Logarithmic-Time Single Deleter, Multiple Inserter Wait-Free Queues and Stacks

Prasad Jayanti and Srdjan Petrovic

Department of Computer Science, Dartmouth College,
Hanover, New Hampshire, USA
{prasad,spetrovic}@cs.dartmouth.edu

Abstract. Despite the ubiquitous need for shared FIFO queues in parallel applications and operating systems, there are no sublinear-time wait-free queue algorithms that can support more than a single enqueuer and a single dequeuer. Two independently designed algorithms—David's recent algorithm [1] and the algorithm in this paper—break this barrier. While David's algorithm is capable of supporting multiple dequeuers (but only one enqueuer), our algorithm can support multiple enqueuers (but only one dequeuer). David's algorithm achieves $O(1)$ time complexity for both enqueue and dequeue operations, but its space complexity is infinite because of the use of infinite sized arrays. The author states that he can bound the space requirement, but only at the cost of increasing the time complexity to $O(n)$, where n is the number of dequeuers. A significant feature of our algorithm is that its time and space complexities are both bounded and small: enqueue and dequeue operations run in $O(\lg n)$ time, and the space complexity is $O(n + m)$, where n is the number of enqueuers and m is the actual number of items currently present in the queue. David's algorithm uses *fetch&increment* and *swap* instructions, which are both at level 2 of Herlihy's Consensus hierarchy, along with *queue*. Our algorithm uses the LL/SC instructions, which are universal. However, since these instructions have constant time wait-free implementation from CAS and restricted LL/SC that are widely supported on modern architectures, our algorithms can run efficiently on current machines. Thus, in applications where there are multiple producers and a single consumer (e.g., certain server queues and resource queues), our algorithm provides the best known solution to implementing a wait-free queue. Using similar ideas, we can also efficiently implement a stack that supports multiple pushers and a single popper.

1 Introduction

In parallel systems, *shared data objects* provide the means for processes to communicate and cooperate with each other. Atomicity of these shared data objects has traditionally been ensured through the use of locks. Locks, however, limit parallelism and cause processes to wait on each other, with several consequent drawbacks, including deadlocks, convoying, priority inversion, and lack of fault-tolerance to process crashes. This sparked off extensive research on the design

R. Ramanujam and S. Sen (Eds.): FSTTCS 2005, LNCS 3821, pp. 408–419, 2005.

of *wait-free* data objects, which ensure that every process completes its operation on the data object in a bounded number of its steps, regardless of whether other processes are slow, fast or have crashed [2]. We refer to this bound (on the number of steps that a process executes to complete an operation on the data object) as the *time complexity* (of that operation).

Early research sought to demonstrate the feasibility of implementing wait-free data objects, culminating in Herlihy's universal construction [2,3], which transforms *any* sequential implementation $\mathcal{A}$ of a data object into a wait-free shared implementation $\mathcal{B}$ of that data object. However, the worst-case time complexity of performing an operation on the data object $\mathcal{B}$ is $\Omega(n)$, where n is the number of processes sharing $\mathcal{B}$. In fact, it has been proved later that this linear dependence of time complexity on n is unavoidable with any universal construction [4]. Thus, if *sublinear time* wait-free data objects are our goal, it is imperative that algorithms exploit the semantics of the specific object (e.g., counter, queue, stack) being implemented. In recent years, this approach has indeed led to sublinear time wait-free algorithms for implementing a variety of shared data objects, e.g., the class of closed objects (which include counters and swap objects) [5], f-arrays [6], and LL/SC objects [7,8,9].

A *shared FIFO queue* is one of the most commonly used data objects in parallel applications and operating systems. Not surprisingly, a number of algorithms have been proposed for implementing queues. However, most of these algorithms are only nonblocking[1] and not wait-free. With the exception of a recent algorithm by David [1], which we will discuss shortly, wait-free queue algorithms either had excessive time complexity of $\Omega(n)$ [3,7,10,11], or implemented a queue in the restricted case when there is only one enqueuer and one dequeuer [12,13]. Specifically, until recently, no sublinear time wait-free queue algorithm was discovered, despite the ubiquitous need for queues. When we considered why there has been so little success in designing efficient wait-free queue algorithms, we found a plausible explanation, which we now describe. Normally, we represent a queue as a (linear) linked list of elements with variables *front* and *rear* holding the pointers to the first and the last elements of the list. To enqueue an element e, a process p must perform a sequence of steps: first it must read *rear* to locate the currently last element, then adjust that element to point to the new element e and, finally, adjust *rear* to point to e. If p stops after performing only a few of these steps, other processes (that need to enqueue elements into the queue) have no option but to help p (otherwise the implementation won't be wait-free). Thus, if k processes are concurrently performing enqueue operations, the last process to complete the enqueue may have to help all other processes, resulting in $\Omega(k)$ time complexity for its enqueue operation. In the worst case, k may be as high as n, the maximum number of processes sharing the queue. Hence, the worst-case time complexity of an enqueue operation is linear in n.

[1] A nonblocking queue is strictly weaker than a wait-free queue: when multiple processes attempt to execute operations on a nonblocking queue, all but one process may starve.

The above informal reasoning suggests that any wait-free queue algorithm, if it aspires to break the linear time barrier, must necessarily be based on a more creative data structure than a linear linked list. This paper proposes one such novel data structure where a queue is represented as a binary tree whose leaves are linear linked lists. Based on this data structure, we design an efficient wait-free queue algorithm, but it has one limitation: the algorithm does not allow concurrent dequeue operations. In other words, our algorithm requires that the dequeue operations be executed one after the other, but allows the dequeue to be concurrent with any number of enqueue operations. The significant features of our algorithm are:

1. Sublinear Time Complexity: The worst-case time complexity of an *enqueue* or a *dequeue* operation is $O(\lg n)$, where n is the maximum number of processes for which the queue is implemented.
2. Space Efficiency: At any time t, the algorithm uses $O(m + n)$ space, where m is the actual number of items present in the queue at time t.

Our algorithm uses LL/SC instructions which act like read and conditional-write, respectively. More specifically, the LL(X) instruction by process p returns the value of the location X, while the SC(X, v) instruction by p checks whether some process updated the location X since p's latest LL, and if that isn't the case it writes v into X and returns *true*; otherwise, it returns *false* and leaves X unchanged. Although these instructions are not directly supported in hardware, they have constant time and space wait-free implementation from CAS and restricted LL/SC, which are widely supported on modern architectures [7,8,9]. Consequently, our algorithms can run efficiently on current machines. In applications where there are multiple producers and a single consumer (e.g., certain server queues and resource queues), our algorithm provides the best known solution to implementing a wait-free queue.

Concurrently with our research and independently, David designed a sublinear-time wait-free queue algorithm [1] that imposes a different limitation than ours: his algorithm does not allow concurrent enqueue operations (but allows an enqueue operation to run concurrently with any number of dequeue operations). In contrast, our algorithm does not allow concurrent dequeue operations, but allows a dequeue to run concurrently with any number of enqueue operations. In the following we describe David's result and contrast it with ours.

David's algorithm implements a wait-free *single enqueuer, multiple dequeuer* queue while our algorithm implements a *multiple enqueuer, single dequeuer* queue. His algorithm uses *fetch&increment* and *swap* instructions while ours uses LL/SC instructions. The enqueue and dequeue operations run in $O(1)$ time in his algorithm while they run in $O(\lg n)$ time in our algorithm. David's result is interesting because (1) it shows for the first time that it is possible to implement a sublinear-time, non-trivial, wait-free queue (i.e., a queue that supports more concurrent operations than just a single enqueue and a single dequeue) from

objects of the same power as the queue itself,[2] and (2) it achieves the best possible running time of $O(1)$ for enqueue and dequeue operations. His algorithm, however, is not practical: it uses arrays of infinite length and therefore has an infinite space complexity. The author states that it is possible to bound the space complexity of his algorithm, but at the cost of increasing the time complexity of the algorithm to $O(n)$. In contrast, our algorithm, besides achieving sublinear time complexity, is also space efficient: its space complexity is $O(n + m)$, where n is the total number of processes sharing the queue and m is the actual number of items currently present in the queue.

The ideas introduced in this paper have a more general applicability than for just implementing queues: they have proved useful in designing a wait-free *multiple pusher, single popper* stack with the same time and space complexities as for queue, and seem to have potential for efficiently implementing priority queues (which we are currently exploring).

1.1 Algorithmic Ideas in a Nutshell

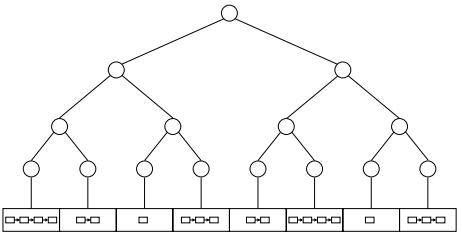

Fig. 1. Main data structure for the multiple enqueuer, single dequeuer queue.

The key idea behind our algorithm is to distribute the "global" queue $\mathcal{Q}$ (that we wish to implement) over n "local" queues, where n is the maximum number of processes sharing $\mathcal{Q}$. More specifically, the algorithm keeps one local queue at each process. To enqueue an element e, a process p first obtains a time stamp t and then inserts the pair (e, t) into its local queue. (Processes that concurrently attempt to get time stamps are allowed to get the same time stamp, which makes it possible to obtain a time stamp in just two machine instructions, as we will see later.) Thus, the local queue at p contains only the elements inserted by p, in the order of their insertion. The front element of the *global* queue is the earliest of the front elements of the local queues (*i.e.*, an element with the smallest time stamp over all local queues). The naive strategy to locate this earliest element would be to examine the front elements of all of the local queues, which leads to $O(n)$ running time for the dequeue operation. We use a cleverer strategy that views the front elements of the local queues as the leaves of a binary tree (see Figure 1) and, using ideas first proposed by Afek, Dauber and Touitou [14], propagate the minimum of these elements to the root of the tree. Significantly, even though many processes might be acting concurrently, this propagation works correctly and takes only $O(\lg n)$ time. The dequeuer reads the root to determine the local queue that has the earliest element, removes that element from the local queue, and finally propagates the new front element of that local queue towards the root (to ensure that the root reflects the new minimum element).

[2] By the power of an object we mean its level in the Herlihy's Consensus Hierarchy [2]. All of *fetch&increment*, *swap*, and *queue* are at level 2 of the consensus hierarchy, and are thus equally powerful by this measure.

In the above strategy, the local queue at process p is accessed (possibly concurrently) by at most two processes—the enqueuer (namely, process p) and the lone dequeuer. The enqueuer executes either an *enqueue* operation or a *read-front* operation, which returns the front element without affecting the state of the queue. The dequeuer executes either a *dequeue* operation or a *read-front* operation. The *read-front* operation is needed because, in order to propagate the front element of the local queue towards the root of the binary tree, the enqueuer and the dequeuer need the means to determine what the front element of the local queue is. In Section 2 we describe how to implement a local queue supporting these operations using only *read* and *write* operations on shared variables. Then, in Section 3, we describe how to implement a global queue from the local queues, using the strategy described in the previous paragraph.

2 Single Enqueuer, Single Dequeuer Queue

In this section we present an important building block for our main queue implementation, namely, a *single enqueuer, single dequeuer queue object* that supports three operations—*enqueue(v)*, *dequeue()*, and *readFront*(). The *enqueue(v)* operation inserts an element v into the queue. The *dequeue()* operation removes and returns the front element of the queue. The *readFront*() operation reads the value of the front element without removing that element from the queue. If the queue is empty, both *dequeue()* and *readFront*() return $\perp$.

A single enqueuer, single dequeuer queue can be accessed by two processes: the *enqueueing process*, which can invoke an *enqueue()* or a *readFront*() operation, and a *dequeuing process*, which can invoke a *dequeue()* or a *readFront*() operation. To distinguish between the *readFront*() operations performed by an enqueuer and a dequeuer, we refer to the two operations as $readFront_e$ and $readFront_d$, respectively.

Figure 2 presents an implementation of a single enqueuer, single dequeuer queue. This algorithm is similar to the two-lock queue algorithm by Michael and Scott [15], except that it additionally supports the *readFront()* operation. (We need this operation for our main queue algorithm, presented in the next section.) As we will shortly see, the presence of the *readFront()* operation significantly complicates the algorithm design.

The queue is represented as a singly linked list terminated by a dummy node. Variables `First` and `Last` point, respectively, to the first and the last node in the list. Enqueueing and dequeueing elements from the queue consists of inserting and removing nodes from the linked list in a way similar to a sequential linked-list implementation of a queue. In particular, to enqueue the value v into the queue, process p performs the following steps. First, p creates a new node (Line 1). Next, p locates the last node in the list, i.e., the dummy node (Line 2) and writes the value v into that node (Line 3). Finally, p completes its operation by inserting a new (dummy) node to the end of the list (Lines 4 and 5). To dequeue an item from the queue, process p first checks whether the list contains only a single node (Lines 6 and 7). If it does, then that node is a dummy node and

```
Types
    valuetype = Any type
    nodetype = record val: valuetype; next: pointer to nodetype end
Shared variables
    First, Last, Announce, FreeLater: pointer to nodetype
    Help: valuetype
Initialization
    First = Last = new Node()
    FreeLater = new Node()

    procedure enqueue(v)
1:  newNode = new Node()
2:  tmp = Last
3:  tmp.val = v
4:  tmp.next = newNode
5:  Last = newNode

    procedure readFronte()
                                        returns valuetype
17: tmp = First
18: if (tmp == Last) return ⊥
19: Announce = tmp
20: if (tmp ≠ First)
21:     retval = Help
22: else retval = tmp.val
23: return retval

    procedure dequeue()
                                        returns valuetype
6:  tmp = First
7:  if (tmp == Last) return ⊥
8:  retval = tmp.val
9:  Help = retval
10: First = tmp.next
11: if (tmp == Announce)
12:     tmp' = FreeLater
13:     FreeLater = tmp
14:     free(tmp')
15: else free(tmp)
16: return retval

    procedure readFrontd()
                                        returns valuetype
24: tmp = First
25: if (tmp == Last) return ⊥
26: return tmp.val
```

Fig. 2. Implementation of the single enqueuer, single dequeuer queue object from *read/write* registers.

so p returns $\perp$ (Line 7). Otherwise, p reads the value stored at the front node (Line 8), removes that node from the list (Line 10), and frees up memory used by that node (Lines 14 or 15). Finally, p completes its operation by returning the value it read from the front node (Line 16).

Observe that in the above algorithm, the enqueue and dequeue operations work on separate parts of the list, namely, its front and back. Hence, there is no contention between the two operations, which is why the above algorithm is so simple. However, if we add a *readFront()* operation to our set of operations, we have a problem: both $readFront_e()$ and *dequeue()* now operate on the same node of the list, namely, its front node. This contention on the first element of the list complicates the design of the two operations, as we describe below.

To perform a $readFront_e$ operation, process p first reads the pointer to the front node in the list (Line 17) and then checks whether the list contains only a single node (Line 18). If it does, then that node is a dummy node and so p

returns $\perp$ (Line 18). Otherwise, at the time when p read the pointer to the front node, the front node indeed contained a valid value for p's operation to return. However, it is quite possible that after p's reading of the pointer a dequeue operation deleted the front node and freed up the memory used by it. Thus, p cannot even attempt to read the value at the front node since it may result in a memory access to a memory that has already been freed.

So, instead of reading the value at the front node, p first writes the pointer to the front node into the variable `Announce` (Line 19), notifying a dequeuing operation that it is interested in the front node and asking it not to free that node from the memory. Next, p reads the pointer to the front node again (Line 20). If the pointer is the same as before, then p knows that a dequeuer has not yet removed the front node from the queue. (Notice that the front node could not have been removed and re-inserted into the queue because $readFront_e$ is executed by the enqueuer and hence no insertions take place during that operation.) More importantly, p can be certain that a dequeuer will notice p's announcement (at Line 11) and as a result will not free up that node from memory. (Instead, a dequeuer will store the pointer to the front node into the `FreeLater` variable, and will free it up only after p changes its announcement—Lines 12–14.) Hence, p simply reads and returns the value stored at the front node (Lines 22 and 23). If, on the other hand, the pointer to the front node has changed, then a dequeuer has already removed the front node from the queue and potentially freed up that node from memory. So, p must obtain a valid value for its operation to return by other means, and it does so with help from a dequeuing operation, which, prior to removing the front node from the list, first writes the value stored at the front node into the variable `Help` (Line 9). When p notices that the pointer to the front node has changed, it knows that some dequeuing operation removed the front node from the list and has therefore written into `Help` the value at that node. Hence, by simply reading the variable `Help`, p can obtain a valid value for its operation to return. In particular, p will obtain either the value stored at the original front node, or, if multiple nodes have been removed, it will obtain a value stored at a node that was at the front of the list at some point during p's operation. Hence, p reads and returns the value stored in `Help` (Lines 21 and 23) and terminates its operation.

The algorithm for $readFront_d$ is very simple: process p first checks whether the list contains only a single node (Lines 24 and 25). If it does, then that node is a dummy node, and so p returns $\perp$ (Line 25). Otherwise, p returns the value stored at the front node of the list (Line 26). The reason why $readFront_d$ is much simpler than $readFront_e$ is that the former operation never overlaps with a *dequeue()* operation (since they are both executed by the same process). Hence, when p reads the pointer to the front node, p knows that the front node will not be removed or freed before it reads the value stored at that node. Therefore, p is guaranteed to return the valid value.

Based on the above, we have the following theorem.

Theorem 1. *The algorithm in Figure 2 is a linearizable [16] wait-free implementation of the single enqueuer, single dequeuer queue from read/write regis-*

ters. The time complexities of `enqueue`*,* `dequeue`*,* $\texttt{readFront}_\texttt{e}$*, and* $\texttt{readFront}_\texttt{d}$ *operations are* 5*,* 10*,* 6*, and* 3*, respectively. The space consumption of the algorithm at any time t is $O(m)$, where m is the number of elements in the queue at time t.*

3 Multiple Enqueuer, Single Dequeuer Queue

In this section we present our main result, namely, the wait-free implementation of a *multiple enqueuer, single dequeuer queue object* with time complexity of $O(\lg n)$. The high-level intuition for this algorithm was presented in Section 1.1; the reader should consult that section before proceeding further.

We begin by a formal definition of this object. An n-process multiple enqueuer, single dequeuer queue object supports two operations—*enqueue(p,v)* and *dequeue(p)*. The *enqueue(p,v)* operation enables process p to insert an element v into the queue. The *dequeue(p)* operation enables process p to remove the front element from the queue. Multiple processes are allowed to execute the *enqueue()* operation concurrently. However, at any point in time, at most one process can be executing the *dequeue()* operation. The *dequeue()* operation can overlap with any number of *enqueue()* operations.

The algorithm is presented in Figure 3. It maintains two data structures: an array `Q` of size n, holding one single enqueuer, single dequeuer queue for each process, and a complete binary tree `T` of size n built on top of that array. To remove any ambiguity, the algorithm refers to the enqueue and dequeue operations on the single enqueuer, single dequeuer queues by the names *enqueue2* and *dequeue2*, respectively.

To enqueue an element v into the queue, process p first obtains a new time stamp by reading the variable `counter` (Line 1). Next, p increments `counter` by adding 1 to the time stamp it has obtained and SC-ing the result back into `counter` (Line 2). Notice that by the end of this step, the value in `counter` is strictly greater than p's time stamp, regardless of whether p's SC on `counter` succeeds or not. (This is because even if p's SC fails, some other process must have performed a successful SC and has therefore incremented the `counter`.) Notice also that it is quite possible for two different processes to obtain the same time stamp. To ensure that all time stamps are unique, a process adds its process id to the back of its time stamp.

After constructing the time stamp, process p inserts the element v along with the time stamp into its local queue (Line 3). Notice that, since p increments the variable `counter` during each enqueue operation, successive enqueue operations by p obtain time stamps in a strictly increasing order. Consequently, the time stamps of the elements in p's local queue are ordered as follows: the front element has the smallest time stamp, the next element has a higher time stamp, and so on. The front element of the *global* queue is the earliest of the front elements of all the local queues, i.e., an element with the smallest time stamp over all local queues. To help a dequeuer locate this earliest element quickly, process p propagates the time stamp at the front element of its local queue toward the

Types
valuetype = Any type
queue2type = Single enqueuer, single dequeuer queue
treetype = Complete binary tree with n leaves
Shared variables
`counter`: integer (`counter` supports LL and SC operations)
`Q`: **array** $[1\,..\,n]$ **of** queue2type
`T`: treetype
Initialization
`counter` $= 0$

procedure `enqueue`(p, v)
1: $tok =$ `LL(counter)`
2: `SC(counter,` $tok + 1$`)`
3: `enqueue2(Q[`p`], (`v`, (`tok, p`)))`
4: `propagate(Q[`p`])`

procedure `propagate`(Q)
5: $currentNode = Q$
6: **repeat**
7: $currentNode =$ `parent(`$currentNode$`)`
8: **if** $\neg$`refresh()`
9: `refresh()`
10: **until**$(currentNode ==$ `root(T))`

procedure `dequeue`(p) **returns** valuetype
11: $[t, q] = read($`root(T)`$)$
12: **if** $(q == \perp)$ **return** $\perp$
13: $ret =$ `dequeue2(Q[`q`])`
14: `propagate(Q[`q`])`
15: **return** $ret.val$

procedure `refresh`() **returns** boolean
16: `LL(`$currentNode$`)`
17: read time stamps in $currentNode$'s children
Let $minT$ be the smallest time stamp read
18: **return** `SC(`$currentNode, minT$`)`

Fig. 3. Implementation of the n-process multiple enqueuer, single dequeuer queue object from LL/SC variables and *read/write* registers.

root of the binary tree (Line 4). At each internal tree node s that p visits, it makes sure that the following invariant holds:

Invariant: Let s be any node in the binary tree. Let $E(s)$ be the set of all elements in the local queues located at the leaves of the subtree rooted at s whose time stamps have been propagated up to s. Then, at any point in time, s contains the smallest time stamp among all elements in $E(s)$.

We will describe shortly how the above Invariant is maintained. For now, we assume that the Invariant holds and look at how it impacts the implementation of the dequeue operation.

To dequeue an element from the queue, process p first reads the time stamp stored at the root node (Line 11). If the time stamp is $\perp$, it means that the queue is empty and so p returns $\perp$ (Line 12). Otherwise, by the Invariant, the time stamp $[t, q]$ that p reads at the root is the smallest time stamp among all elements in the local queues that have been propagated to the root. Furthermore, the element with that time stamp must be located at the front of q's local queue.

Therefore, to obtain the front element a of the global queue, p simply visits q's local queue and removes the front element from it (Line 13). Since the smallest time-stamped element has been removed from q's local queue, some of the nodes in the binary tree need to be updated in order to reflect this change (and maintain the Invariant). Hence, p propagates the new front element to the root, updating each node along the way (Line 14). At the end, p returns the element a that it dequeued from q's local queue (Line 15).

Notice that, at the time when p reads the root, it is possible that there exists another element b with a smaller time stamp than a that hasn't yet been propagated to the root. In this case, p is in fact not returning the element with the smallest time stamp among all local queues. However, the fact that (1) b has a smaller time stamp than a, and (2) b hasn't been propagated to the root, implies that the enqueue operations that inserted elements a and b overlap. Thus, we can assume that the operation that inserted a has taken effect (i.e., was linearized) before the operation that inserted b, and hence p was correct in returning a.

We now describe the mechanism by which nodes in the tree are updated to maintain the Invariant. More specifically, we describe the manner in which a process p propagates the change at the front of some local queue $\mathtt{Q}[q]$ to all the nodes on the path from $\mathtt{Q}[q]$ to the root, thereby ensuring that the Invariant holds at each node on that path. (Notice that the other tree nodes are unaffected by the change and for that reason do not need to be updated.) This propagation step is similar to the way propagation is handled in Afek, Dauber, and Touitou's universal construction [14]. It is captured by a procedure `propagate`(), described below.

Process p starts at the local queue $\mathtt{Q}[q]$ (Line 5). During the next $\lg n$ iterations, p repeatedly performs the following steps. First, p sets its current node to be the parent of the old current node (Line 7). Next, p attempts to update the minimum time stamp at the current node by a call to `refresh`() (Line 8). If `refresh`() returns *true*, it means that p's attempt to update the current node has succeeded. If p's attempt fails, p simply calls `refresh`() again (Line 9). The interesting feature of the algorithm is that, by the end of this call, the current node is sure to have an updated minimum time stamp, regardless of whether the second `refresh`() fails or not. To see why that is so, we must look at how the `refresh`() procedure is implemented.

During each `refresh`() call, p performs the following steps. First, p LL's the current node s (Line 16). Next, p reads the time stamps from all the children of s (Line 17). (If the current node is a leaf, then reading of the children consists of reading the time stamp of the first element of the local queue.) Then, p finds the smallest time stamp t among all the time stamps it read. Observe that, since the time stamps at the children (by the Invariant) contain the smallest time stamps among all elements in their respective subtrees that have been propagated to those nodes, t is guaranteed to be the smallest time stamp among all elements in the subtree rooted at s that have been propagated to s. So, p installs the value t into s by performing an SC on s (Line 18). If p's SC succeeds, time stamp t

is installed into s (thus ensuring that the Invariant holds) and p returns *true*. Otherwise, p's attempt to install t into the current node has failed (due to some other process performing a successful SC between p's LL at Line 16 and its SC at Line 18) and p returns *false* (Line 18).

We now justify an earlier claim that, after p calls `refresh`() twice, the current node is sure to have an updated minimum time stamp, even if both `refresh`() calls fail. Observe that if the first `refresh`() call fails, then some process q had performed a successful SC operation on the current node during that call. Similarly, if the second `refresh`() call fails, then some process r had performed a successful SC operation on the current node during that call. Since r's SC succeeded, r must have LL-ed the current node after q performed its successful SC. Consequently, r must have also read the time stamps at current node's children after q performed its successful SC and before r performed its successful SC. Hence, r reads the time stamps during the time interval of p's two calls to `refresh`(). So, the time stamps that r reads from the current node's children are the time stamps that p would have also read from current node's children. Therefore, r's successful SC installs the correct minimum time stamp into the current node. As a result, by the end of p's second `refresh`() call, the current node is sure to have an updated minimum time stamp, thus ensuring that the Invariant holds.

We now analyze the time and space complexities for the above algorithm. First, observe that the tree updating process takes $O(\lg n)$ time. Hence, the time complexity for both dequeue and enqueue operations is $O(\lg n)$. Second, by Theorem 1, the space complexity of a local queue is proportional to the actual number of elements in the queue. Therefore, if m is the total number of items in the global queue at time t, then the space used by all local queues at time t is $O(m)$. Since the tree uses $O(n)$ space at any time, the space consumption for the queue algorithm at time t is $O(m + n)$.

Notice that the algorithm stores at each node in the tree a time stamp consisting of an unbounded integer and a process id. Since these two values are stored together in the same machine word, it means that if the length of the machine word is b, $b - \lg n$ bits are left for the unbounded integer. Since the value of the unbounded integer is proportional to the number of enqueue operations invoked on the queue, it follows that the algorithm allows at most $2^b/n$ enqueue operations to be performed. This restriction is not a limitation in practice. For instance, on most modern architectures, we have $b = 64$. Hence, even if the number of processes accessing the algorithm is as large as 16,000, the algorithm allows for at least 2^{50} enqueue operations to be performed. If there are a million enqueue operations performed each second, the algorithm can thus safely run for more than 35 years!

The following theorem summarizes the above discussion.

Theorem 2. *Let b be the length of the machine word, and n be the number of processes. Under the assumption that at most $2^b/n$ enqueue operations are invoked, the algorithm in Figure 3 is a linearizable wait-free implementation of the multiple enqueuer, single dequeuer queue from LL/SC words and registers.*

The time complexity for both the dequeue and enqueue operations is $O(\lg n)$. The space consumption of the algorithm at any time t is $O(m + n)$ machine words, where m is the number of elements in the queue at time t.

Acknowledgments

We thank the anonymous referees for their valuable comments on an earlier version of this paper.

References

1. David, M.: A single-enqueuer wait-free queue implementation. In: Proceedings of the 18th International Conference on Distributed Computing. (2004) 132–143
2. Herlihy, M.: Wait-free synchronization. ACM TOPLAS **13** (1991) 124–149
3. Herlihy, M.: A methodology for implementing highly concurrent data structures. ACM Transactions on Programming Languages and Systems **15** (1993) 745–770
4. Jayanti, P.: A lower bound on the local time complexity of universal constructions. In: Proceedings of the 17th Annual Symposium on Principles of Distributed Computing. (1998)
5. Chandra, T., Jayanti, P., Tan, K.Y.: A polylog time wait-free construction for closed objects. In: Proceedings of the 17th Annual Symposium on Principles of Distributed Computing. (1998)
6. Jayanti, P.: f-arrays: implementation and applications. In: Proceedings of the 21st Annual Symposium on Principles of Distributed Computing. (2002) 270 – 279
7. Anderson, J., Moir, M.: Universal constructions for large objects. In: Proceedings of the 9th International Workshop on Distributed Algorithms. (1995) 168–182
8. Jayanti, P., Petrovic, S.: Efficient and practical constructions of LL/SC variables. In: Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing. (2003)
9. Moir, M.: Practical implementations of non-blocking synchronization primitives. In: Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing. (1997) 219–228
10. Afek, Y., Weisberger, E., Weisman, H.: A completeness theorem for a class of synchronization objects. In: Proceedings of the 12th Annual Symposium on Principles of Distributed Computing. (1993) 159–170
11. Li, Z.: Non-blocking implementation of queues in asynchronous distributed shared-memory systems. Master's thesis, University of Toronto (2001)
12. Lamport, L.: Specifying concurrent program modules. ACM Transactions on Programming Languages and Systems **5** (1983) 190–222
13. Michael, M., Scott, M.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing. (1996) 267–276
14. Afek, Y., Dauber, D., Touitou, D.: Wait-free made fast. In: Proceedings of the 27th Annual ACM Symposium on Theory of Computing. (1995) 538–547
15. Michael, M., Scott, M.: Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. Journal of Parallel and Distributed Computing (1998) 1–26
16. Herlihy, M., Wing, J.: Linearizability: A correctness condition for concurrent objects. ACM TOPLAS **12** (1990) 463–492