node = new_node(value)
loop
   if DecideWhetherToAccessQueue() and
                 EnqueueAttempt(Q, node) then
      return
   else
      if TryToEliminateEnqueue(Q, node,
                                  seen_tail) then
         return
      end if
   end if
end loop

TryToEliminateEnqueue(Q: pointer to Queue_t,
        pointer to node_t, seen_tail: uint):boolean

node→seq = seen_tail;
i = random(collision_array_size)
⟨colnode,ver⟩ = Q→Collisions[i]
if colnode == EMPTY then
   if CAS(&Q→Collisions[i], ⟨EMPTY,ver⟩,
                              ⟨node,ver+1⟩) then
      ShortDelay()
      colnode = Q→Collisions[i].node
      if (colnode == DONE) or
         (not CAS(&Q→Collisions[i], ⟨colnode,ver+1⟩,
                              ⟨EMPTY,ver+1⟩)) then
         Q→Collisions[i] = ⟨EMPTY,ver+1⟩
         return true
      end if
   end if
end if
return false
```

Figure 3: Data structures and Enqueue operation

```
Dequeue(Q: pointer to Queue_t,
         pvalue: pointer to valtype):boolean

loop
   if DecideWhetherToAccessQueue() then
      res = DequeueAttempt(Q, pvalue)
      if res == SUCCESS then
         return true
      else if res == QUEUE_EMPTY then
         return false
      end if
   else
      if TryToEliminateDequeue(Q, pvalue) then
         return true
      end if
   end if
end loop

TryToEliminateDequeue(Q: pointer to Queue_t,
         pvalue: pointer to valtype):boolean

seen_head = NumDeqs(Q)
i = random(collision_array_size)
⟨node,ver⟩ = Q→Collisions[i]
if node ∉ {EMPTY, DONE} then
   if node→seq ≤ seen_head then
      *pvalue = node→value
      if CAS(&Q→Collisions[i],⟨node,ver⟩,
                              ⟨DONE,ver⟩) then
         return true
      end if
   end if
end if
return false
```

Figure 4: Dequeue operation

node initialized with the value to be enqueued, and then repeatedly attempts either to Enqueue the value using the central queue, or to find a Dequeue operation with which to eliminate, depending on guidance from the heuristic implemented by DecideWhetherToAccessQueue. The operation returns when it succeeds using either approach. The structure of a Dequeue operation is similar. We defer a detailed discussion of node management to a full version of the paper.

It remains to describe the elimination mechanism. TryToEliminateEnqueue stores the the number of previous Enqueues on the central queue (recorded earlier) in the thread's node, and then attempts to find an empty slot in the elimination array. It does this by choosing a slot at random, and then determining if the slot contains EMPTY. If not, the elimination attempt fails. Otherwise, the thread attempts to replace the EMPTY value with a pointer to its node using compare-and-swap (CAS). If this CAS fails, then the elimination attempt fails. Otherwise, the Enqueue has installed its node into the elimination array, so it waits for a short time, hoping that a Dequeue finds the node and eliminates. If it does so, it changes the node pointer to DONE, as explained below. Therefore, the Enqueue operation can detect elimination by checking to see if the node pointer has been changed to DONE. If so, the Enqueue has been eliminated, so it can return. Otherwise, the thread uses CAS to attempt to change its entry in the elimination array back to EMPTY, so that it can return to the main Enqueue procedure to retry. If this CAS fails, it can only be because a Dequeue operation has changed the node to DONE, so again the Enqueue is successfully eliminated in this case.

When a Dequeue operation attempts to eliminate, it chooses a slot in the elimination array at random, and checks to see if there is an Enqueue waiting to eliminate in that slot (if the node pointer is not DONE or EMPTY, then there is an elimination attempt in progress by an Enqueue operation). If not, the attempt to eliminate fails. Otherwise, the Dequeue attempts to change the node pointer to DONE, indicating to that Enqueue operation that the elimination was successful. If it does, it simply returns the value from the node. However, as explained earlier, it is not always safe to eliminate with an Enqueue operation. The Dequeue operation that hopes to eliminate must first confirm that the number of Dequeues performed on the central queue is at least the number of Enqueues performed before the candidate Enqueue operation began. This check is performed by comparing the number recorded in the node by the Enqueue to the result of calling NumDeqs on the central queue.

Finally, in order to avoid the ABA problem [15], pointers in the elimination array are augmented with version numbers, which are incremented each time a node is installed into the elimination array. This avoids the following potential problem. A Dequeue could read a pointer from the elimination array and determine that the Enqueue is safe to eliminate. However, before the Dequeue performs its CAS to "claim" the value to return, the node could be removed from the elimination array, recycled, and reused in the elimination array by another Enqueue operation that is *not* yet safe to eliminate, and the Dequeue's CAS could succeed, thereby causing elimination with the ineligible Enqueue.

2.3 Heuristics for Elimination

Our FIFO elimination technique allows threads to choose dynamically between accessing the central queue and attempting to eliminate. Our presentation assumes a function DecideWhetherToAccessQueue, which returns true if we should attempt the operation on the central queue, and false if we should try to eliminate. This function can implement any heuristic choice, and may additionally incorporate traditional backoff techniques by delaying for some time before returning. However, the heuristic should always eventually attempt to complete on the central queue in order to ensure lock-freedom. Below we discuss considerations in designing such heuristics, and in seeking elimination partners.

Under high load, elimination is preferable, because using the nonscalable central queue will result in poor performance. Under low load, finding an eligible operation with which to eliminate may take too long, and the number of "aged" enqueues is most likely low. Furthermore, because load is low, accessing the central queue should be fast.

The DecideWhetherToAccessQueue heuristic is in our implementation is designed to use elimination as a backoff mechanism for the central queue. Thus, every operation alternates between attempting to complete on the central queue and attempting to eliminate some number of times. As the experiments presented in Section 4 show, on a Niagara-based Sun Fire^{TM}T200 system, the resulting algorithm performs comparably at low load and outperforms MS-queue with backoff at high load because the eliminated operations relieve contention on the central queue. We expect that the margin of improvement would continue to increase with increasingly larger machines. However, to achieve a truly scalable queue, the following factors must be considered.

First, we must consider the scalability of the elimination mechanism itself. An important consideration is locality: it is preferable to eliminate with an operation that is "nearby" in the machine in order to avoid communication bottlenecks. In our experiments, local groups of processors shared (logical) elimination arrays that were allocated in local memory (in architectures having a notion of memory that is local to processors), rather than choosing slots at random in a single elimination array, as presented in our pseudocode above. In experiments artificially constructed to allow maximum elimination and to always avoid the central queue, we have determined that our elimination mechanism scales up very well on a 144-core Sun Fire^{TM}E25K system. However, our elimination-as-backoff implementation did not exhibit such good scalability on this machine. We speculate on the reasons for this below, and discuss possible remedies.

If every operation always accesses the central queue before attempting to eliminate, as it does in an elimination-as-backoff scheme, then throughput will not scale in large machines under high load. Ideally, in order to achieve true scalability, we would like to avoid the need for synchronizing on the central queue. This can happen only if the central queue remains empty (so that NumEnqs and NumDeqs operations access only cached read-only state of the central queue). Therefore, if we wish to achieve scalable performance under high load in arbitrarily large machines, we must use a heuristic that attempts to make the central queue empty (for example by favoring elimination for Enqueues more than for Dequeues until the central queue is empty), and then causes all operations to eliminate. Of course, in reality it will not always be possible to achieve this ideal, but these considerations may be useful in designing effective heuristics for adapting to high load. We have not had time to evaluate these ideas but we hope to do so for a full version of this paper. Based on our experience with the artificial experiments discussed above, we are optimistic that an implementation based on these techniques will significantly outperform MS-queue in larger machines.

A variety of strategies for adapting to load on the elimination array are possible. For example, backoff on the elimination array in both time and space can be considered, as in [7, 21]. In such arrangements, operations attempt to eliminate on a location chosen at random from a sub-range of the elimination array that grows or shrinks based on perception of the load. Such approaches can dynamically "spread the load" over relevant parts of the elimination array so that operations seeking to eliminate are dense enough to find each other, but spread out enough that they do not interfere excessively with other elimination attempts. Dynamic backoff techniques can also be used to control how long an Enqueue waits in the elimination array for a partner Dequeue operation, etc. We have not explored the space of possible adaptation techniques in detail; this paper provides a proof of concept for using elimination to implement scalable FIFO queues, but not an exhaustive study of possible ways to apply the technique.

3. OVERVIEW OF CORRECTNESS PROOF

Linearizability [9] requires that, for every execution of a set of threads invoking operations on the queue, there exists some ordering of the set `Ops` of queue operations in the execution such that a) the ordering represents a legal FIFO queue history, and b) the ordering respects the concurrent partial order of the operations in the execution.

To prove the first part, we first need to specify the set of all legal FIFO queue histories. Furthermore, in order to reason about the behavior of the algorithm, we must precisely state the semantics of the counting queue used in the algorithm. Our proof uses *augmented queue histories* which generalize both FIFO queue and counting queue histories.

Definition 1: An *augmented queue history* $H = \langle \texttt{OpId}, \texttt{Type}, \texttt{Val}, \texttt{Ret}, \texttt{Elim?} \rangle$[2] of length n consists of the following functions:

`OpId`:	$0..n-1 \rightarrow$ `Ops`
`Type`:	$0..n-1 \rightarrow \{E, D, ND, NE\}$
`Elim?`:	$0..n-1 \rightarrow$ boolean
`Val`:	$0..n-1 \rightarrow$ `valtype`
`Ret`:	$0..n-1 \rightarrow \{FAIL, EMPTY, OK\} \cup$ `valtype` $\cup$ integer

A history represents information about each of a sequence of operations, such as its type (Enqueue(Attempt),

[2]We sometimes refer to the component functions of a history H by name without explicitly referring to H where this will not cause confusion.

Dequeue(Attempt), NumEnqs, or NumDeqs), which high-level operation it is associated with, whether it is eliminated, and what parameter and return values it has.

The following definitions express the number of operations before position j in history H that satisfy predicate P, and the index in H of the jth operation satisfying P.

Definition 2: For a history H of length n, a predicate P, and an integer $j, 0 \leq j \leq n$, $\#(H, P, j) \equiv |\{k | 0 \leq k < j \wedge P(j)\}|$.

Definition 3: For a history H of length n, a predicate P, and an integer $j, 0 \leq j < n$, $ndx(H, P, j) \equiv \min k | \#(H, P, k+1) = j$.

Definition 4: An augmented queue history H is *legal* if it satisfies the $\texttt{LEGALAUG}?(H)$ (see Figure 5).

$\texttt{LEGALFIFO}?$ and $\texttt{LEGALCNTQ}?$ (Figure 5) restrict $\texttt{LEGALAUG}?$ to define legal FIFO and counting queue histories.

Our proof begins with a representation E of an arbitrary finite and complete[3] execution of our algorithm. E contains one event for each low-level atomic action in the algorithm, including invocations and responses of implemented Enqueue and Dequeue operations, as well as invocations and responses of counting queue operations invoked by the algorithm. Each event in E is labeled with the high-level Enqueue or Dequeue operation to which it belongs. The partial order $\prec_E$ over $\texttt{Ops}$ is defined such that $O1 \prec_E O2$ holds iff $O1$ completes before O2 begins in E. Functions $\texttt{Type}_E$, $\texttt{Elim}?_E$, etc. map each high-level operation to its type (E or D), whether it was eliminated, etc.

Because Q is a linearizable counting queue, there exists an ordering H_0 of the EnqueueAttempt, DequeueAttempt, NumDeqs, and NumEnqs operations in E such that a) $\texttt{LEGALCNTQ}?(H_0)$ holds, and b) if a counting queue operation O_1 completes before another counting queue operation O_2 begins in E, then O_1 is ordered before O_2 by H_0. Note that $\texttt{LEGALCNTQ}?(H_0)$ does not constrain the $\texttt{OpId}$ component of H_0; we set the $\texttt{OpId}$ of each operation in H_0 to be the high-level Enqueue or Dequeue operation to which the counting queue operation belongs in E. We also choose $\texttt{Elim}?_{H_0}$ so that it maps *all* operations to *false*. Note that each high-level Enqueue operation in E has one NumEnqs operation in H_0, each noneliminated Enqueue (resp., Dequeue) operation has exactly one successful EnqueueAttempt (resp., DequeueAttempt) in H_0, each eliminated Dequeue operation has at least one NumDeqs operation in H_0, etc.

In the main part of our proof, we inductively construct a sequence of augmented queue histories H_i, $i = 1, 2, \ldots$, where H_{i+1} is produced by adding a pair of eliminated operations from E into H_i. The final history H_f in this sequence contains all high-level operations.

We show that each H_i we construct in this way is a legal augmented queue history, and in addition that it satisfies $\texttt{GOODORDER}?(H_i, E)$ (see Figure 5). This latter property is used to aid in our induction, and to show that the order of all high-level operations included in H_i respects the partial order $\prec_E$ (see Conjunct (7) of $\texttt{GOODORDER?}$).

[3]A *complete* execution is one in which every operation invocation has a matching response. Proving linearizability for all complete executions suffices to prove linearizability for all histories because any history can be extended to a complete history.

Finally, we show that we can remove all extraneous counting queue operations, such as NumEnqs and NumDeqs operations, and failed EnqueueAttempt and DequeueAttempt operations from H_f, and the resulting sequence F, which contains exactly the set of high-level operations from E, is a legal FIFO queue history, proving the following theorem.

Theorem 1: Our algorithm is a linearizable implementation of a FIFO queue.

Below we present some key definitions and properties of the proof, and describe some interesting aspects of the proof. $\texttt{LEGALAUG}?(H_0)$ follows from the fact that Q is a linearizable counting queue. It is easy to prove that $\texttt{GOODORDER}?(H_0, E)$ also holds because H_0 does not contain any eliminated operations and because Q is linearizable.

To construct H_{i+1} from H_i, we choose an eliminated Enqueue operation—call it $O1$—that we have not chosen previously. We then choose two indexes $epos(H_i, O1)$ and $dpos(H_i, O2)$, which indicate where we will insert the eliminated Enqueue operation and its partner Dequeue operation, respectively, in H_i to get H_{i+1}. To explain how we choose these indexes and how we use them to construct H_{i+1} from H_i, we present several definitions, and then explain the intuition behind them.

Definition 5: For an eliminated Enqueue operation $O1$, $ne(O1, H)$ is the (unique) value j, $0 \leq j < length(H)$ satisfying $\texttt{Type}_H(j) = NE \wedge \texttt{OpId}_H(j) = O1$.

Definition 6: For an eliminated Dequeue operation $O1$, $lnd(O1, H)$ is the largest value j, $0 \leq j < length(H)$ satisfying $\texttt{Type}_H(j) = ND \wedge \texttt{OpId}_H(j) = O1$.

Definition 7: $neepos(H, j) \equiv ndx(H, \texttt{SE}, \#(H, \texttt{SD}, j) + 1) - 1$.

Definition 8: The *partner* function[4] maps each eliminated Enqueue or Dequeue operation to its elimination partner.

Definition 9: We consider three cases in defining $epos(H, O1)$ and $dpos(H, O1)$ for an eliminated Enqueue operation $O1$.

Case 1: $pqsize(H, lnd(partner(O1), H)) > 0$. In this case, $epos(H, O1)$ is defined to be $neepos(H, lnd(partner(O1), H))$ and $dpos(H, O1)$ is defined to be $lnd(partner(O1), H)$.

Case 2: $pqsize(H, lnd(partner(O1), H)) \leq 0 \wedge lnd(partner(O1), H) < ne(O1, H)$. In this case, $epos(H, O1)$ and $dpos(H, O1)$ are both defined to be $ne(O1, H)$.

Case 3: $pqsize(H, lnd(partner(O1), H)) \leq 0 \wedge lnd(partner(O1), H) > ne(O1, H)$. In this case, $epos(H, O1)$ and $dpos(H, O1)$ are both defined to be $lnd(partner(O1), H)$.

The following shows how H_{i+1} is constructed from H_i.

Definition 10: For an eliminated Enqueue operation $O1$ in E but not in H_i, let $O2 = partner(O1)$, and define H_{i+1} as follows. For each $f \in \{\texttt{OpId}, \texttt{Type}, \texttt{Elim}?, \texttt{Val}, \texttt{Ret}\}$:

- if $j \in [0, epos(H_i, O1)]$, then $f_{H_{i+1}}(j) = f_{H_i}(j)$.
- if $j \in [epos(H_i, O1) + 2, dpos(H_i, O1) + 1]$, then $f_{H_{i+1}}(j) = f_{H_i}(j - 1)$.
- if $j \in [dpos(H_i, O1)+3, length(H_i)+2)$, then $f_{H_{i+1}}(j) = f_{H_i}(j - 2)$.

[4]In the full proof, we prove that this function exists.

$\texttt{SE}(H,j) \equiv \texttt{Type}(j) = E \ \wedge\ \texttt{Ret}(j) = \mathit{OK}$ $\qquad$ $\texttt{CSE}(H,j) \equiv \texttt{SE}(H,j) \ \wedge\ \neg\texttt{Elim?}(j)$

$\texttt{SD}(H,j) \equiv \texttt{Type}(j) = D \ \wedge\ \texttt{Ret}(j) \in \texttt{valtype}$ $\qquad$ $\texttt{CSD}(H,j) \equiv \texttt{SD}(H,j) \ \wedge\ \neg\texttt{Elim?}(j)$

$\texttt{LEGALAUG?}(H) \equiv \forall j, 0 \le j < \mathit{length}(H) ::$

(1) $(\texttt{Type}(j) = E \ \Rightarrow\ \texttt{Val}(j) \in \texttt{valtype} \ \wedge\ \texttt{Ret}(j) \in \{\mathit{FAIL}, \mathit{OK}\}) \ \wedge$

(2) $(\texttt{Type}(j) = D \ \Rightarrow\ \texttt{Ret}(j) \in \{\mathit{FAIL}, \mathit{EMPTY}\} \ \cup\ \texttt{valtype}) \ \wedge$

(3) $(\texttt{Type}(j) = \mathit{PT} \ \Rightarrow\ \texttt{Ret}(j) = \#(H, \texttt{CSE}, j)) \ \wedge$

(4) $(\texttt{Type}(j) = \mathit{PH} \ \Rightarrow\ \texttt{Ret}(j) = \#(H, \texttt{CSD}, j)) \ \wedge$

(5) $(\texttt{Type}(j) = D \ \wedge\ \texttt{Ret}(j) = \mathit{EMPTY} \ \Rightarrow\ \#(H, \texttt{SD}, j) \ge \#(H, \texttt{SE}, j)) \ \wedge$

(6) $(\texttt{Type}(j) = D \ \wedge\ \texttt{Ret}(j) \in \texttt{valtype} \ \Rightarrow\ \#(H, \texttt{SD}, j) < \#(H, \texttt{SE}, j) \ \wedge\ \texttt{Ret}(j) = \texttt{Val}(\mathit{ndx}(H, \texttt{SE}, \#(H, \texttt{SD}, j) + 1)))$

$\texttt{LEGALCNTQ?}(H) \equiv \texttt{LEGALAUG?}(H) \ \wedge\ (\forall j : 0 \le j < \mathit{length}(H) :: \texttt{Type}(j) \in \{E, D\} \ \Rightarrow\ \neg\texttt{Elim?}(j))$

$\texttt{LEGALFIFO?}(H) \equiv \texttt{LEGALAUG?}(H) \ \wedge\ (\forall j : 0 \le j < \mathit{length}(H) :: \texttt{Type}(j) \in \{E, D\} \ \wedge\ \texttt{Ret}(j) \ne \mathit{FAIL})$

$\mathit{pqsize}(H,j) \equiv \#(H, \texttt{CSE}, j) - \#(H, \texttt{CSD}, j)$ $\qquad$ $\mathit{aqsize}(H,j) \equiv \#(H, \texttt{SE}, j) - \#(H, \texttt{SD}, j)$

$\texttt{GOODORDER?}(H, E) \equiv \forall j : 0 \le j < \mathit{length}(H), \forall O1, O2 \in \texttt{Ops} ::$

(1) $\mathit{pqsize}(H,j) \ge 0 \ \wedge$

(2) $\mathit{aqsize}(H,j) \ge 0 \ \wedge$

(3) $\mathit{pqsize}(H,j) \le \mathit{aqsize}(H,j)$

(4) $(\mathit{pqsize}(H,j) = 0 \ \Rightarrow\ (\mathit{aqsize}(H,j) = 0 \ \vee\ (\texttt{Type}_H(j-1) = E \ \wedge\ \texttt{Elim?}_H(j-1)))) \ \wedge$

(5) $(\texttt{Type}_E(O1) = E \ \wedge\ \texttt{Elim?}_E(O1) \ \wedge\ \mathit{pqsize}(H, \mathit{lnd}(\mathit{partner}(O1), H)) > 0 \ \Rightarrow$
$\mathit{neepos}(H, \mathit{lnd}(\mathit{partner}(O1), H)) \ge \mathit{ne}(O1, H)) \ \wedge$

(6) $(O1 \prec_E O2 \ \wedge\ \texttt{Type}_E(O1) = E \ \wedge\ \texttt{Elim?}_E(O1) \ \Rightarrow$
$(\forall k : k \in [0, \max(\mathit{ne}(O1,H), \mathit{lnd}(\mathit{partner}(O1), H)))] :: \texttt{OpId}_H(k) \ne O2) \ \wedge$

(7) $(O1 \prec_E O2 \ \wedge\ \texttt{OpId}_H(j) = O1 \ \Rightarrow\ (\forall k : k \in [0, j] :: \texttt{OpId}_H(k) \ne O2))$

Figure 5: Definitions of legal histories for augmented, counting, and FIFO queues, and additional properties required for inductive linearizability proof.

For $j = \mathit{epos}(H_i, O1) + 1$, $\texttt{OpId}_{H_{i+1}}(j) = O1$, $\texttt{Type}_{H_{i+1}}(j) = E$, $\texttt{Elim?}_{H_{i+1}}(j) = \mathit{true}$, $\texttt{Val}_{H_{i+1}}(j) =$ value enqueued by $O1$, and $\texttt{Ret}_{H_{i+1}}(j) = \mathit{OK}$.

For $j = \mathit{dpos}(H_i, O1) + 2$, $\texttt{OpId}_{H_{i+1}}(j) = \mathit{partner}(O1)$, $\texttt{Type}_{H_{i+1}}(j) = D$, $\texttt{Elim?}_{H_{i+1}}(j) = \mathit{true}$, and $\texttt{Ret}_{H_{i+1}}(j) =$ value enqueued by $O1$.

The intuition behind the above definitions is that we wish to insert the eliminated Enqueue operation and its partner Dequeue operation into H_i to produce H_{i+1} in such a way that it is still a legal augmented history, and so that we can prove that the order of operations in H_{i+1} still respects $\prec_E$.

We begin with the last NumDeqs operation of the eliminated Dequeue operation. If Q is empty after that operation (i.e., the number of noneliminated successful Dequeue operations before it is the same as the number of noneliminated successful Enqueue operations before it), then we can infer from Conjunct (4) of $\texttt{GOODORDER?}(H_i, E)$ that the abstract queue is also empty at that point (i.e., the number of *all* successful Dequeue operations before that point is equal to the number of *all* successful Enqueue operations before that point). Clearly we can insert the eliminated Enqueue operation immediately followed by the eliminated Dequeue operation at a position at which the abstract queue is empty. We also have to construct H_{i+1} in such a way that we can later prove that the constructed order respects the partial order over operations in E. It turns out that if the eliminated Enqueue operation's NumEnqs operation is ordered after the eliminated Dequeue operation's NumDeqs operation, then we would not be able to do so if we inserted the pair after the Dequeue's last NumDeqs. However, because the Enqueue's NumEnqs operation returns a value that is at most the value returned by the Dequeue's last NumDeqs operation, we can prove that if the Enqueue's NumEnqs operation is ordered after the Dequeue's NumDeqs operation, then the counting queue and the abstract queue are also empty at the Enqueue's NumEnqs operation, so we can order the pair there instead (see Case 2 for the above definition), and this allows us to prove the necessary ordering property.

The most interesting case is the one in which the counting queue (and therefore the abstract queue) is *not* empty at the Dequeue's last NumDeqs. In this case, we must insert the eliminated Enqueue operation earlier in the history in order to maintain a correct FIFO ordering of operations in the history (see Case 1 above). The definition essentially says *insert the eliminated Enqueue operation immediately before the* $(n+1)$*-st Enqueue operation in* H_i*, where* n *is the number of successful Dequeue operations before the place at which we will insert the eliminated Dequeue operation.* This clearly preserves correct FIFO ordering of successful Enqueue and Dequeue operations.

There are two main challenges. First, we must ensure that a Dequeue operation that returns EMPTY in H_i still does so correctly in H_{i+1} (i.e., Conjunct (5) of $\texttt{LEGALAUG?}$ is preserved). If we insert the eliminated operation pair into H_i in such a way that an EMPTY-returning Dequeue operation falls between them, then although Q is still empty immediately before that operation, the abstract queue would not be, as we would have inserted a successful Enqueue operation before it, with the matching Dequeue operation after it. We show in our proof that whenever we are using Case 1 of Definition 9, there is no Dequeue operation that returns $EMPTY$ between the inserted Enqueue and Dequeue operations.

The other challenge is ensuring that we construct the histories H_i in such a way that we can prove that the ordering of Enqueue and Dequeue operations in H_f respects the partial order over these operations in E (i.e., the construction of each H_i preserves Conjunct (7) of `GOODORDER`?).

This is mostly straightforward because we can easily show how most high-level operations are ordered relative to counting queue operations invoked either by the high-level operation or by its elimination partner; in the latter case, it is usually straightforward to relate the ordering of the partner's counting queue operation to a counting queue operation of the high-level operation in question, because of the way the elimination mechanism forces two eliminating operations to synchronize with each other on the elimination array.

However, it is significantly more complicated when we use Case 1 of Definition 9 because the Enqueue operation is not explicitly ordered near any counting queue operation that either it or its partner invokes. A key property required to prove the necessary ordering properties is that an Enqueue operation eliminated according to Case 1 of Definition 9 is ordered *after* its NumEnqs operation. This is nontrivial because we have insisted that, for scalability, elimination must be possible when the central queue is empty, which allows our elimination heurisitics to avoid access to the nonscalable central queue under heavy load. Because of this requirement, we cannot show that the Enqueue is ordered after its NumEnqs operation by a simple counting argument. Instead, we strengthen our induction in such a way that we can infer that an Enqueue's NumEnqs operation is ordered before the Enqueue operation itself (see Conjunct (5) of `GOODORDER`?).

The most challenging parts of this proof are choosing where eliminated operations are inserted, and strengthening the induction appropriately to allow the induction to go through and to imply that all operations added so far respect the partial order over operations in E. Most of the proof consists of reasonably straightforward—though tedious—counting arguments, usually broken into several cases depending on where the j under consideration falls relative to the inserted operations. Generally the cases in which j is between the inserted operations are the most challenging, as H_{i+1} is identical to H_i up to position $epos(i, e)$, and for positions after $dpos(i, e)$, we have inserted one Enqueue and one Dequeue before it, which generally cancel each other out, making the counting arguments trivial.

4. PERFORMANCE

The experiments described below were conducted on a Sun Fire$^{\text{TM}}$T200 Niagara-based multiprocessor. Niagara is a single-chip interleaved multithreaded multiprocessor (CMT) with eight computing cores, each with four hardware strands.

We implemented our algorithms in the *C++* programming language, compiled by the Sun *CC* compiler 5.5, using the flags `-xO5 -xarch=v8plusa` and run with the Solaris$^{\text{TM}}$10 (SunOS$^{\text{TM}}$5.10) operating system. To reduce variability due to operating system activity unrelated to our experiments, each thread was bound to a particular hardware strand. In order to avoid performance problems due to false cache line sharing, important data structures (Queue nodes, elimination slots, etc.) were always allocated in separate cache-line-sized chunks of memory.

We compared the performance of the MS-Queue algorithm to an elimination-enabled variation of itself. In both cases, each thread repeatedly performs either an Enqueue or a Dequeue operation, choosing randomly between the two. For each operation, the thread first tries to access the shared MS-queue directly. Upon failure, unlike the MS-Queue algorithm, the elimination-enabled algorithm attempts to find an elimination partner in the elimination array.

We conducted several experiments to optimize the backoff heuristics for both algorithms. For the MS-Queue, we varied `min-` and `max-backoff` over wide ranges. Because we were most interested in scalability under heavy load, our optimization process selected values that yielded the best performance for the largest (32-thread) test case without unduly damaging performance for smaller test cases. (We have not yet experimented with heuristics that adapt these parameters to current load.)

For the elimination algorithm, we varied several parameters, including:

`elimination-group-size` number of threads that share a set of elimination slots.

`elimination-slots-per-group` number of elimination slots shared by each group of elimination-group-size threads.

`eliminiation-n-dequeuer-tries` number of times a dequeuer will select and examine a new elimination slot before going back to access the central queue.

`elimination-wait-time` how long an enqueuer waits for an elimination partner (dequeuer) to arrive after putting its item into an elimination slot.

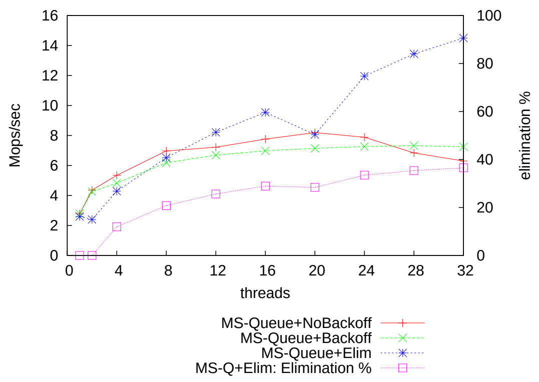
The best results were achieved with a group size of 16, 4 slots per group, `elimination-n-dequeuer-tries` set to 3, and `elimination-wait-time` set to 3000[5].

Figure 6 shows the throughput for MS-Queue, with and without backoff, and for the elimination algorithm, for different numbers of concurrently operating threads. The Enqueue/Dequeue ratio is 30/70 in the left graph and 49.5/51.5 in the right graph. Both graphs contain a fourth line showing the elimination percentage achieved by the elimination algorithm (to be read using the right-hand y axis).

With small numbers of threads, the performance of all algorithms is similar, with a slight advantage to the traditional MS-Queue. Again, we optimized our parameters for best performance at high load without ruining performance for smaller numbers of threads. In principle, the heuristic that decides whether to eliminate or to access the central queue directly can mimic the MS-queue at low loads, thereby closing this gap, but we have not yet experimented with heuristics that attempt to adapt to load.

As the number of threads increase, so too do the number of eliminations, positively affecting the throughput of the elimination-enabled algorithm. On an eight-core, 32-strand test, the elimination queue algorithm beat the throughput of MS-queue by about a factor of two for the 30/70 test, and by about 40% for the 49.5/51.5 test. It is not surprising that the 30/70 test yields better results, since when the

[5]The elimination-wait-time is expressed as the number of iterations of a short delay loop.



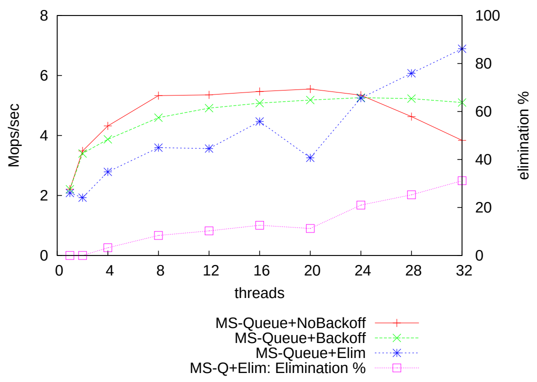


Figure 6: Throughput/Elim % graphs. Enqueue/Dequeue ratios are $30/70$ (left) and $49.5/51.5$ (right).

queue is short, the "aging period" is correspondingly short, so Enqueues become eligible for elimination more quickly.

These results suggest that the throughput of of the elimination queue will continue to increase with higher levels of concurrency, while MS-queue clearly will not. Furthermore, even at the levels of contention in the experiments presented here, we have several ideas for achieving greater levels of elimination in the elimination queue, thereby increasing the improvement of our algorithm over MS-queue. Therefore, our algorithm is promising for the very large multiprocessor machines of the future, as well as emerging CMT chips.

5. CONCLUDING REMARKS

We have shown that, with care, the elimination technique already known to be useful for making stacks, pools and counters scalable, can also be applied to FIFO queues. Our preliminary performance results indicate that this technique improves performance even with relatively small numbers of concurrent threads and we are optimistic that the improvement due to elimination will only increase with larger numbers of concurrent threads.

Future work includes evaluating different heuristics for deciding whether to access the central queue or to eliminate in order to effectively adapt to a range of loads.

Acknowledgements:. We are grateful to Victor Luchangco for useful discussions, to Ron Larson for his help with access to the SunFire E25K machine and to the Java Performance team at Sun for access to the SunFire T200 machine used for the performance experiments described in Section 4.

6. REFERENCES

[1] A. Agarwal and M. Cherian. Adaptive backoff synchronization techniques. In *Proceedings of the 16th International Symposium on Computer Architecture*, pages 396–406, May 1989.

[2] W. Aiello, C. Busch, M. Herlihy, M. Mavronicolas, N. Shavit, and D. Touitou. Supporting increment and decrement operations in balancing networks. *Lecture Notes in Computer Science*, 1563:393–403, 1999.

[3] D. Dolev and N. Shavit. Bounded concurrent time-stamping. *SIAM J. Comput.*, 26(2):418–455, 1997.

[4] J. Goodman, M. Vernon, and P. Woest. Efficient synchronization primitives for large-scale cache-coherent shared-memory multiprocessors. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, pages 64–75. ACM Press, April 1989.

[5] A. Gottlieb, R. Grishman, C. Kruskal, K. McAuliffe, L. Rudolph, and M. Snir. The NYU ultracomputer - designing an MIMD parallel computer. *IEEE Transactions on Computers*, C-32(2):175–189, February 1984.

[6] A. Gottlieb, B. D. Lubachevsky, and L. Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Trans. Program. Lang. Syst.*, 5(2):164–189, 1983.

[7] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *SPAA '04: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 206–215. ACM Press, 2004.

[8] M. Herlihy, V. Luchangco, P. Martin, and M. Moir. Nonblocking memory management support for dynamic-sized data structures. *ACM Trans. Comput. Syst.*, 23(2):146–196, 2005.

[9] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[10] K. Hwang and F. A. Briggs. *Computer Architecture and Parallel Processing*. McGraw-Hill, Inc., 1990.

[11] E. Ladan-Mozes and N. Shavit. An optimistic approach to lock-free fifo queues. In *proceedings of the 18th International Conference on Distributed Computing (DISC)*, pages 117–131. Springer-Verlag GmbH, 2004.

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

[13] D. Lea. The java concurrency package (JSR-166). `http://gee.cs.oswego.edu/dl/concurrency-interest/index.html`.

[14] J. M. Mellor-Crummey. Concurrent queues: Practical fetch-and-ϕ algorithms. Technical Report Technical

Report 229, University of Rochester, November 1987.

[15] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC '96)*, pages 267–275, New York, USA, May 1996. ACM.

[16] M. Moir. Practical implementations of non-blocking synchronization primitives. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, pages 219–228, 1997.

[17] S. Prakash, Y.-H. Lee, and T. Johnson. Non-blocking algorithms for concurrent data structures. Technical Report 91–002, Department of Information Sciences, University of Florida, 1991.

[18] S. Prakash, Y.-H. Lee, and T. Johnson. A non-blocking algorithm for shared queues using compare-and-swap. *IEEE Transactions on Computers*, 43(5):548–559, 1994.

[19] W. N. Scherer and M. L. Scott. Nonblocking concurrent data structures with condition synchronization. In *Proceedings of the 18th International Symposium on DIStributed Computing*, pages 174–187, Berlin, Heidelberg, New York, October 2004. Springer.

[20] N. Shavit and D. Touitou. Elimination trees and the construction of pools and stacks. *Theory of Computing Systems*, 30:645–670, 1997.

[21] N. Shavit and A. Zemach. Combining funnels: a dynamic approach to software combining. *J. Parallel Distrib. Comput.*, 60(11):1355–1387, 2000.

[22] H. S. Stone. *High-performance computer architecture.* Addison-Wesley Longman Publishing Co., Inc., 1987.

[23] J. Stone. A simple and correct shared-queue algorithm using compare-and-swap. In *Proceedings of the 1990 conference on Supercomputing*, pages 495–504. IEEE Computer Society Press, 1990.

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

[25] J. Valois. Implementing lock-free queues. In *Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems*, pages 64–69, 1994.

APPENDIX

A. TRANSFORMING THE MS-QUEUE

The central queue should be lock-free in the following sense: if some operation takes a sufficient number of steps, then some operation completes *successfully* (an EnqueueAttempt or DequeueAttempt operation is *successful* if it does not return “fail”). Furthermore, EnqueueAttempt and DequeueAttempt should be wait-free: they should complete (successfully or not) in a bounded number of their own steps. These requirements prevent livelock on the central queue, and allow our algorithm to intervene and attempt elimination if an operation does not succeed on the central queue.

For concreteness, we now briefly explain how we modify the MS-queue algorithm to implement the required central queue. We assume that the reader is familiar with the MS-queue. The EnqueueAttempt and DequeueAttempt operations are obtained by taking one iteration of the retry loop in the corresponding operation in the MS-queue algorithm. Further, in order to facilitate the NumDeqs and NumEnqs operations, each successful EnqueueAttempt operation associates a serial number with the enqueued value, as follows. Each enqueued node contains a serial number, which is derived by adding one to the value in the node pointed to by Tail immediately before the new node is installed. It is straightforward to make this adaptation. Because MS-queue always contains at least one queue node, even when it is empty, the serial number of the most recently enqueued element is always available.

NumDeqs and NumEnqs are also straightforward. Roughly speaking, NumDeqs reads the Head pointer, and returns the serial number from the node it points to. (Recall that the first node is always a dummy node, so the serial number in it represents the number of dequeues performed so far.) NumDeqs must also use standard techniques to detect interference from concurrent operations and retry. Generally this involves rereading the Head pointer to ensure that it did not change while the contents of the node were read.

The NumEnqs operation is similarly straightforward, with one exception. It reads the Tail pointer and returns the serial number of the last node in the queue, which represents the number of Enqueue operations performed previously. However, recall that in the MS-queue algorithm, the Tail can sometimes “lag” the end of the queue by one node, so NumEnqs may have to perform the “helping” in the MS-queue algorithm in order to be sure that it obtains the serial number of the most recently enqueued element.

B. SEQUENCE AND VERSION NUMBERS

Bounded serial numbers. While we have explained our algorithm using unbounded serial numbers, in reality the serial number must of course be bounded. However, if we use 64-bit serial numbers, then they will not wrap around in the lifetime of any realistic system. Because nodes are not visible to other threads while they are being initialized, it is easy to implement lock-free serial numbers of arbitrary length. Bounded timestamps [3] can also be used, if applicable, because we only compare serial numbers.

Bounded version numbers. Similarly, version numbers used to avoid ABA in the elimination array are bounded and could in principle cause incorrect behavior. A variety of techniques for avoiding such behavior in practice are known, including a) using sufficient bits for the version number that wraparound cannot cause an error in practice [16], b) using bounded tags [16], and c) using memory management techniques to ensure that a node is not prematurely recycled in such a way that it can allow the ABA problem [8].