A Single-Enqueuer Wait-Free Queue Implementation

Matei David

Department of Computer Science, University of Toronto
matei@cs.toronto.edu

Abstract. We study wait-free linearizable Queue implementations in asynchronous shared-memory systems from other consensus number 2 objects, such as Fetch&Add and Swap. The best previously known implementation allows at most two processes to perform Dequeue operations. We provide a new implementation, when only one process performs Enqueue operations and any number of processes perform Dequeue operations. A nice feature of this implementation is the fact that both Enqueue and Dequeue operations take constant time.

1 Introduction

An asynchronous shared-memory distributed system provides the user with a collection of shared objects. Different systems might provide different types of shared objects, hence an algorithm written for one system might have to be completely rewritten to work in another system. A general way to make all algorithms written for a source system work in a target system is to use the objects of the target system in simulating every object of the source system. An *implementation* of an object consists of a set of procedures simulating the primitive operations of the implemented object, written using the objects in the target system.

The main tool determining whether objects of one type T' can be implemented from objects of another type T is the consesnsus hierarchy, introduced by Herlihy in [Her91] and refined by Jayanti in [Jay93]. If k is the consensus number of T, then it can be used to implement any other type in a system of at most k processes. Furthermore, if k' is the consensus number of type T' and $k < k'$, then there are objects of type T' which cannot be implemented from type T in a system of more than k processes.

There are some questions that the consensus hierarchy does not answer. If two types are on the same level k, it is not clear whether one type can implement the other in a system of more than k processes. Even for level 2, [Her91] leaves as an open problem whether Fetch&Add objects can be used to implement any other object whose type has consensus number 2 in a system of three or more processes.

The Queue is an important and well studied shared object type, used in many distributed algorithms. However, distributed systems usually provide lower-level types, such as Register, Fetch&Add and Compare&Swap, so, in general, one has to implement a Queue object from the available base types. We know that the Queue type has consensus number 2, and from Herlihy's results in [Her91], we know that wait-free Queue implementations exist for any number of processes in systems providing consensus number ∞ types, such as Compare&Swap. But some (old) systems only provide types with consensus number 2, such as Test&Set, Fetch&Add and Swap. To this date, it is an open problem whether any of these types can be used to implement a wait-free Queue in a system with three or more processes.

In [AWW93], Afek, Weisberger and Weisman consider the class *Common2* of commutative and overwriting read-modify-write types of consensus number 2, which includes most familiar types such as Test&Set, Fetch&Add and Swap. They show that any type in Common2 can be implemented in a wait-free manner from any consensus number 2 type in a system with any number of processes. By transitivity of wait-free implementations, their result implies that a wait-free Queue implementation exists from Common2 types if and only if such an implementation exists from any consensus number 2 type.

Let *Basic2* denote the set of types of consensus number 2 that can be implemented from types in Common2 in a system with any number of processes. The results of [AWW93] imply that using only Basic2 types in an algorithm carries with it the guarantee that the algorithm can be ported to any system providing types of consensus number 2. It is not known whether Queue is in Basic2.

However, some restricted Queue implementations exist. Herlihy and Wing present in [HW90] a non-blocking implementation of a Limited-Queue object shared by n processes from Fetch&Add and Swap objects. The Limited-Queue object type is similar to the Queue object type with the exception that Dequeue operations are not defined when the queue is in the empty state. Li gives a regular, unlimited, non-blocking Queue implementation in [Li01] and observes that the implementation in [HW90] is in fact a single-dequeuer wait-free Queue implementation. That is, if only one process is allowed to perform Dequeue operations, the implementation becomes wait-free, and the Queue is no longer limited.

In [Her91], Herlihy showed that in a system of n processes, any object can be implemented from Consensus objects shared by all n processes. Using ideas from Herlihy's universal construction, Li modifies the implementation in [HW90] and obtains in [Li01] a two-dequeuer wait-free Queue implementation from Common2 types. Furthermore, Li conjectures that there is no Queue implementation from Common2 types which would allow three processes to perform both Enqueue and Dequeue operations. In an attempt to narrow down the difficulty involved in implementing one such object, Li proposes a stronger conjecture: there is no three-dequeuer Queue implementation from Common2 types.

In both wait-free implementations, the code for the Enqueue procedures is very simple. However, the number of accesses to shared objects during Dequeue procedures is not bounded by any constant, i.e. it is wait-free but not bounded wait-free.

In this paper, we present a new wait-free Queue implementation from Common2 types, for one enqueuer process and any number of dequeuer processes. This disproves the stronger of Li's conjectures. In our single-enqueuer Queue implementation, the conceptually difficult part of the computation is done by the Enqueue procedure, and the Dequeue procedures are very simple. Unlike Li's implementations, our implementation is very time efficient, using at most three accesses to shared objects for both Enqueue and Dequeue procedures. Although the algorithm is simple, proving its correctness is complicated.

This paper is organized as follows. In Sect. 2, we briefly talk about our model of computation. In Sect. 3, we present our new single-enqueuer Queue implementation from Common2 objects. The main ideas for the proof of correctness are presented in Sect. 4. Section 5 contains a discussion of possible extensions of our algorithm. In particular, we give a scheme which would reduce the space requirements of our algorithm, and we talk about why our single-enqueuer Queue implementation cannot be extended to allow for two enqueuer processes in a manner similar to that in which Li extends the single-dequeuer Queue implementation in [HW90] to a two-dequeuer Queue implementation.

2 System Model

The system we consider is an asynchronous shared-memory distributed system. It consists of a number of processes and a collection of shared objects. Processes start from their initial state, execute deterministic sequential programs and communicate by accessing shared objects. During an atomic *step*, a process performs an operation on a certain shared object and receives a response from that object. In this setting, the crash failure of a process can be simulated by considering executions in which that process is no longer taking any steps. Between steps, processes can perform an arbitrary amount of local computation. This assumption captures the fact that process P cannot get any information about the computation of process P' except for what is conveyed by the operations P' performs on shared objects.

Each shared object has a type, an initial state and a set of rules specifying what operations on this object are available to each process in the system. The *type* of an object contains its sequential specification, which defines how that object reacts as operations are sequentially applied on it. In this paper, the Queue type supports only Enqueue and Dequeue operations, and the Queue object we are implementing allows every process to apply either Enqueue or Dequeue operations, but not both. Afek et al. show that no generality is lost in assuming that for every Common2 object O used in our implementation, every processes can perform on O every operation specified by O's type [AWW93].

The Queue implementation consists of one Enqueue procedure $E:\text{Enqueue}(x)$ for the enqueuer process E, and one Dequeue procedure $D:\text{Dequeue}$ for every dequeuer process D. The Enqueue procedure always returns the special value OK. The Dequeue procedure returns either a value retrieved from the Queue, or the special value ε in case the Queue is empty.

In a *run* R of the implementation, each process P starts from its initial state and sequentially executes access procedures of the form $P : OP$, completing one before starting the next. Given a run R, one can partition the subsequence of steps taken by any process P into contiguous blocks, such that each block contains the steps that P is taking while executing some access procedure. We define a *procedure instance* to be the set of steps in one such block. We say that a procedure instance by process P is *complete* if, after P executes the last step of this instance appearing in R, the access procedure contains only local computation before returning a result.

The only correctness condition we consider is *linearizability* [HW90], which states that, no matter how the steps in the execution of the access procedure $P\colon OP$ are interleaved with the steps in the executions of other access procedures by other processes, $P\colon OP$ has to appear to be atomic, occurring at some moment between its first and last steps, in a way that respects the sequential specification of the implemented object. Our Queue implementation is linearizable.

An implementation is *wait-free* [Her91] if every process will complete the execution of every access procedure within finitely many steps, regardless of the steps performed by other processes in the system and, in particular, regardless of whether other processes have crashed. An implementation is *b-bounded wait-free* if no access procedure requires more than b steps. Notice that bounded wait-freedom is a stronger condition than wait-freedom. Our Queue implementation is 3-bounded wait-free.

3 Algorithm

The first attempt to implement a Queue object for one enqueuer E and n dequeuers $D_1, \ldots D_n$ would probably be to use an array of Register objects to store the values in the Queue, together with a pair of head and tail pointers. E would add items in the array at the location indicated by the tail pointer, while dequeue processes would retrieve values from the location indicated by the head pointer. This does not work because several dequeue processes may try to read the same location, and there is no easy way for them to agree which one should return that value. A slightly more elaborate approach would be to use a Fetch&Add object for the head pointer, and have each dequeue procedure reserve a unique cell to read by a simple Fetch&Add(1) operation. This does not work because dequeue procedures might end up reading a cell before the enqueuer process has a chance to write something there. Afterward, if the enqueuer puts an element in that location, it might happen that no dequeue procedure will ever read the cell again, causing the enqueued element to simply vanish. We have been able to fix this situation by using Swap objects instead of Registers as the array cells, a design which allows the enqueuer process to detect and adapt to the situation in which a dequeuer has overtaken it.

The algorithm in Fig. 1 is our Queue implementation from Common2 objects and Registers, for one enqueuer process E and n dequeuer processes $D_1, \ldots, D_n$. We are using a one-dimensional array HEAD of Fetch&Increment objects, each initialized to 0, a two-dimensional array ITEMS of Swap objects, each initialized to $\perp$, and one Register ROW initialized to 0. The set of values that may be held by a cell of ITEMS is $V \cup \{\perp, \top\}$, where V is the set of values that may be enqueued. The two variables *tail* and *enq_row* are two persistent local variables of E, initialized to 0. Elements enqueued by E are written in consecutive cells on the row ROW of ITEMS. When E detects that it has been overtaken by a dequeue process, it starts using a fresh row. Dequeue processes read the active row of ITEMS from ROW and order themselves on a given row using HEAD. Both arrays HEAD and ITEMS are infinite. Since in any run, any enqueue or dequeue instance has at most three steps, the implementation is clearly 3-bounded wait-free. The main ideas for the proof of correctness are given in Sect. 4.

Access procedure E : Enqueue(x), for all $x \in V$:

```
1. (step 1) val ⟵ Swap(ITEMS[enq_row, tail], x)
2.          if val = $\top$
            then
3.              increment(enq_row)
4.              tail ⟵ 0
5. (step 2)     Swap(ITEMS[enq_row, tail], x)
6. (step 3)     Write(ROW, enq_row)
            end if
7.          increment(tail)
8.          return OK
```

Access procedure D_i : Dequeue, for all $1 \le i \le n$:

```
1. (step 1) deq_row ⟵ Read(ROW)
2. (step 2) head ⟵ Fetch&Increment(HEAD[deq_row])
3. (step 3) val ⟵ Swap(ITEMS[deq_row, head], $\top$)
4.          if val = $\perp$
            then
5.              return $\varepsilon$
            else
6.              return val
            end if
```

Fig. 1. Main Algorithm

Informally, the algorithm works as follows. The cells in the two dimensional array ITEMS are initialized to a default value, $\perp \notin V$. Whenever they are accessed during an Enqueue procedure, their value is updated to contain the element to be enqueued. Whenever they are accessed by a Dequeue procedure, their value is updated to contain $\top \notin V$. By design, each cell in the array ITEMS will be used at most once by an Enqueue operation, and at most once by a Dequeue operation.

In order to perform a Dequeue operation, process D_i reads from ROW the value of the active row in the two-dimensional array ITEMS. This is the row which was last used to enqueue a value by an Enqueue procedure which has already finished. Having obtained the value of this row in its local variable *deq_row*, process D_i selects the column *head* of a cell to query on this row using the Fetch&Increment object HEAD[*deq_row*]. It then proceeds to query the Swap object ITEMS[*deq_row*, *head*] and update its value to $\top$. If the value retrieved is not $\bot$, then some value to be enqueued was written in this location and the process dequeues that value. Otherwise, this location was never used by an Enqueue operation, and in this case the dequeuer process finds an empty queue.

The process E performing Enqueue operations has two local *persistent* variables, *enq_row* and *tail*. They are persistent in the sense that their values are not lost from one invocation of the enqueue procedure to the next. The value of the variable *enq_row* mirrors the value of the shared register ROW, while *tail* contains the smallest index of a Swap object not already used by an Enqueue procedure on row *enq_row* of ITEMS.

In order to perform an Enqueue operation, process E writes the value to be enqueued in the array location ITEMS[*enq_row*, *tail*] and retrieves the latter's value. If this value was $\top$, then some Dequeue operation has already accessed this cell before E had a chance to write to it. In this case, the Enqueue procedure will abandon the current row and start using the next row for storing the values in the Queue. Notice that no dequeuer could have used the new row that the enqueuer writes to in its second step (line 5), because the index of that row appears in ROW only after the third step by the enqueuer. Hence, the result obtained by the enqueuer to its second step is always $\bot$.

The access procedures above consist of local computation and accesses to shared objects, that is, steps. A complete execution of the Enqueue procedure can consist of at most three steps, in lines 1, 5 and 6. A complete execution of the Dequeue procedure always consists of three steps, in its first three lines.

For example, Fig. 2 presents a possible state of the shared variables in this implementation. Exactly two Enqueue procedures with arguments 1 and 2 were started, both were completed, and neither of them executed the body of the `if` statement in lines 3 through 6. Exactly four Dequeue procedures were started and executed at least their first two steps. All four of them obtained the result 0 in their first step, and they obtained the results $0, 1, 2, 3$ in their second steps, respectively. The Dequeue procedures with ($deq\_row = 0, head = 0$) and ($deq\_row = 0, head = 2$) were completed and output the values 1 and ε, respectively. The Dequeue procedures with ($deq\_row = 0, head = 1$) and ($deq\_row = 0, head = 3$) only executed their first two steps, and if either was allowed to take another step, they would output 2 and ε, respectively.

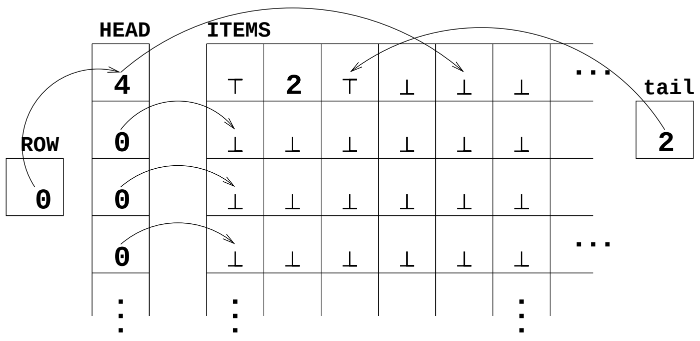

Fig. 2. A possible state of the shared variables in this implementation

In Fig. 3, a new Enqueue procedure with argument 3 is started and completed. This procedure applied a Swap operation with argument 3 to the cell ITEMS$[0, 2]$, obtained the result $\top$ for that step, and it then executed the body of the `if` statement. This is the situation in which a dequeuer accesses a cell of ITEMS before the enqueuer.

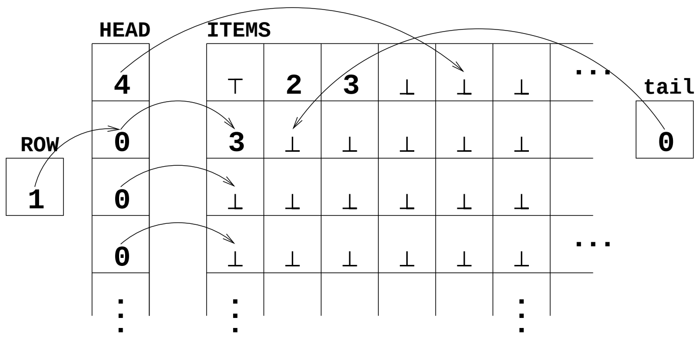

Fig. 3. The state after another Enqueue procedure is started and completed

The state in Fig. 4 is the result of a possible execution extending the one which led to the state in Fig. 3. Three more Enqueue procedures with arguments $4, 5, 6$ were started, and all of them were completed. None of these Enqueue procedures executed the body of the `if` statement. One more Dequeue procedure was started and executed its first two steps, obtaining $(\mathit{deq\_row} = 1, \mathit{head} = 0)$. This Dequeue procedure was completed, and it output 3. Furthermore, one of the two incomplete Dequeue procedures from the state in Fig. 2 was completed, the one with $(\mathit{deq\_row} = 0, \mathit{head} = 3)$, and it output ε.

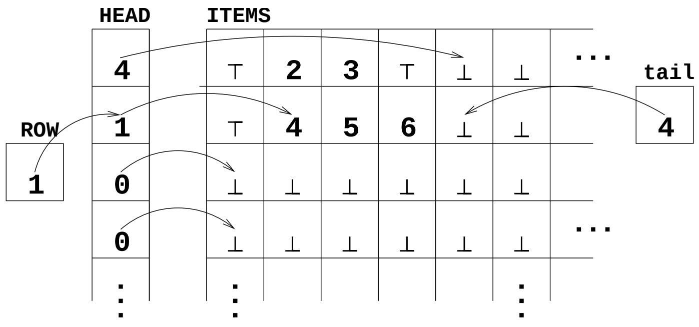

Fig. 4. Yet another possible state, extending the previous one

4 Proof of Linearizability

Due to space constraints, we only give the key ideas needed to prove that our algorithm is linearizable. More specifically, we explain how to assign linearization points for access procedures in an arbitrary run R of this implementation. A formal proof of linearizability is presented in [Dav04].

First, we introduce some notation. For π an enqueue instance in R, let enq_row_π and $tail_\pi$ denote the values of the local variables *enq_row* and *tail*, respectively, at the beginning of the execution of π. Let val_π denote the result of the first step of π (line 1). For ϕ a dequeue instance in R, let deq_row_ϕ denote the result of the first step of ϕ (line 1). If ϕ has at least two steps, let $head_\phi$ denote the result of its second step (line 2). If ϕ is complete, that is, if it contains three steps, let val_ϕ denote the result of its third step (line 3).

Enqueue Instances. We consider two kinds of enqueue instances. We say that an enqueue instance π is a *regular* enqueue instance if $val_\pi \neq \top$, so E does not execute the body of the `if` statement during π. A complete regular enqueue instance consists of only one step. We say that π is a *jump* enqueue instance if $val_\pi = \top$, referring to the fact that it "jumps" to the next row of the array ITEMS. A complete jump enqueue instance consists of three steps.

Since all enqueue instances in R are executed sequentially by the same process E, no two enqueue instances are concurrent. Furthermore, only the last enqueue instance in R can be incomplete, because in a run R, a process must finish the execution of an access procedure before starting the next one. If the value of ROW is r, then there was exactly one jump enqueue instance π with $enq\_row_\pi = i$, for every $i = 0, \ldots, r-1$.

An Association between Enqueue Instances and Dequeue Instances. We also need to classify dequeue instances. To do that, we need a method which associates a dequeue instance ϕ with the enqueue instance π which enqueued the value that ϕ is dequeuing.

For a dequeue instance ϕ with at least two steps, we say that ϕ *reserves* the cell at row deq_row_ϕ and column $head_\phi$ of ITEMS. This is the only cell of ITEMS that ϕ will access. Conversely, no dequeue instance other than ϕ will access that cell. We establish a relation between dequeue instances and enqueue instances as follows. Let ϕ be a dequeue instance with at least two steps. If there exists an enqueue instance π such that:

- π accesses the cell in ITEMS reserved by ϕ, and
- if ϕ has three steps, then π accesses that cell before ϕ (in its third step),

then we define $\rho(\phi) = \pi$. It can be shown that if π exists, then π is unique, so the definition is sound. If no such enqueue instance exists, we leave $\rho(\phi)$ undefined. The following correlation between a complete dequeue instance ϕ and $\rho(\phi)$ exists: if ϕ returns ε, then $\rho(\phi)$ is not defined; if ϕ returns a value other than ε, then that value was enqueued by $\rho(\phi)$.

Dequeue Instances. We consider three types of dequeue instances.

A dequeue instance ϕ consisting of at least two steps is a *type I* dequeue instance if $\rho(\phi) = \pi$ is defined and the step in which π accesses the cell reserved by ϕ occurs *after* step 2 of ϕ. By definition of ρ, the step in which π accesses that cell has to precede the third step of ϕ, should the latter exist in R. It can be shown that π is a regular enqueue instance. Informally, a complete type I dequeue instance ϕ will return a value other than ε, but when ϕ reserves a cell (in step 2), the value is not yet in the cell.

A dequeue instance ϕ consisting of at least two steps is a *type II* dequeue instance if there exists a complete jump enqueue instance π' such that $enq\_row_{\pi'} = deq\_row_\phi$ and the third step of π' precedes the second step of ϕ. It can be shown that π' is unique and that $\rho(\phi)$ is undefined, i.e. no enqueue instance is associated with ϕ. Hence, ϕ cannot also be a type I dequeue instance. Informally, between step 1 and step 2 of a type II dequeue instance, a jump enqueue instance has incremented ROW. If complete, ϕ will return ε.

A dequeue instance ϕ consisting of at least two steps is a *type III* dequeue instance if it is neither type I nor type II. A type III dequeue instance may or may not return ε.

Linearization Points. To show that our algorithm is linearizable, we assign linearization points for all complete enqueue instances and all dequeue instances which perform at least two steps in R. We argue that the linearization point of any procedure instance α occurs during the execution of α, i.e. at or after the first step of α and, if α is complete, at or before the last step of α.

- A complete regular enqueue instance is linearized at its first (and only) step, in line 1.
- A complete jump enqueue instance is linearized at its third (and last) step, in line 6.

– A type I dequeue instance ϕ is linearized at the first step (line 1) of $\pi = \rho(\phi)$, immediately *after* π. We know in this case that π is a regular enqueue instance, so by definition of a type I dequeue instance, the second step of ϕ precedes the first step of π. Furthermore, if ϕ is complete, then its third step occurs after the first step of π. Hence, the linearization point of ϕ occurs at some point during its execution.
– For a type II dequeue instance ϕ, let π' be the unique complete jump enqueue instance such that $\mathit{enq\_row}_{\pi'} = \mathit{deq\_row}_{\phi}$ and the third step of π' occurs before the second step of ϕ. We linearize ϕ at the third step (line 6) of π', immediately *before* π'. Clearly, the first step of ϕ precedes the third step of π', for otherwise $\mathit{deq\_row}_{\phi} \neq \mathit{enq\_row}_{\pi'}$.
It turns out that many type II dequeue instances may be linearized at the same third step of some jump enqueue instance π'. In this case, we order these dequeue instances arbitrarily. Informally, this does not cause any problem because the queue is empty at that point and they all output ε.
– A type III dequeue instance is linearized at its second step, in line 2.

Responses of Incomplete Dequeue Instances. The linearization points we have defined provide us with a total order on the sequence of operations that are performed on the Queue object during a run of our implementation. To prove linearizability, we have to define responses for the incomplete instances we have chosen to linearize. In our case, the only incomplete instances we linearize are dequeue instances which perform at least two steps. Let ϕ be such a dequeue instance. If $\rho(\phi) = \pi$ is defined, let the response of ϕ to be the value enqueued by π. If $\rho(\phi)$ is undefined, let the response of ϕ be ε.

Completing the Proof of Linearizability. By defining linearization points and responses of incomplete linearized dequeue instances in a run R, we have generated a sequence $\sigma(R)$ of operations and responses on the implemented Queue object. To complete the proof of linearizability, we have to show that there exists a sequence of states of the Queue object, that starts with the empty state, and is consistent with $\sigma(R)$. This is formally done in [Dav04] by defining a Queue state based on the states of the shared objects in the system, followed by a somewhat tedious case analysis of how various steps of enqueue and dequeue procedures modify the states of the shared objects and, thus, the state of the Queue object.

5 Conclusions

The results in this paper, together with the ones in [Li01], establish that there exist wait-free linearizable Queue implementations from Common2 objects when there is either only one enqueuer or at most two dequeuers. The question whether there exists a wait-free linearizable fully-accessible Queue implementation from Common2 objects for three processes (or more) remains open, as is Herlihy's question about whether Fetch&Add objects can be used to implement every consensus number 2 type in a system of three (or more) processes.

Our implementation uses a one dimensional array HEAD and a two dimensional array ITEMS. Both arrays are assumed to be infinite. However, the array HEAD and one dimension (the number of rows) of ITEMS can both be made finite, with $O(n)$ rows, where n is the number of dequeuers. The idea is to reuse rows of ITEMS when it is safe to do so. We cannot reuse a row until we are sure that no dequeuer has reserved a cell on that row but not yet accessed it. To this end, every dequeuer will start by reading ROW, announcing the value retrieved in a single-writer Register, and then reading ROW again. If ROW changes between the two reads, the dequeuer outputs ε. Otherwise, it continues as before, with its Fetch&Increment and Swap operations. This way, whenever the enqueuer has to jump to the next row, it can read what row each dequeuer is operating on, and then select an unused row. This takes $O(n)$ steps, plus the time to reinitialize the cells of ITEMS on the selected row to $\perp$. Since any number of cells may have been previously used on that row, going through them all would be wait-free but not bounded wait-free. We can avoid this and maintain bounded wait-freedom by having the enqueuer increment a sequence number and storing it with a row index in the shared variable ROW. Every cell of ITEMS would then have a time-stamp of its own, and every time a process retrieves a value with an old time-stamp, it should treat that cell as being fresh (that is, containing $\perp$). The implementation will then be $O(n)$-bounded wait-free. We have not incorporated this scheme into the algorithm because the emphasis in this paper is on the existence of an algorithm rather than its efficiency.

Another interesting improvement, from a practical point of view, is to limit the size of each row by, say, the maximum number of items present in the Queue at any one time (should such a maximum exist). Even though we have been unable to design such a scheme, perhaps one could do it by somehow having the enqueuer jump to a new row when the current one becomes full. However, in this situation, it is unclear how to dequeue elements from the old row.

Li obtains in [Li01] a two-dequeuer Queue implementation by modifying the single-enqueuer implementation of [HW90], using ideas from Herlihy's universal construction in [Her91]. Specifically, he develops a mechanism by which two dequeuer processes agree on a total order on the Dequeue operations to be performed, and subsequently perform those operations much like they would in the original single-enqueuer case. We attempted to apply a similar mechanism in order to obtain a two-enqueuer implementation from our single-enqueuer implementation, but without success. Informally, the problem appears to lie with the interaction between the enqueuer and dequeuer processes: in the single-dequeuer implementation considered in [Li01], the communication between enqueuer processes and dequeuer processes is achieved exclusively through Register objects; while in our single-enqueuer implementation, this communication is achieved through both the Register object ROW, and the Swap objects in the array ITEMS. Li obtains a two-dequeuer implementation by (i) having the two dequeuer processes agree on a total order for the Dequeue operations; (ii) having each dequeuer execute the steps of each of the Dequeue operations, including those initiated by the other dequeuer process; and (iii) having the two dequeuer processes agree on the result of each Dequeue operation. When trying to extend our single-enqueuer implementation to allow two enqueuers, part (iii) is irrelevant (since all enqueue operations produce the same result, OK) and part (i) can be achieved by having the two enqueuer processes agree on a sequence of Enqueue operations. The problem lies with part (ii), specifically with the fact that we were unable to find any way in which two enqueuers can work together while performing a single Enqueue operation. In Li's extended implementation, each of the Register objects used for communication between enqueuer and dequeuer processes only influences the steps taken by the two dequeuer processes. If we were to apply the same method to our implementation, accesses to the shared Swap objects in the array ITEMS would influence not only the steps of the two enqueuer processes, but also the steps of one dequeuer process. For example, consider the situation in which enqueue processes E_1 and E_2 are working together to perform some enqueue operation. Suppose E_1 first applies its Swap operation to the cell ITEMS$[r, c]$, and then E_2 applies its own Swap operation to the same cell. At that moment, E_2 cannot tell if some dequeue process accessed that cell before E_1, so E_2 cannot tell if E_1 has to jump to the next row of ITEMS or not. This is merely an informal argument of why Li's method cannot be straightforwardly applied. The existence of an implementation for two enqueuers and three or more dequeuers remains open.

6 Acknowledgments

This work was supported by the Natural Sciences and Engineering Research Council of Canada. I thank my supervisor, Faith Ellen Fich, for her helpful advice, her constructive criticism and her relentless proofreading of my work.

References

[AWW93] Yehuda Afek, Eytan Weisberger, and Hanan Weisman. A completeness theorem for a class of synchronization objects. In *Proceedings of the 12th ACM Symposium on Principles of Distributed Computing*, pages 159–170, 1993.

[Dav04] Matei David. Wait-free linearizable queue implementations. Master's thesis, Univ. of Toronto, 2004.

[Her91] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[HW90] Maurice Herlihy and Jeanette Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):495–504, January 1990.

[Jay93] Prasad Jayanti. On the robustness of Herlihy's hierarchy. In *Proceedings of the 12th ACM Symposium on Principles of Distributed Computing*, pages 145–158, 1993.

[Li01] Zongpeng Li. Non-blocking implementation of queues in asynchronous distributed shared-memory systems. Master's thesis, Univ. of Toronto, 2001.