Wait-Free Queues With Multiple Enqueuers and Dequeuers *

Alex Kogan
Department of Computer Science
Technion, Israel
sakogan@cs.technion.ac.il

Erez Petrank
Department of Computer Science
Technion, Israel
erez@cs.technion.ac.il

Abstract

The queue data structure is fundamental and ubiquitous. Lock-free versions of the queue are well known. However, an important open question is whether practical wait-free queues exist. Until now, only versions with limited concurrency were proposed. In this paper we provide a design for a practical wait-free queue. Our construction is based on the highly efficient lock-free queue of Michael and Scott. To achieve wait-freedom, we employ a priority-based helping scheme in which faster threads help the slower peers to complete their pending operations. We have implemented our scheme on multicore machines and present performance measurements comparing our implementation with that of Michael and Scott in several system configurations.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features – Concurrent programming structures; E.1 [*Data structures*]: Lists, stacks, and queues

General Terms Algorithms, Performance

Keywords concurrent queues, wait-free algorithms

1. Introduction

The proliferation of multicore systems motivates the research for efficient concurrent data structures. Being a fundamental and commonly used structure, first-in first-out (FIFO) queues[1] have been studied extensively, resulting in many highly concurrent algorithms (e.g., [14, 19, 21]). A concurrent queue algorithm supports linearizable `enqueue` and `dequeue` operations, which add and remove elements from the queue while observing the FIFO semantics.

A highly desired property of any concurrent data structure implementation, and queues in particular, is to ensure that a process (or a thread) completes its operations in a bounded number of steps, regardless of what other processes (or threads) are doing. This property is known in the literature as (bounded) *wait-freedom* [9, 11, 15]. It is particularly important in systems where strict deadlines for operation completion exist, e.g., in real-time applications or when operating under a service level agreement (SLA), or in heterogenous execution environments where some of the threads may perform much faster or slower than others. Yet, most previous queue implementations (e.g., [14, 17, 19, 21, 24, 25]) provide the weaker *lock-free* property; lock-freedom ensures that among all processes accessing a queue, at least one will succeed to finish its operation. Although such non-blocking implementations guarantee global progress, they allow scenarios in which all but one thread starve while trying to execute an operation on the queue. The few wait-free queue constructions that exist, as discussed later, are either stem from general transformations on sequential objects and are impractical due to significant performance drawbacks, or severely limit the number of threads that may perform one or both of the queue operations concurrently.

In fact, when considering concurrent data structures in general, one realizes that with only a few exceptions (e.g., [5, 22]), wait-free constructions are very rare in practice. A possible reason for this situation is that such constructions are hard to design in an efficient and practical way. This paper presents the first practical design of wait-free queues, which supports multiple concurrent dequeuers and enqueuers. Our idea is based on the lock-free queue implementation by Michael and Scott [19], considered to be one of the most efficient and scalable non-blocking algorithms in the literature [11, 14, 24]. Our design employs an efficient helping mechanism, which ensures that each operation is applied exactly once and in a bounded time. We achieve wait-freedom by assigning each operation a dynamic age-based priority and making threads with younger operations help older operations to complete. Moreover, we believe the scheme used to design the operations of our queue can be useful for other data structures as well. In addition to the base version of our wait-free algorithm, we propose several optimizations to boost the performance when the contention is low and/or the total number of threads in the system is high.

We have implemented our design in Java and compared it with an implementation of Michael and Scott's lock-free queue [19] using several benchmarks and various system configurations. Our performance evaluation shows that the wait-free algorithm is typically slower than the lock-free implementation, but the slowdown depends on the operating system configuration. Issues such as scheduling policy, memory management, policy of allocating threads to computing cores might have a crucial impact on the produced interleavings in thread executions. Interestingly, the wait-free queue is even faster than the lock-free one, in some realistic cases, in spite of its much stricter progress guarantee.

2. Related Work

Lock-free queue implementations are known for more than two decades, starting from works by Treiber [23], Massalin and Pu [17] and Valois [25]. The algorithm presented by Michael and Scott [19]

* This work was supported by the Israeli Science Foundation grant No. 283/10. The work of A. Kogan was also supported by the Technion Hasso Plattner Center.

[1] This paper deals with FIFO queues only, in the following referred simply as *queues*, for brevity.

PPoPP'11, February 12–16, 2011, San Antonio, Texas, USA.

almost 15 years ago is often considered the most scalable lock-free queue implementation to date [11, 14, 24]. Several recent works propose various optimizations over this implementation [14, 21].

Yet, until now, a practical wait-free implementation of the queue data structure remained an important open question. All previous proposals limited concurrency in queue operations. The first such implementation was introduced by Lamport [16] (also in [11]); it allows only one concurrent enqueuer and dequeuer. Also, the queue in [16] is based on a statically allocated array, which essentially bounds the number of elements that the queue may contain. More recently, David [8] proposed a wait-free queue that supports multiple dequeuers, but only one concurrent enqueuer. His queue is based on infinitely large arrays. The author states that he may get a bounded-array based implementation at the price of increased time complexity. Following David's work, Jayanti and Petrovic [13] proposed a wait-free queue implementation supporting multiple enqueuers, but only one concurrent dequeuer. The elements of their queue are stored in dynamically allocated nodes.

Thus, we are not aware of any previous implementation of a wait-free queue, which supports multiple concurrent enqueuers and multiple concurrent dequeuers. In particular, we are not familiar with an implementation that does not require such a queue to be statically allocated and does not use primitives stronger than compare-and-swap (CAS), widely available on modern processors.

In the context of non-blocking concurrent data structures, we should also mention *universal constructions* [10, 11]. These constructions are generic methods to transform any sequential object into lock-free (or wait-free) linearizable concurrent object. The first such construction was proposed by Herlihy [10]. His idea was to allow each concurrent process to create a private copy of the object, apply its changes locally and then attempt to modify the shared root pointer of the object to point the private copy. For the wait-free transformation, he employed a similar mechanism to ours, in which each process writes in a special array the details of the operation it is going to perform, while other processes may access this array and assist slower peers to complete their operations.

Herlihy's construction has two very significant performance drawbacks. First, the copying can be very costly, especially for large objects. Second, the construction precludes disjoint-access parallelism since it requires an atomic update of a single root pointer. Thus, even if concurrent operations modify different parts of the structure, as happens in queues, they still will contend. Although many optimizations were proposed to address one or both of these limitations [3, 4, 7, 20], universal constructions are hardly considered practical. It is worth mentioning that the recent work by Chuong et al. [7] gives a refined construction for queues, which allows concurrent `enqueue` and `dequeue` operations. Their queue, however, is implemented as a statically-allocated array, thus the upper bound on the total number of elements that can be inserted into the queue has to be set upfront. In addition, their work does not provide any performance evaluation.

3. Wait-Free Queue Algorithm

We start by presenting the idea of our wait-free queue, including the scheme by which the queue operations are implemented (Section 3.1). Following that, we provide the full implementation in Java of our base algorithm (Section 3.2). Preferring the simplicity of the presentation of principal ideas over performance optimality, we defer the discussion of potential optimizations of the base algorithm to Section 3.3. Utilizing the fact that Java is a garbage-collected language, in our base algorithm we do not deal with memory management and problems related to that. We elaborate on how our algorithm can be implemented in other runtime environments, such as C++, in Section 3.4. In our code, we follow the style of the lock-free algorithm by Michael and Scott [19] as it appears in [11]. For simplicity, we assume the queue stores integer values, though the extension for any generic type is trivial.

3.1 The idea in a nutshell

Similarly to the queue in [19], our wait-free implementation is based on the underlying singly-linked list and holds two references to the head and tail of the list, respectively called `head` and `tail`. Our implementation significantly extends the helping technique already employed by Michael and Scott in their original algorithm [19]. Every thread t_i starting an operation on the queue chooses a phase number, which is higher than phases of threads that have previously chosen phase numbers for their operations (we will explain later how the phase number is chosen). Then it records this number, along with some additional information on the operation it is going to execute on the queue, in a special `state` array.

Next, t_i traverses the `state` array and looks for threads with entries containing a phase number that is smaller or equal than the one chosen by t_i. Once such a thread t_j is found (it can be also t_i itself), t_i tries to help it execute its operation on the queue, i.e., to insert t_j's node into the queue (`enqueue`) or to remove the first node from the queue on behalf of t_j (`dequeue`). The thread t_i learns the details on t_j's operation from the `state` array. Finally, when t_i has tried to help all other threads with a phase number not larger than it has, it can safely return the execution to the caller of the (`dequeue` or `enqueue`) operation. That is, t_i can be confident that its own operation on the queue has been completed either by t_i or by some other concurrently running and helping thread.

Our algorithm is carefully designed to handle concurrent assistance correctly, and in particular, to avoid applying the same operation more than once. Essentially, this is the most sophisticated part of our design. The idea is to break each type of operation (i.e., `dequeue` and `enqueue`) into three atomic steps, so that steps belonging to the same operation can be executed by different threads, yet they cannot interleave with steps of other concurrent operations of the same type. These steps form the implementation scheme of the operations, and are presented hereby. (Under "internal structure of the queue" we mean the underlying linked list along with the `head` and `tail` references).

1. Initial change of the internal structure of the queue, such that all concurrent threads performing the operation of the same type realize that there is some operation-in-progress. At the end of this step, the operation-in-progress is linearized.
2. Updating the entry in the `state` array belonging to the thread that invoked the operation-in-progress with the fact that the thread's operation was linearized.
3. Finalizing the operation-in-progress, fixing the internal structure of the queue.

For the `enqueue` operation, we utilize the lazy nature of the original implementation [19], where this operation is done in two distinct phases. There, a thread tries first to append a new node to the end of the underlying linked list and then updates `tail` to refer the newly appended node. The important aspect of this lazy implementation is that other threads do not execute the first phase of their `enqueue` operation until the `tail` reference is updated, ensuring that at most one node can be beyond the node referenced by `tail` in the underlying linked list. (We refer to such a node as *dangling*). This lazy implementation fits nicely into the scheme presented above: The initial change of the queue in Step (1) is the appending of a new node to the end of the underlying linked list. Following this step, other threads cannot start their `enqueue` operations, but only can learn about the existence of an operation-in-progress and assist its completion with the next two steps of the scheme.

```
 1: class Node {
 2:     int value;
 3:     AtomicReference<Node> next;
 4:     int enqTid;
 5:     AtomicInteger deqTid;
 6:     Node (int val, int etid) {
 7:         value = val;
 8:         next = new AtomicReference<Node>(null);
 9:         enqTid = etid;
10:         deqTid = new AtomicInteger(-1);
11:     }
12: }

13: class OpDesc {
14:     long phase;
15:     boolean pending;
16:     boolean enqueue;
17:     Node node;
18:     OpDesc (long ph, boolean pend, boolean enq, Node n) {
19:         phase = ph;
20:         pending = pend;
21:         enqueue = enq;
22:         node = n;
23:     }
24: }
```

Figure 1. Internal structures.

```
25: AtomicReference<Node> head, tail;
26: AtomicReferenceArray<OpDesc> state;

27: WFQueue () {
28:     Node sentinel = new Node(-1, -1);
29:     head = new AtomicReference<Node>(sentinel);
30:     tail = new AtomicReference<Node>(sentinel);
31:     state = new AtomicReferenceArray<OpDesc>(NUM_THRDS);
32:     for (int i = 0; i < state.length(); i++) {
33:         state.set(i, new OpDesc(-1, false, true, null));
34:     }
35: }

36: void help(long phase) {
37:     for (int i = 0; i < state.length(); i++) {
38:         OpDesc desc = state.get(i);
39:         if (desc.pending && desc.phase <= phase) {
40:             if (desc.enqueue) {
41:                 help_enq(i, phase);
42:             } else {
43:                 help_deq(i, phase);
44:             }
45:         }
46:     }
47: }

48: long maxPhase() {
49:     long maxPhase = -1;
50:     for (int i = 0; i < state.length(); i++) {
51:         long phase = state.get(i).phase;
52:         if (phase > maxPhase) {
53:             maxPhase = phase;
54:         }
55:     }
56:     return maxPhase;
57: }

58: boolean isStillPending(int tid, long ph) {
59:     return state.get(tid).pending && state.get(tid).phase <= ph;
60: }
```

Figure 2. Auxiliary methods.

In the original implementation [19], the `dequeue` operation is implemented in one atomic step (the update of `head`), and thus its adaption for our scheme is more complicated. We solve it by adding a field to each node of the underlying linked list: A thread starts the `dequeue` operation by writing its ID into this field at the first node of the list. More precisely, a thread t_i executing the `dequeue` operation on behalf of t_j writes the ID of t_j. This is the first step of our scheme. Other threads cannot remove nodes from the list before they assist the thread whose ID is written in the first node to complete the next two steps of the scheme.

Special care should be given to the case of an empty queue. Since a `dequeue` operation on an empty queue can be executed by t_i on behalf of another thread t_j, t_i cannot simply throw an exception (or return a special "empty" value), as this will occur in a wrong execution context. Thus, t_i has to indicate this special situation in t_j's entry of the `state` array. Also, to avoid t_i from racing with another thread t_k that helps t_j's `dequeue` operation and thinks that the queue is actually non-empty, we require t_k to update t_j's entry in `state` with a reference to the first node in the underlying list. This update has to take place before t_k executes the first step of the scheme described above, i.e., before t_k puts t_j's ID into the special field of the first node in the list. All further details are elaborated and exemplified pictorially in Section 3.2.

It remains to describe how t_i chooses its phase number. To provide wait-freedom, we need to ensure that every time t_i chooses a number, it is greater than the number of any thread that has made its choice before t_i. For this purpose, we chose a simple implementation in which t_i calculates the value of the maximal phase stored in the `state` array and chooses this value plus one. The idea is inspired by the doorway mechanism proposed by Lamport in his famous Bakery algorithm for mutual exclusion [15]. Alternatively, one may use an atomic counter, which can be incremented and read atomically. This idea is described in Section 3.3.

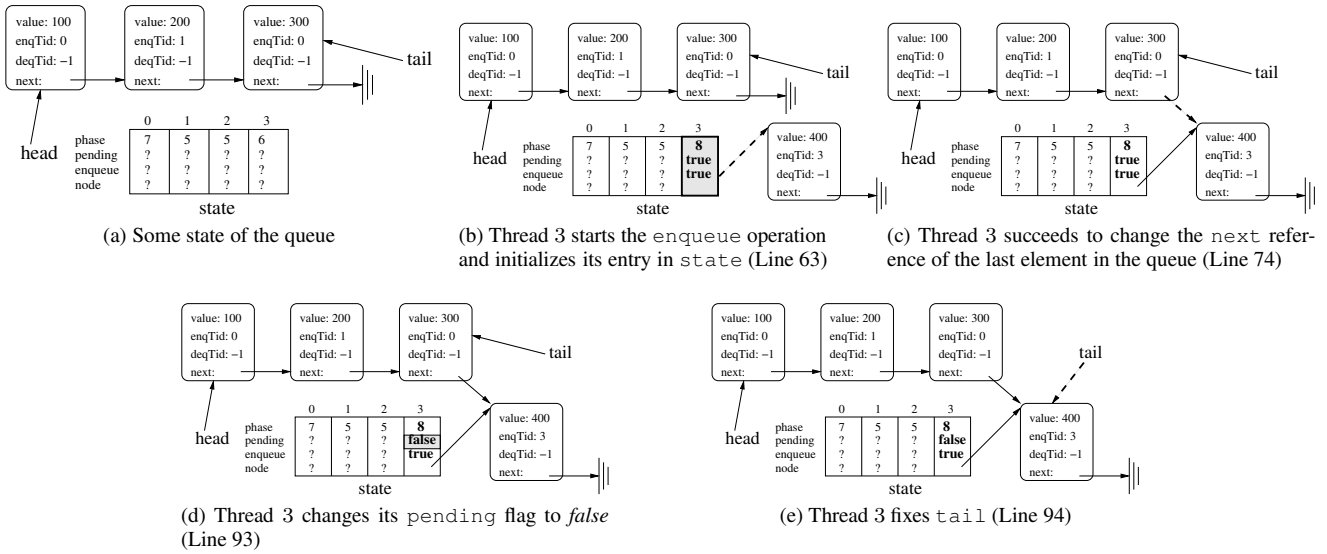
3.2 Implementation

Internal structures and auxiliary methods

The code for internal structures used by our queue implementation is given in Figure 1, while auxiliary methods are in Figure 2. Our queue uses a singly-linked list with a sentinel node as an underlying representation, and relies on two inner classes: `Node` and `OpDesc`. The `Node` class (Lines 1–12) is intended to hold elements of the queue's underlying list. In addition to the common fields, i.e., a value and atomic reference to the next element (implemented with `AtomicReference` class of Java), `Node` contains two additional fields: `enqTid` and `deqTid`. As their names suggest, these fields hold the ID of the thread that performs or has already performed (probably, helped by another thread) the insertion or removal of the node to/from the queue. (In the following, we refer to ID of a thread as *tid*).

In more detail, when a node is created by a thread in the beginning of the `enqueue` operation, the thread records its tid in the `enqTid` field of the new node. This field will be used by other threads to identify the thread that tries to insert that particular node into the queue, and help it if needed. Similarly, during the `dequeue` operation, the tid of the node trying to remove the first node of the queue is recorded in the `deqTid` field of the first node in the underlying linked list. Again, this field is used to identify and help the thread performing a `dequeue` operation. Notice that while `enqTid` is set by only one thread (the one that wants to insert a node into the queue), `deqTid` may be modified by multiple threads concurrently performing a `dequeue` operation. As a result, while the former field is a regular (non-atomic) integer, the latter field is implemented as an atomic integer.

The `OpDesc` class (Lines 13–24) defines an operation descriptor record for each thread. This record contains information about



(a) Some state of the queue

(b) Thread 3 starts the `enqueue` operation and initializes its entry in `state` (Line 63)

(c) Thread 3 succeeds to change the `next` reference of the last element in the queue (Line 74)

(d) Thread 3 changes its `pending` flag to *false* (Line 93)

(e) Thread 3 fixes `tail` (Line 94)

Figure 3. The flow of the `enqueue` operation performed solely by Thread 3.

the phase at which the thread has performed (or is performing right now) its last operation on the queue (`phase` field), the type of the operation (`enqueue` field), the flag specifying whether the thread has a pending operation (`pending` field), and a reference to a node with a meaning specific to the type of the operation (`node` field). In case of the `enqueue` operation, the `node` field holds a reference to a node with a value that the considered thread tries to insert into the queue. In case of the `dequeue` operation, this field refers to a node preceding the one with a value that should be returned from the operation (this will be clarified later in this section when the `dequeue` operation is explained).

In addition to the definition of the inner classes, the `Queue` class contains several fields. Being based on a linked list, it has (atomic) references to the head and the tail of the list (Line 25). Additionally, it contains an array of atomic references to the `OpDesc` records of all threads, called `state` (Line 26). The size of the array, denoted as `NUM_THRDS`, is assumed to be known and equal to the number of threads that might perform operations on the queue[2]. In the following, where it is not ambiguous, we refer to the entry of a thread t_i in the `state` array as *t_i's state*.

The auxiliary methods are straight-forward. The queue constructor (Lines 27–35) initializes the underlying linked list with one sentinel (dummy) node. In addition, it initializes the `state` array appropriately. `help()` (Lines 36–47) accepts a phase number. It runs through all operation descriptors and calls `help_enq()` or `help_deq()` methods (detailed later in this section) according to the operation type recorded in the descriptor. Notice that `help_enq()` and `help_deq()` are invoked only for operations that are still pending and have a phase smaller or equal than the one passed as a parameter to `help()`. `maxPhase()` method (Lines 48–57) simply runs through the `state` array and calculates the maximal phase written in the operation descriptors. `isStillPending()` method (Lines 58–60) accepts tid and phase and checks whether the thread with ID equal to tid has a pending operation in a phase smaller or equal than the passed *phase* parameter.

[2] More precisely, `NUM_THRDS` can be an upper bound, not necessarily strict, on the number of threads that might access the queue.

`enqueue` operation

The flow of the operation is presented in Figure 3. For simplicity, Figure 3 considers the case in which a thread executes `enqueue` without interference from other threads. The implementation is given in Figure 4.

When a thread t_i calls the `enqueue` operation (i.e., `enq()` method of the queue), it first calculates the phase for the operation by adding 1 to the result returned from `maxPhase()` (Line 62). Then it creates a new node to hold the value to be enqueued and updates the corresponding entry in the `state` array with a new operation descriptor record (Line 63). We assume here that each thread can access its tid, while the tid is a number in the range $[0, \ldots, \texttt{NUM\_THRDS}-1]$. In Section 3.3 we discuss how the latter can be relaxed. Note also that the new node is created with `enqTid` field set to the tid of the thread (i in our example). Later in the code, this field will be used to locate the entry of t_i in the `state` array in order to complete t_i's `enqueue` operation.

Next, t_i calls `help()` (Line 64), which was explained above. When this call returns, all operations having phase smaller or equal than the one chosen by t_i in Line 62 are linearized, including the current `enqueue` operation of t_i. Finally, t_i calls the `help_finish_enq()` method (Line 65), which ensures that when t_i returns from the call to `enq()`, the `tail` reference of the queue does not refer a node before the one that was just enqueued by t_i. This is needed for correctness, as explained later in this section.

Next, we describe the implementation of the `help_enq()` method given in Lines 67–84. This method is called from `help()` for pending `enqueue` operations. It accepts two parameters, which are the ID of a thread t_x (i.e., x) who has a pending `enqueue` operation and the phase of the thread t_y actually calling `help_enq()`. Notice that these can be the same thread (i.e., $x = y$) or two different threads. Among other things, both parameters are used to verify that the `enqueue` operation of t_x for which `help_enq()` was called is still pending.

The actual implementation of `help_enq()` is a customized version of the `enq()` method in [19]. It checks whether the queue is in a state in which `tail` actually refers to the last node in the underlying linked list (Lines 69–72). If so, it tries to insert the new node just behind the last node in the list (Line 74). With respect to

```
61: void enq(int value) {
62:   long phase = maxPhase() + 1;                          ▷ cf. Figure 3b
63:   state.set(TID, new
        OpDesc(phase, true, true, new Node(value, TID)));
64:   help(phase);
65:   help_finish_enq();
66: }

67: void help_enq(int tid, long phase) {
68:   while (isStillPending(tid, phase)) {
69:     Node last = tail.get();
70:     Node next = last.next.get();
71:     if (last == tail.get()) {
72:       if (next == null) {                    ▷ enqueue can be applied
73:         if (isStillPending(tid, phase)) {
74:           if (last.next.compareAndSet(next, state.get(tid).node)) {
                                                          ▷ cf. Figure 3c
75:             help_finish_enq();
76:             return;
77:           }
78:         }
79:       } else {                          ▷ some enqueue is in progress
80:         help_finish_enq();                        ▷ help it first, then retry
81:       }
82:     }
83:   }
84: }

85: void help_finish_enq() {
86:   Node last = tail.get();
87:   Node next = last.next.get();
88:   if (next != null) {
89:     int tid = next.enqTid;          ▷ read enqTid of the last element
90:     OpDesc curDesc = state.get(tid);
91:     if (last == tail.get() && state.get(tid).node == next) {
92:       OpDesc newDesc = new                          ▷ cf. Figure 3d
            OpDesc(state.get(tid).phase, false, true, next);
93:       state.compareAndSet(tid, curDesc, newDesc);
94:       tail.compareAndSet(last, next);               ▷ cf. Figure 3e
95:     }
96:   }
97: }
```

Figure 4. `enqueue` operation

the scheme in Section 3.1, this is the first step of the scheme. The new node is located using the `state` array and tid parameter of `help_enq()` (Line 74). If succeeded, or if there is already some node after the one pointed by `tail`, `help_finish_enq()` is invoked (Line 75 or 80, respectively).

As its name suggests, `help_finish_enq()` is called to finish the progressing `enqueue` operation. In the original implementation of [19], this means to update the `tail` reference to refer to the newly added node. In our case, we need to update also the state of the thread whose node was just inserted to the list in Line 74. The entry of this thread in the `state` array is located by the `enqTid` field written in the newly added node (Line 89). After verifying that the corresponding entry in the `state` array still refers to the newly added node (Line 91), a new operation description record with the `pending` flag turned off is created and written in Lines 92–93. This is the second step of the scheme in Section 3.1. The verification in Line 91 is required for correctness to avoid races between threads running `help_finish_enq()` concurrently. Afterwards, the `tail` reference is updated to refer to the newly added (and the last) node in the list (Line 94), which is the third step of the scheme. Notice that the CAS operation in Line 93 may succeed more than once for the same newly added node. Yet, utilizing the fact that only one node can be dangling, our implementation ensures that the `state` array remains consistent and the `tail` reference is updated only once per each node added to the list.

Having `help_finish_enq()` explained, we can argue why its invocation in Line 65 is necessary. Consider a thread t_i trying to apply an `enqueue` operation and being helped concurrently by another thread t_j. If t_j executes Line 93 and gets suspended, t_i might realize that its operation is already linearized (e.g., by checking the condition in Line 68), return to the caller of `enq()` and start another operation on a queue setting a new operation descriptor. In such a scenario, all further `enqueue` operations will be blocked (since the condition in Line 91 does not hold anymore) until t_j will resume and update the `tail` reference, breaking the wait-free progress property.

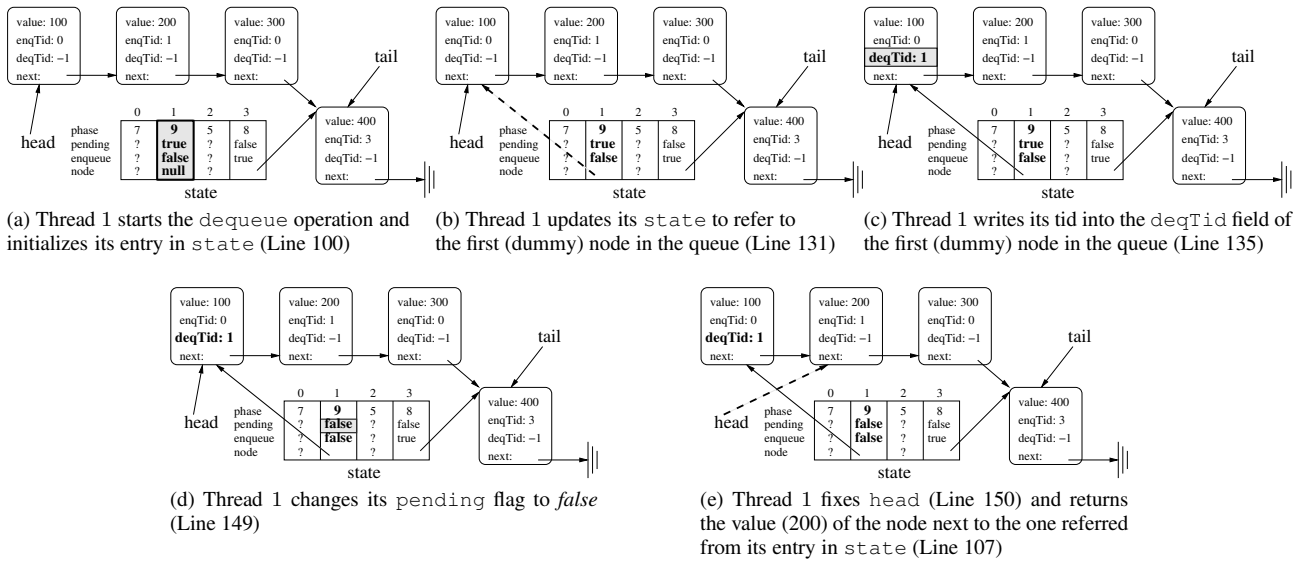
In addition, notice that both condition checks in Lines 68 and 73 are necessary for correctness. Removing the check in Line 68 (i.e., replacing this line with **while** (*true*) statement) will break the progress property, as a thread may always find some `enqueue` operation in progress and execute the call to `help_finish_enq()` at Line 80 infinitely many times. Along with that, removing the check in Line 73 will break the linearizability. This is because a thread t_i might pass the test in Line 68, get suspended, then resume and add an element to the queue, while at the same time, this element might have been already added to the queue by another concurrent thread during the time t_i was suspended.

`dequeue` operation

The flow of the operation is presented in Figure 5, while the code is given in Figure 6. As in the case of `enqueue`, a thread t_i starts the `dequeue` operation by choosing the phase number (Line 99), updating its entry in the `state` array (Line 100) and calling `help()` (Line 101). When the latter call returns, the current `dequeue` operation of t_i, as well as all operations having a phase not larger than t_i's phase, are linearized. Then t_i calls `help_finish_deq()` (Line 102), which ensures that when t_i returns from `deq()`, the `head` reference does not refer to a node with a `deqTid` field equal to i. Again, this is required for correctness. Finally, t_i examines the reference to a node recorded in its `state` (Lines 103–107). If it is null, then the operation was linearized when the queue was empty. Otherwise, t_i returns the value of the node following the one recorded in t_i's state in the underlying linked list. Intuitively, this is the value of the first real node in the queue at the instant t_i's `dequeue` operation linearizes, and the entry in the `state` array refers to the preceding sentinel at that time.

The code of `help_deq()`, called from `help()` for pending `dequeue` operations, is given in Lines 109–140. The parameters received by this method have the same meaning as in `help_enq()`. Similarly, its implementation is intended to be a customized version of the `deq()` method in [19]. Yet, although `deq()` in [19] is relatively simple, `help_deq()` appears to be slightly more complicated, carefully constructed to fit the scheme of Section 3.1.

For ease of explanation, let us assume first that the queue is not empty. In order to help t_i to complete the `dequeue` operation and remove an element from the queue, a thread t_j running `help_deq()` has to pass through four stages, illustrated in Figure 5: (1) place a reference from t_i's state to the first node in the underlying list, (2) update the `deqTid` field of the first node in the list with i, (3) update the `pending` field in the state of t_i to *false*, and (4) update `head` to refer to the next node in the underlying list. The first two stages are handled in `help_deq()`, while the last two are in the `help_finish_deq()` method. With respect to the scheme described in Section 3.1, Stages (2)–(4) implement the three steps of the scheme, while the additional Stage (1) is required to handle correctly the case of applying `dequeue` on an empty queue, as described later in this section.



(a) Thread 1 starts the `dequeue` operation and initializes its entry in `state` (Line 100)

(b) Thread 1 updates its `state` to refer to the first (dummy) node in the queue (Line 131)

(c) Thread 1 writes its tid into the `deqTid` field of the first (dummy) node in the queue (Line 135)

(d) Thread 1 changes its `pending` flag to *false* (Line 149)

(e) Thread 1 fixes `head` (Line 150) and returns the value (200) of the node next to the one referred from its entry in `state` (Line 107)

Figure 5. The flow of the `dequeue` operation performed solely by Thread 1 after Thread 3 finishes its `enqueue` operation in Figure 3.

The `dequeue` operation of t_i is linearized at the instant t_j (or any other concurrent thread running `help_deq()` with *tid* parameter equal to i) succeeds to complete Stage (2). We say that at that time, t_j *locks* the sentinel referred by `head`. (To avoid any confusion, we emphasize that here the lock has only a logical meaning). Before that happens, t_j examines the node referred from t_i's `state` (Line 129). If it is different from the head of the queue as was read in Line 111, t_j tries to complete Stage (1) (Lines 130–131) (otherwise, some other thread has done it before). If failed, t_j realizes that some other thread has updated t_i's `state` and thus, t_j resets its operation, returning to the beginning of the **while** loop (Line 132). If succeeded, or if t_i's `state` already refers to the head of the queue, t_j tries to complete Stage (2) (Line 135). If succeeded again, then t_j locks the sentinel. Otherwise, some other thread did that (maybe even for the same `dequeue` operation). In any case, t_j calls `help_finish_deq()` (Line 136), which is supposed to complete the `dequeue` operation of the thread that locks the sentinel. Notice the condition checks in Lines 110 and 128. Similarly to the checks in `help_enq()`, the check in Line 110 is necessary to ensure the wait-freedom, while the check in Line 128 is required for the correct linearizability.

When t_j finds that the queue might be empty, i.e., when `head` and `tail` refer to the same node, it checks whether there is some `enqueue` operation in progress (Line 116). If so, it helps first to complete this operation (Line 123) and returns to the beginning of the **while** loop. Otherwise, t_j realizes that the queue is empty, and tries to update t_i's state accordingly (Lines 119–120). This is where Stage (1), i.e., putting a reference from t_i's `state` to the first node in the list, is required. Otherwise, a race between two helping threads, one looking at an empty queue and the other on a non-empty queue, is possible.

It remains to explain the details of `help_finish_deq()`. This method completes the `dequeue` operation in progress. First, it tries to update the `pending` field in the state of a thread t_k whose tid is written in the node referred by `head` (Lines 148–149). Notice that just like in `help_finish_enq()`, the CAS operation in Line 149 may succeed more than once for the same `dequeue` operation in progress. Finally, `head` is updated (Line 150), completing the final Stage (4) of the `dequeue` operation in progress.

3.3 Optimizations and enhancements

One of the shortcomings of the base implementation presented in Section 3.2 is that the number of steps executed by each thread when there is no contention still depends on n, the total number of threads that might perform an operation on the queue. This dependence appears in the calculation of the phase number in the `maxPhase()` method and in the `help()` method, where the `state` array is traversed.

It is possible, however, to resolve this drawback by changing the implementation in the following way. First, in order to calculate the phase number, a queue might have an internal `maxPh` field, which would be read by each thread initiating an operation on the queue and increased by an atomic operation, such as CAS or *Fetch-and-Add*[3]. Second, a thread may traverse only a chunk of the `state` array in a cyclic manner in the `help()` method. That is, in the first invocation of `help()`, it would traverse indexes 0 through $k-1 \ mod \ n$ (in addition to its own index) for some $1 \leq k < n$, in the second invocation – indexes $k \ mod \ n$ through $2k-1 \ mod \ n$, and so on. Notice that this modification will preserve wait-freedom since a thread t_i may delay a particular operation of another thread t_j only a limited number of times, after which t_i will help to complete t_j's operation. Alternatively, each thread might traverse a random chunk of the array, achieving probabilistic wait-freedom. In any case, the running time of contention-free executions will be improved. In case of contention, however, when all n threads try to execute an operation on the queue concurrently, the operations may essentially take a number of steps dependent on n. Yet another option is to apply techniques of [2] to have the time complexity of the algorithm to depend on the number of threads concurrently accessing the queue rather than n.

In addition, our base implementation can be enhanced in several ways. First, notice that any update of `state` is preceded with an allocation of a new operation descriptor. These allocations might be wasteful (both from performance and memory consumptions aspects) if the following CAS operation fails while trying to update

[3] Notice that in the case of a CAS-based phase counter, a thread does not need to check the result of the CAS. A failure of its CAS operation would simply imply that another thread has chosen the same phase number.

```
98:  int deq() throws EmptyException {
99:    long phase = maxPhase() + 1;                          ▷ cf. Figure 5a
100:   state.set(TID, new OpDesc(phase, true, false, null));
101:   help(phase);
102:   help_finish_deq();
103:   Node node = state.get(TID).node;
104:   if (node == null) {
105:     throw new EmptyException();
106:   }
107:   return node.next.get().value;
108: }

109: void help_deq(int tid, long phase) {
110:   while (isStillPending(tid, phase)) {
111:     Node first = head.get();
112:     Node last = tail.get();
113:     Node next = first.next.get();
114:     if (first == head.get()) {
115:       if (first == last) {                          ▷ queue might be empty
116:         if (next == null) {                             ▷ queue is empty
117:           OpDesc curDesc = state.get(tid);
118:           if (last == tail.get() && isStillPending(tid, phase)) {
119:             OpDesc newDesc = new
                      OpDesc(state.get(tid).phase, false, false, null);
120:             state.compareAndSet(tid, curDesc, newDesc);
121:           }
122:         } else {                        ▷ some enqueue is in progress
123:           help_finish_enq();                    ▷ help it first, then retry
124:         }
125:       } else {                                      ▷ queue is not empty
126:         OpDesc curDesc = state.get(tid);
127:         Node node = curDesc.node;
128:         if (!isStillPending(tid, phase)) break;
129:         if (first == head.get() && node != first) {
130:           OpDesc newDesc = new                      ▷ cf. Figure 5b
                    OpDesc(state.get(tid).phase, true, false, first);
131:           if (!state.compareAndSet(tid, curDesc, newDesc)) {
132:             continue;
133:           }
134:         }
135:         first.deqTid.compareAndSet(-1, tid);        ▷ cf. Figure 5c
136:         help_finish_deq();
137:       }
138:     }
139:   }
140: }

141: void help_finish_deq() {
142:   Node first = head.get();
143:   Node next = first.next.get();
144:   int tid = first.deqTid.get();      ▷ read deqTid of the first element
145:   if (tid != -1) {
146:     OpDesc curDesc = state.get(tid);
147:     if (first == head.get() && next != null) {
148:       OpDesc newDesc = new                          ▷ cf. Figure 5d
              OpDesc(state.get(tid).phase, false, false,
              state.get(tid).node);
149:       state.compareAndSet(tid, curDesc, newDesc);
150:       head.compareAndSet(first, next);              ▷ cf. Figure 5e
151:     }
152:   }
153: }
```

Figure 6. `dequeue` operation

the `state` array (e.g., CAS in Lines 93 or 120). This issue can be easily solved by caching allocated descriptors used in unsuccessful CASes and reusing them when a new descriptor is required.

Second, when a thread finishes an operation on a queue, its operation descriptor remains to refer a node in the underlying linked list (unless it was a `dequeue` operation from an empty queue). This node might be considered later by the garbage collector as a live object, even though it might have been removed from the queue a long time ago. This issue can be solved by setting a dummy operation descriptor record into the state of the thread just before it exits the `deq()` and `enq()` methods. This dummy record should have a null reference in its `node` field.

Third, our base implementation lacks validation checks that might be applied before executing (costly) CAS operations. For example, we might check whether the `pending` flag is already switched off before applying CAS in Lines 93 or 149. Although such checks might be helpful in performance tuning [14], they would definitely complicate the presentation of our algorithm. As stated before, we preferred simple presentation of principal ideas over optimality where possible, leaving (minor) performance tuning optimizations for future work.

In the base version of our algorithm, we assume threads to have unique IDs in a range between 0 and some known constant bound. However, the same thread can use different IDs in subsequent operations on the queue as long as they do not collide with IDs of other threads concurrently accessing the queue. As a result, the assumption above can be relaxed: To support applications in which threads are created and deleted dynamically and may have arbitrary IDs, threads can get and release (virtual) IDs from a small name space through one of the known long-lived wait-free renaming algorithms (e.g., [1, 6]). Also, our algorithm employs a phase counter, which theoretically may wrap and harm the wait-free progress property of our implementation. Since this counter is implemented as a 64-bit integer, this possibility is impractical.

3.4 Memory management and ABA issues

The algorithm presented in Section 3.2 relies on the existence of a garbage collector (GC) responsible for memory management, and in particular, for dynamic reclamation of objects not in use. Since a wait-free GC is not known to exist, the presentation of our algorithm would not be complete without discussing how the algorithm can manage its memory in a wait-free manner.

Also, another important benefit of GC is the elimination of all occurrences of the ABA problem [11] that originate from early reclamation of objects. The ABA problem denotes the situation where a thread may incorrectly succeed in applying a CAS operation even though the contents of the shared memory have changed between the instant it read the old value from the memory and the instant it applied the CAS with a new value. Such a situation may occur if a thread was suspended between the two instants, while many insertions and deletions executed on the queue in the interim period brought the contents of the location read by the thread into the identical state. In garbage-collected languages, such as Java, the problem does not exist, since an object referenced by some thread cannot be reallocated for any other use.

In order to adapt our algorithm for runtime environments in which GC is not implemented, we propose to use the Hazard Pointers [18] technique. The technique is based on associating *hazard pointers* with each thread. These pointers are single-writer multi-reader registers used by threads to mark (point on) objects that they may access later. When an object is removed from the data structure (in our case, by `dequeue` operation), the special `RetireNode` method is called, which recycles the object only if there are no hazard pointers pointing on it.

The integration of the Hazard Pointers technique into our algorithm requires a small modification of the latter. Specifically, we need to add a field into the operation descriptor records to hold a value removed from the queue (and not just a reference to the sentinel through which this value can be located). This is in order to be able to call `RetireNode` right at the end of `help_deq()`, even though the thread that actually invoked the corresponding

dequeue operation might retrieve the value removed from the queue (e.g., execute Line 107) much later. Since our algorithm has a structure very similar to that of [19], further integration of the Hazard Pointers technique with our algorithm is very similar to the example in [18]. The exact details are out of scope of this short paper. We also notice that the same technique helps to prevent the ABA problem described above, and, since this technique is wait-free [18], our integrated solution remains wait-free.

4. Performance

We evaluated the performance of our wait-free queue comparing it to the lock-free queue by Michael and Scott [19], known as the most efficient lock-free dynamically allocated queue algorithm in the literature [11, 14, 24]. For the lock-free queue, we used the Java implementation exactly as it appears in [11]. We run our tests using three different system configurations: the first one consisted of an Intel blade server featuring two 2.5GHz quadcore Xeon E5420 processors operating under CentOS 5.5 Server Edition system, the second one is the same machine operating under Ubuntu 8.10 Server Edition system, while the third configuration consisted of a machine featuring two 1.6GHz quadcore Xeon E5310 processors operating under RedHat Enterprise 5.3 Linux Server. All machines were installed with 16GB RAM and were able to run concurrently 8 threads. All tests were run in Sun's Java SE Runtime version 1.6.0 update 22, using the HotSpot 64-Bit Server VM, with `-Xmx10G -Xms10G` flags.

In addition to the base version detailed in Section 3.2, we have also evaluated two optimizations mentioned in Section 3.3:

1. In each operation on the queue, a thread t_i tries to help only one thread, choosing the candidates in a cyclic order of the entries in the `state` array. As in the base version, the helping is actually done only if the candidate has a pending operation with a phase smaller or equal than t_i's.
2. The phase number is calculated using an atomic integer, i.e., each thread gets the value of that integer plus one and tries to increment it atomically using CAS, instead of traversing the `state` array in `maxPhase()`.

As mentioned in Section 3.3, these two modifications preserve wait-freedom. Following the methodology of Michael and Scott [19] and of Ladan-Mozes and Shavit [14], we evaluated the performance of the queue algorithms with the following two benchmarks:

- `enqueue-dequeue` pairs: the queue is initially empty, and at each iteration, each thread iteratively performs an `enqueue` operation followed by a `dequeue` operation.
- 50% `enqueues`: the queue is initialized with 1000 elements, and at each iteration, each thread decides uniformly at random and independently of other threads which operation it is going to execute on the queue, with equal odds for `enqueue` and `dequeue`.

We measured the completion time of each of the algorithms as a function of the number of concurrent threads. For this purpose, we varied the number of threads between 1 and 16 (i.e., up to twice the number of available cores). Each thread performed $1,000,000$ iterations. Thus, in the first benchmark, given the number of threads k, the number of operations is $2000000 \cdot k$, divided equally between `enqueue` and `dequeue`. In the second benchmark, the number of operations is $1000000 \cdot k$, with a random pattern of roughly 50% of `enqueues` and 50% of `dequeues`. Each data point presented in our graphs is the average of ten experiments run with the same set of parameters. The standard deviation of the results was negligible, and thus not shown for better readability.

The results for the first benchmark for each of the three system configurations we worked with are presented in Figure 7. The results reveal that the relative performance of the concurrent algorithms under test is intimately related to the system configuration. While in the RedHat and Ubuntu-operated machines, the lock-free algorithm is an unshakable winner, in the CentOS-operated machine its superiority ends when the number of threads approaches the number of available cores. After that point, the performance of the lock-free algorithm in the CentOS-operated machine falls behind the optimized wait-free version, even though our wait-free algorithms are expected for longer time needed for the bookkeeping of the `state` and trying to help slower threads. It is also worth noting that the relative performance of the concurrent algorithms in the other two system configurations is not identical. While on the RedHat-operated machine, the ratio of the optimized wait-free algorithm to the lock-free algorithm remains around 3, in the Ubuntu-operated machine this ratio continuously decreases with the number of threads, approaching 2.

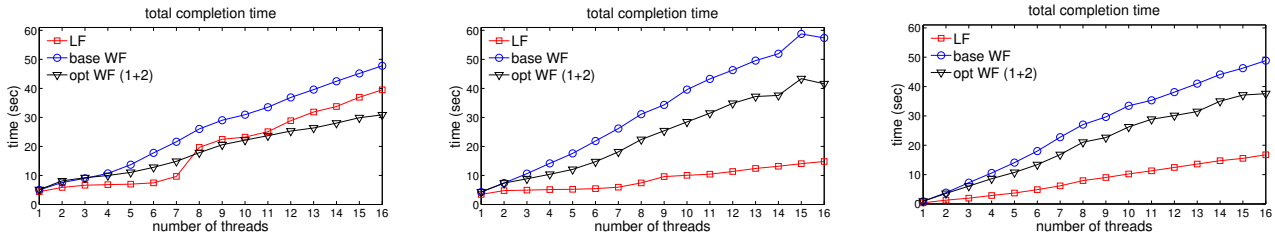
Figure 8 presents results for the second benchmark for each of the three system configurations. The total completion time for all algorithms exhibits the similar behavior as in the first benchmark, but is roughly 2 times smaller. This is because this benchmark has only a half of the total number of operations of the first benchmark. The relative performance of the lock-free and wait-free algorithms is also similar. Interestingly, the CentOS-operated machine consistently exhibits better performance for the optimized wait-free algorithm when the number of threads increases.

When comparing the performance of the base wait-free algorithm with its optimized version, one can see that the latter behaves better as the number of threads increases (Figures 7 and 8). As discussed in Section 3.3, this happens since the optimized version does not have to traverse the `state` array, which increases in size with the increase in the number of threads. To evaluate which of the two optimizations has greater impact on the performance improvement, we implemented each of them separately and compared the performance of all four variations of our wait-free algorithm: the base (non-optimized) version, two versions with one of the optimizations each, and the version featuring both optimizations. Due to lack of space, we present only results for the `enqueue-dequeue` benchmark and only for the CentOS and RedHat-based configurations (Figure 9). We note that the relative performance in the other benchmark and in the third system configuration is the same.

As Figure 9 suggests, the performance gain is achieved mainly due to the first optimization, i.e., the modified helping mechanism in which a thread helps to at most one other thread when applying an operation on the queue. This is because this optimization reduces the possibility for scenarios in which all threads try to help the same (or a few) thread(s), wasting the total processing time. The impact of the second optimization (the maintenance of the phase number with CAS rather than by traversing `state`) is minor, yet it increases with the number of threads.

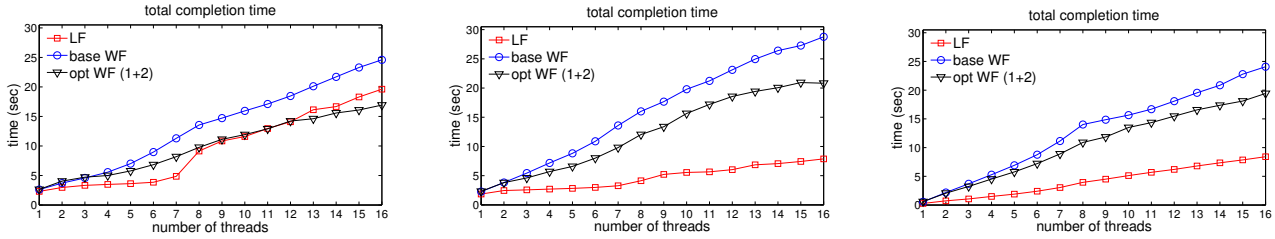
In addition to the total running time, we compared the space overhead of the algorithms, that is the amount of heap memory occupied by queues and by threads operating on them. For this purpose, we used the `--verbocegc` flag of Java, which forces Java to print statistics produced by its GC. These statistics include information on the size of live objects in the heap. We run the `enqueue-dequeue` benchmark with 8 threads, while one of the threads periodically invoked GC. To calculate the space overhead, we took the average of these samples (nine samples for each run). We varied the initial size of the queue between 1 and $10,000,000$ elements, in multiples of 10. To produce a data point, we run ten experiments with the same set of parameters and took the average.

Figure 10 shows the space overhead of the base and optimized versions of wait-free algorithms relatively to the lock-free one. The



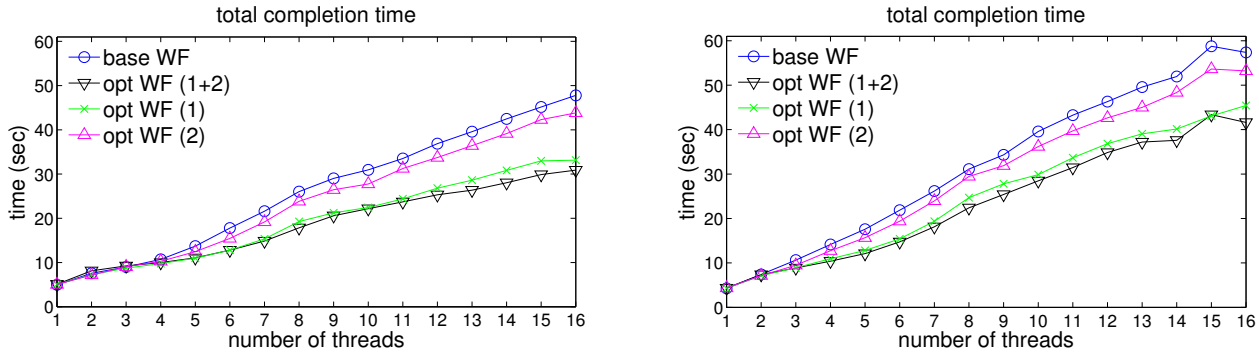
(a) CentOS-operated machine (b) RedHat-operated machine (c) Ubuntu-operated machine

Figure 7. Performance results of the `enqueue-dequeue` benchmark.



(a) CentOS-operated machine (b) RedHat-operated machine (c) Ubuntu-operated machine

Figure 8. Performance results of the 50% `enqueues` benchmark.



(a) CentOS-operated machine (b) RedHat-operated machine

Figure 9. The impact of optimizations in the `enqueue-dequeue` benchmark.

shown results were produced on the RedHat-operated machine, while the results for the other two machines are similar. It follows that for small queues, the size of live objects maintained by wait-free algorithms is comparable to that of the lock-free implementation. This is because in these cases, the heap is dominated by objects, which are not part of queues. When the queue size increases, the ratio of sizes of maintained live objects reaches 1.5, meaning that our wait-free algorithms incur roughly 50% more space overhead over the lock-free one. This is because each node in the wait-free queue needs additional space for `deqTid` and `enqTid` fields.

5. Correctness

Due to lack of space, we provide only a sketch of a proof that our algorithm implements a concurrent wait-free queue. We start by describing the computational model assumed by our algorithm. Next, we briefly describe the proof of linearizability and wait-freedom property of our algorithm. In the process, we explain the semantics of our queue and define the linearization points for its `enqueue` and `dequeue` operations.

5.1 Model

Our model of multithreaded concurrent system follows the linearizability model of [12] and assumes an asynchronous shared memory system, where programs are executed by n deterministic threads, which communicate by executing atomic operations on shared variables from some predefined, finite set. Threads are run on processors, and the decision of which thread will run on which processor is performed solely by a scheduler. It is very likely that the number of available processes is much smaller than n, and execution of any thread may be paused at any time and for arbitrary long due to page fault, cache miss, expired time quantum, etc. We assume that each thread has an ID, denoted as tid, which is a value between 0 and $n-1$. In Section 3.3 we discuss how to relax this assumption. In addition, we assume each thread can access its tid and n.

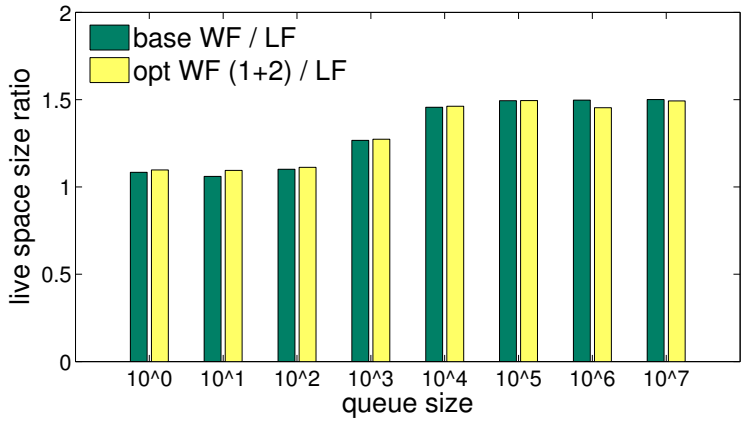


Figure 10. Space overhead evaluation as a function of the initial queue size.

When scheduled to run, a thread performs a sequence of computational steps. Each step may be either a local computation or an invocation of at most one atomic operation on a shared variable. We assume that our shared memory supports atomic reads, writes and compare-and-swap operations. The latter, abbreviated as CAS, is defined as follows: CAS(v, exp, new) changes the value of the shared variable v to new if and only if its value just before CAS is applied is equal to exp. In this case, CAS returns *true* and we say it was *successful*. Otherwise, the value of v is not changed, CAS returns *false* and we say it was *unsuccessful*.

A *configuration* of the system is a vector of a finite size that stores the states of n threads and the values of all shared variables. The state of a thread includes the values of thread's local variables, registers and the program counter. A (finite or infinite) sequence of steps, starting from an *initial configuration*, forms an *execution*. In the initial configuration, all shared variables have predefined initial values and all threads are in their initial states.

A concurrent queue is a data structure whose operations are linearizable [12] to those of a sequential queue. A sequential queue is a data structure that supports two operations: `enqueue` and `dequeue`. The first one accepts an element as an argument and inserts it into the queue. The second one does not accept arguments, removes and returns the oldest value from the queue. In the case of an empty queue, `dequeue` returns a special value (or throws an exception) and the queue remains unchanged.

5.2 Linearizability

We refer to a `dequeue` operation that returns a value as *successful*. If a `dequeue` operation ends by throwing an exception, we refer to it as *unsuccessful*. The `enqueue` operation is always successful. In the following, we define the linearization points for each of the queue operations. Notice that the source lines mentioned in the definition below can be executed either by the thread t_i that invoked the corresponding operation or by another concurrent thread t_j that tries to help t_i by running either `help_enq()` or `help_deq()` method with the tid parameter set to i.

DEFINITION 1. *The linearization points for each of the queue operations are as follows:*

- `Enqueue` *operation is linearized at the successful CAS in Line 74.*
- *Successful* `dequeue` *operation is linearized at the successful CAS in Line 135.*
- *Unsuccessful* `dequeue` *operation is linearized in Line 112.*

To prove the correctness of the linearization points defined above, we recall that our queue is based on a singly-linked list, represented by `head` and `tail` references, where the node referenced by `head` is called *dummy*. We define the state of the queue as a sequence of values of nodes in the list, starting from the node referenced by the `next` reference of the dummy node. (If such a node does not exist, that is the `next` reference of the dummy node is null, we say that the queue is empty).

In the full proof, we need to show that new nodes are always inserted at the end of the list (that is, after the last node reachable from `head` in the list), while the insertion order is consistent with the linearization order of `enqueue` operations. In addition, we need to show that nodes are always removed from the beginning of the list, while the order of removals is consistent with the linearization order of `dequeue` operations. This can be done in a way similar to [19] by careful inspection of the source lines that change the structure of the list .

In the second part of the proof, we need to show that each operation is linearized exactly once. In contrast to most lock-free implementations, including [19], in our case this statement is not trivial due to the concurrent assistance employed by our algorithm. Two central Lemmas in this part are stated below (their proof is omitted).

LEMMA 1. *In any execution and for any* `enqueue` *operation with value v_i invoked by thread t_j, the following steps occur in the stated order, and each of them occurs exactly once:*

1. *A node with value v_i is appended at the end of the linked list (Line 74).*
2. *The* `pending` *flag in the state of t_j is switched from* true *to* false *(Line 93).*
3. `Tail` *is updated to refer the node with value v_i (Line 94).*

LEMMA 2. *In any execution and for any successful* `dequeue` *operation invoked by thread t_j, the following steps occur in the stated order, and each of them occurs exactly once:*

1. *The* `deqTid` *field of a node referenced by* `head` *is updated with j (Line 135).*
2. *The* `pending` *flag in the state of t_j is switched from* true *to* false *(Line 149).*
3. `Head` *is updated to refer a node next to the one that has j in its* `deqTid` *field (Line 150).*

Notice that the steps in Lemmas 1 and 2 correspond to the steps in the scheme presented in Section 3.1

5.3 Wait-freedom

To prove wait-freedom property, we have to show that every call to either `enqueue` or `dequeue` returns in a bounded number of steps. In our implementation, both `enq()` and `deq()` call `help()`, which iterates a finite number of times (equal to the size of the `state` array), invoking `help_enq()` or `help_deq()` at most once in each iteration. Thus, in order to prove wait-freedom, we need to show that any invocation of `help_enq()` or `help_deq()` returns after a bounded number of steps.

We prove first that our implementation is lock-free. For this purpose, we need to show that every time a thread t_i executes an iteration of the **while** loop in either `help_enq()` or `help_deq()`, some thread t_j makes progress. That is, either one of the steps of Lemmas 1 or 2 occurs, or t_j finds the queue empty and executes a successful CAS in Line 120. This claim can be relatively easily concluded from the code.

To complete the proof of wait-freedom, we need to show that the number of operations that may linearize before any given operation is bounded. This follows directly from the way threads choose phase numbers, which is similar to the doorway mechanism in the Bakery algorithm [15]. This mechanism ensures that a thread t_i starting an `enq()` or a `deq()` method after t_j has chosen its phase number, will not finish the method before t_j's operation is linearized. Thus, after a bounded number of steps, a thread executing `help_enq()` or `help_deq()` will compete only with threads running the same method and helping the same operation.

6. Conclusions

FIFO queue is a fundamental data structure, found in many software system. Until now, no practical wait-free implementation of the queue was known. In this paper, we have shown the first such implementation that enables an arbitrary number of concurrent enqueuers and dequeuers. The significance of wait-freedom is in its ability to ensure a bounded execution time for each operation.

We have conducted performance evaluation of our implementation comparing it with a highly efficient lock-free algorithm [19]. The results reveal that the actual performance impact of wait-freedom is tightly coupled with the actual system configuration. In particular, we show that although our design requires more operations to provide the wait-free progress guarantee, it can beat the lock-free algorithm in certain system configurations and keep comparable performance ratios in others.

Acknowledgments

We would like to thank Roy Friedman for helpful discussions on the subject. Also, we would like to thank anonymous reviewers whose valuable comments helped to improve the presentation of this paper.

References

[1] Y. Afek and M. Merritt. Fast, wait-free (2k-1)-renaming. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 105–112, 1999.

[2] Y. Afek, D. Dauber, and D. Touitou. Wait-free made fast. In *Proc. ACM Symposium on Theory of Computing (STOC)*, pages 538–547, 1995.

[3] J. Alemany and E. W. Felten. Performance issues in non-blocking synchronization on shared-memory multiprocessors. In *Proc. ACM Symposium on Principles of Distributed Computing (PODC)*, pages 125–134, 1992.

[4] J. H. Anderson and M. Moir. Universal constructions for large objects. *IEEE Trans. Parallel Distrib. Syst.*, 10(12):1317–1332, 1999.

[5] R. J. Anderson and H. Woll. Wait-free parallel algorithms for the union-find problem. In *Proc. ACM Symposium on Theory of Computing (STOC)*, pages 370–380, 1991.

[6] H. Attiya and A. Fouren. Adaptive and efficient algorithms for lattice agreement and renaming. *SIAM J. Comput.*, 31(2):642–664, 2001.

[7] P. Chuong, F. Ellen, and V. Ramachandran. A universal construction for wait-free transaction friendly data structures. In *Proc. ACM Symposium on Parallel Algorithms (SPAA)*, pages 335–344, 2010.

[8] M. David. A single-enqueuer wait-free queue implementation. In *Proc. Conf. on Distributed Computing (DISC)*, pages 132–143, 2004.

[9] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[10] M. Herlihy. A methodology for implementing highly concurrent objects. *ACM Trans. Program. Lang. Syst.*, 15(5):745–770, 1993.

[11] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[12] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[13] P. Jayanti and S. Petrovic. Logarithmic-time single deleter, multiple inserter wait-free queues and stacks. In *Proc. Conf. on Found. of Soft. Technology and Theor. (FSTTCS)*, pages 408–419, 2005.

[14] E. Ladan-Mozes and N. Shavit. An optimistic approach to lock-free fifo queues. *Distributed Computing*, 20(5):323–341, 2008.

[15] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974.

[16] L. Lamport. Specifying concurrent program modules. *ACM Trans. Program. Lang. Syst.*, 5(2):190–222, 1983.

[17] H. Massalin and C. Pu. A lock-free multiprocessor OS kernel. Technical report CUCS-005-91, Computer Science Department, Columbia University, 1991.

[18] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, 2004.

[19] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. ACM Symposium on Principles of Distributed Computing (PODC)*, pages 267–275, 1996.

[20] M. Moir. Laziness pays! Using lazy synchronization mechanisms to improve non-blocking constructions. In *Proc. ACM Symposium on Principles of Distributed Computing (PODC)*, pages 61–70, 2000.

[21] M. Moir, D. Nussbaum, O. Shalev, and N. Shavit. Using elimination to implement scalable and lock-free FIFO queues. In *Proc. ACM Symposium on Parallel Algorithms (SPAA)*, pages 253–262, 2005.

[22] S. Moran, G. Taubenfeld, and I. Yadin. Concurrent counting. In *Proc. ACM Symposium on Principles of Distributed Computing (PODC)*, pages 59–70, 1992.

[23] K. R. Treiber. Systems programming: Coping with parallelism. Technical report RJ-5118, IBM Almaden Research Center, 1986.

[24] P. Tsigas and Y. Zhang. A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems. In *Proc. ACM Symposium on Parallel Algorithms (SPAA)*, pages 134–143, 2001.

[25] J. D. Valois. Implementing lock-free queues. In *Proc. 7th Int. Conf. on Parallel and Distributed Computing Systems*, pages 64–69, 1994.