A Methodology for Creating Fast Wait-Free Data Structures *

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

Lock-freedom is a progress guarantee that ensures overall program progress. Wait-freedom is a stronger progress guarantee that ensures the progress of each thread in the program. While many practical lock-free algorithms exist, wait-free algorithms are typically inefficient and hardly used in practice. In this paper, we propose a methodology called *fast-path-slow-path* for creating efficient wait-free algorithms. The idea is to execute the efficient lock-free version most of the time and revert to the wait-free version only when things go wrong. The generality and effectiveness of this methodology is demonstrated by two examples. In this paper, we apply this idea to a recent construction of a wait-free queue, bringing the wait-free implementation to perform in practice as efficient as the lock-free implementation. In another work, the fast-path-slow-path methodology has been used for (dramatically) improving the performance of a wait-free linked-list.

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 data structures, non-blocking synchronization, wait-free queues, lock-free algorithms

1. Introduction

The evolution of multi-core systems necessitates the design of scalable and efficient concurrent data structures. A common approach to achieve these goals is by constructing *non-blocking* algorithms. Such algorithms ensure that no thread accessing the data structure is postponed indefinitely while waiting for other threads that operate on that data structure.

Most non-blocking data structure implementations provide a *lock-free* progress guarantee, ensuring that among all threads that try to apply an operation, at least one will succeed. While some lock-free algorithms are quite scalable and efficient [7, 20, 24], they all allow workloads in which all but one thread starve. Such workloads cannot occur with *wait-free* algorithms, which ensure that each thread applies its operation in a bounded number of steps, independently of what other threads are doing. This property is valuable in real-time systems, operating systems or systems operating under a service level agreement, with a (hard or soft) deadline on the time required to complete each operation. It is valuable as well in heterogeneous systems, in which threads may execute at different speeds and where faster threads may repeatedly delay slower ones, as might happen, for example, in systems composed of different computing units, such as CPUs and GPUs.

In practice, while having desirable properties, wait-free algorithms are rightly considered inefficient and hard to design [6, 13]. Their inefficiency is said to be largely attributable to the helping mechanism [12], which is a key mechanism employed by most wait-free algorithms. This mechanism controls the way threads help each other to complete their operations, and usually leads to complicated algorithms, as each operation must be ensured to be applied exactly once. In practice, this often results in the usage of a greater number of expensive atomic operations, such as compare-and-swap (CAS), which degrade the performance of wait-free algorithms even when contention is low.

Moreover, most helping mechanisms known to us suffer from two problematic properties. First, upon starting an operation, a thread immediately begins to help other threads, sometimes interfering with their operations and almost always creating higher contention in the system. In most cases, the helped threads could have finished their operations by themselves if only they were given some time to execute without help. Second, the helping mechanisms are designed to operate sequentially, in the sense that all concurrently running threads help other operations in exactly the same order. These two properties cause much useless work due to high redundancy.

In this work, we propose a general methodology, denoted *fast-path-slow-path*, for constructing wait-free data structures. This methodology strives to be as scalable and fast as lock-free algorithms, while guaranteeing a bound on the number of steps required to complete each operation. To accomplish these goals, each operation is built from *fast* and *slow* paths, where the former ensures good performance, while the latter serves as a fall-back to achieve wait-freedom. Normally, the fast path is a customized version of a lock-free algorithm, while the slow path is a customized version of a wait-free one. A thread makes several attempts to apply an operation on the fast path; only if it fails to complete, it switches to the slow path, where the completion is guaranteed. We stress that our design allows threads to execute operations on the fast and slow paths in parallel. In particular, a thread that fails to apply its operation on the fast path does not cause all other threads to waste time waiting for its operation to be completed. Moreover, in contrast to the common definition of a fast path as "a shortcut through a complex algorithm taken by a thread running alone"[13], our design allows several threads to run on the fast path and finish their operations concurrently.

* 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.

PPoPP'12, February 25–29, 2012, New Orleans, Louisiana, USA.

Similarly to other wait-free algorithms, our methodology employs a helping mechanism, but one that avoids the drawbacks discussed above. In particular, the helping is *delayed* and *parallel*. The first property means that threads will delay any helping attempts when concurrent operations exist. The opportunistic idea is that during that delay, the contending operation might be finished, and no further help will be required. The latter property means that threads that do decide to help will attempt to help different operations, reducing the contention and redundant work created by sequential helping.

The applicability and effectiveness of our methodology is demonstrated by two examples. First, we use it to create a fast FIFO wait-free queue. The queue design is implemented in Java[1]. It is evaluated and compared to the lock-free queue by Michael and Scott [20], which is considered one of the most efficient and scalable lock-free queue implementations [13, 16, 25]. We also compare it to the recent wait-free queue by Kogan and Petrank [15]. The evaluation shows that along with the proven wait-free progress guarantee, the queue constructed in this paper delivers in practice the same performance as the queue of Michael and Scott, while both queues perform substantially better than the recent wait-free queue of Kogan and Petrank. As a second example, the fast-path-slow-path methodology has been also used on the wait-free linked-list independently published in these proceedings [23]. There too, this methodology yields a dramatic improvement in the algorithm's performance, while maintaining the wait-free progress guarantee.

In addition to making wait-free algorithms as fast as lock-free ones, the fast-path-slow-path methodology can be used to provide different level of progress guarantees to different entities. For example, it can be used to run real-time threads side-by-side with non-real-time threads, as the real-time threads use the slow path (or both paths) to obtain a predictable maximum response time, whereas the non-real-time threads only use the fast path without operation-level progress guarantee. Another use of this methodology is in distinguishing phases in the execution. Sometimes emergent phases require predictable guaranteed progress for each thread, while at other times, the system can run only the fast path to achieve fast execution in practice with no such progress guarantee. The interoperability of the two paths may be useful in any other scenario that requires different progress guarantees according to various execution parameters.

2. Related work

The idea of designing a concurrent algorithm with a fast path intended for no contention and a slow path for a contended case has been used in various domains of computer science. For example, it appears in solutions for the mutual exclusion problem [2, 17, 26], write barrier implementations in garbage collectors [18], and in implementations of composite locks [13].

Despite (and, maybe, due to) having strong progress guarantees, very few explicitly designed wait-free data structures are known (for the few that do exist, see [15, 23] and references therein). A common generic method for constructing wait-free data structures is to apply *universal constructions*, originally proposed by Herlihy[10, 13]. (See [3] for an updated survey of various improvements proposed since then.) Such constructions provide a generic method for transforming any sequential data structure implementation into a linearizable wait-free implementation. The wait-freedom is usually achieved by using a special *announce* array, in which threads write details on the operations they intend to apply. The contending threads traverse this array in a particular order and help pending operations to complete. Threads help other threads in a way that essentially reduces concurrency and creates large redundancy, especially when the number of contending threads is high: All threads try to apply the same operations, and in exactly the same order. In our work we show that trying to apply operations before involving other threads (i.e., before using the *announce* array) is crucial to the good performance of a wait-free algorithm. Moreover, the helping mechanism proposed in this paper allows parallel and delayed helping, eliminating the drawbacks of most previous wait-free constructions.

A related idea was considered by Moir [21], who proposed to implement wait-free transactions by utilizing a lock-free version of a multi-word CAS operation, denoted MWCAS[2]. A wait-free transaction optimistically applies a MWCAS operation; if it fails, it asks help from a transaction that did succeed. Our methodology is different in many aspects. Most importantly, while Moir's transactions always go through the costly MWCAS operation, which requires, among other things, (logically) locking and unlocking all words referenced by the transaction using expensive atomic operations, our methodology lets most operations finish on the (efficient) fast path, especially in the low contention scenario. The fast path is in practice as fast as the underlying lock-free implementation (e.g., in the queue shown in Section 4, the `dequeue` operation on the fast path requires just a single CAS).

Recently, Fatourou and Kallimanis [5] suggested a wait-free universal construction, where they utilize the idea of *operation combining* [8, 27]. There, a thread accessing the data structure creates a copy of the state of the global object, applies its operation along with all other pending operations of threads accessing the data structure concurrently, and tries to update the shared pointer (to the state of the simulated object) to point to the local (modified) copy of the object's state. This technique produces efficient wait-free algorithms for data structures having a small state (in particular, the queue and the stack), but is not efficient for data structures with larger states (such as linked-lists, trees, or hash tables). Moreover, the construction in [5] builds on the availability of a Fetch&Add atomic primitive that has an additional "wait-free" guarantee in the hardware: it is required that when several Fetch&Add operations conflict, they will be served in a starvation-free manner. This Fetch&Add instruction, especially with the additional guarantee, is not universally supported. Without this primitive, the construction becomes lock-free, and the performance degrades significantly [5].

3. The fast-path-slow-path methodology

Our methodology for constructing fast wait-free data structures is shown schematically in Figure 1. In a nutshell, the idea is that: each operation is built from a *fast path* and a *slow path*, where the former is a version of a lock-free implementation of that operation, and the latter is a version of a wait-free implementation. Both implementations are customized to cooperate with each other. Each thread first tries to apply its operation using the fast path. In most lock-free algorithms known to us, a thread tries to apply its operation by performing CAS (or equivalent) primitives on shared variables until it succeeds; if it fails, the thread retries the operation from the beginning (or some other earlier stage) (e.g., [4, 7, 9] and many others). In our approach, we limit the number of such retries by a global parameter, called `MAX_FAILURES` (denoted by X in Figure 1).

If a thread succeeds in applying its operation on the data structure in less than `MAX_FAILURES` trials, it finishes (i.e., returns the execution to the caller of that operation). If a thread fails in `MAX_FAILURES` trials, it realizes there is high contention and moves to the slow path. There, it publishes its intention to apply an opera-

[1] We also implemented our design in C (cf. Section 4.4).

[2] Moir provides an implementation of MWCAS, which he calls "conditionally wait-free", i.e., a lock-free implementation that accepts a call-back for an external helping mechanism. See [21] for details.

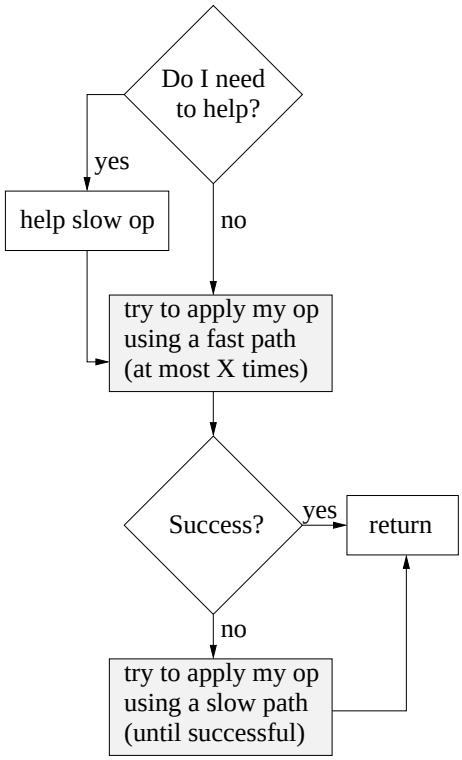


Figure 1. The fast-path-slow-path methodology.

tion in a special `state` array. This array holds one entry per thread, and contains information on the type of operation being applied and a phase number. The latter is a simple counter increased by the thread every time it enters the slow path. After the operation is published in `state`, the thread keeps trying to apply it by following the slow path until it succeeds.

To achieve wait-freedom, we need to ensure that the number of trials in the slow path is bounded. For this purpose, we employ a *delayed helping mechanism*, which is based upon an auxiliary array of helping records. Thus, in addition to an entry in the `state` array, each thread maintains a helping record rec_i. There, it stores the ID of another thread, $curTid_i$, and a phase number, $lastPhase_i$. When a thread t_i starts an operation on the data structure, and before it enters the fast path, it checks whether the thread $curTid_i$ is trying to apply an operation with the same phase number as was recorded earlier in $lastPhase_i$. If so, t_i helps $curTid_i$ first to apply the operation. In any case, t_i updates its helping record with the ID of the next thread (i.e., $curTid_i + 1$ modulo the number of threads) and with the phase number of that next thread, which is read from the `state` array. Only afterwards does t_i proceed with the fast path and, if necessary, with the slow path (cf. Figure 1).

Note that in contrast to many wait-free algorithms (e.g., [3, 5, 11, 15, 21]), this helping mechanism does not require a thread to help another concurrently running operation the moment the thread realizes there is contention. Rather, the helping is delayed to the next time the thread accesses the data structure, in the hope that the contending operation will be finished by that time and no further helping will be necessary.

In order to reduce the amortized management cost of helping records, we enhance the helping mechanism further by adding a counter, called $nextCheck_i$, to each helping record rec_i. Each time a thread t_i creates a new helping record, t_i sets the $nextCheck_i$ counter to a predefined `HELPING_DELAY` constant. On t_i's next operations on the data structure, t_i decrements the $nextCheck_i$ counter for each operation. Only when this counter reaches zero, does t_i check whether its help to $curTid_i$ is required. Thus, t_i actually checks whether its help is required only once in its `HELPING_DELAY` operations.

4. Fast wait-free queue

In this section, we show how the design methodology presented in Section 3 is applied to construct a fast wait-free queue. The design of the fast path and the slow path is based on the lock-free queue by Michael and Scott [20] (referred to hereafter as the MS-queue) and the recent wait-free queue by Kogan and Petrank [15] (referred to as the KP-queue), respectively. A brief description of the key design features of these two algorithms can be found in Appendix A.

Since the design of the MS-queue was published, several papers have proposed various optimized lock-free queues (e.g., [16, 22]). Yet, we chose to base our example on the MS-queue mainly due to its simplicity. We stress that the main purpose of this work is to create a wait-free implementation that is in practice as fast as the lock-free one, thus, obtaining wait-freedom without paying a performance cost. We do not presume to improve on the most efficient concurrent data structures available today.

4.1 Implementation details

4.1.1 Auxiliary structures

Similarly to many dynamically-allocated queues, our queue is based on an internal singly-linked list with references maintained to its head and tail, denoted, respectively, as `head` and `tail`. The auxiliary structures for list nodes, as well as for helping records and entries of the `state` array that were described in Section 3, are given in Figure 2. Notice the `enqTid` field added to the `Node` class. As described in Appendix A.2, this field is used in the KP-queue to identify threads that are in the middle of their `enqueue` operations in order to help them to complete these operation. In our case, this field serves a similar purpose and helps to synchronize between `enqueue` operations running on the slow and fast paths, as will be explained in Section 4.1.2. For the `dequeue` operation, we use a different mechanism (described below in this section) and thus do not require an additional field in the `Node` class.

The `OpDesc` class serves to hold descriptors of operations executed by threads on the slow path. Each thread maintains one instance of this class in the array called `state`. `OpDesc` contains the phase number (which counts the number of thread's operations on the slow path), the flags specifying whether the thread has a pending operation on the slow path and if so, its type, and a reference to a list node; this reference has a meaning specific to the type of operation. In addition, each thread maintains one helping record, which is an instance of the `HelpRecord` class. The structure of `HelpRecord` follows the design presented in Section 3. In particular, it contains fields for the ID of a thread that might need help, the phase number recorded for that thread, and a counter used to determine when the next check for helping should be performed.

The internal fields of the queue are given in Figure 3, and the queue constructor is in Figure 4. While the `tail` reference maintained in our queue is implemented as an `AtomicReference` object (like in the MS-queue [13]), the `head` reference is implemented as an `AtomicStampedReference` object. The latter encapsulates a reference to an object along with an integer stamp. Both reference and stamp fields can be updated atomically, either together or individually (for more information, including how this object can be implemented in C/C++ languages, refer to [13], pp. 235–236). In our case, the stamp has a role similar to the `enqTid` field in `Node`, i.e., to identify the thread that is in the middle of its `dequeue` operation on the slow path. More details are provided in Section 4.1.2.

```
1  class Node {
2    int value;
3    AtomRef<Node> next;
4    int enqTid;

5    Node (int val) {
6      value = val;
7      next = new
         AtomRef<Node>();
8      enqTid = −1;
9    }
10 }
```

```
11 class OpDesc {
12   long phase;
13   boolean pending;
14   boolean enqueue;
15   Node node;

16   OpDesc (long ph, boolean pend,
       boolean enq, Node n) {
17     phase = ph; pending = pend;
18     enqueue = enq; node = n;
19   }
20 }
```

```
21 class HelpRecord {
22   int curTid;
23   long lastPhase;
24   long nextCheck;

25   HelpRecord() { curTid = −1; reset(); }

26   void reset() {
27     curTid = (curTid + 1) %
         NUM_THRDS;
28     lastPhase = state.get(curTid).phase;
29     nextCheck = HELPING_DELAY;
30   }
31 }
```

Figure 2. Auxiliary structures.

```
32 AtomStampedRef<Node> head;
33 AtomRef<Node> tail;

34 AtomRefArray<OpDesc> state;
35 HelpRecord helpRecords[];
```

Figure 3. Queue class fields.

```
36 FastWFQueue () {
37   Node sentinel = new Node(−1);
38   head = new AtomicStampedReference<Node>(sentinel, −1);
39   tail = new AtomicReference<Node>(sentinel);

40   state = new AtomicReferenceArray<OpDesc>(NUM_THRDS);
41   helpRecords = new HelpRecord[NUM_THRDS];

42   for (int i = 0; i < state.length(); i++) {
43     state.set(i, new OpDesc(−1, false, true, null));
44     helpRecords[i] = new HelpRecord();
45   }
46 }
```

Figure 4. Queue constructor.

```
47 void help_if_needed() {
48   HelpRecord rec = helpRecords[TID];
49   if (rec.nextCheck−− == 0) {
50     OpDesc desc = state.get(rec.curTid);
51     if (desc.pending && desc.phase == rec.lastPhase) {
52       if (desc.enqueue) help_enq(rec.curTid, rec.lastPhase);
53       else help_deq(rec.curTid, rec.lastPhase);
54     }
55     rec.reset();
56   }
57 }
```

Figure 5. The method called by threads at the entrance to the fast path. It helps an operation on the slow path if this help is necessary.

4.1.2 Fast path

Do I need to help? Before attempting to apply an operation on the fast path, a thread t_i invokes the `help_if_needed()` method, where it checks whether some other thread needs its help (Figure 5). The thread t_i executing this method reads its helping record (Line 48). According to the value of the `nextCheck` counter, t_i decides whether it has to check the status of the thread whose ID is written in the record (Line 49). If so, it accesses the entry of that thread (denote it as t_j) in `state`, and checks if t_j has a pending operation with a phase number equal to `lastPhase` (from t_i's helping record) (Lines 50–51). In such a case, t_i helps to complete t_j's operation according to the type of operation written in t_j's operation descriptor (Lines 52–53). The methods `help_deq()` and `help_enq()` are part of the slow path and are explained in Section 4.1.3. Finally, t_i resets its helping record (Line 55), recording the current phase number of the next thread (in a cyclic order) and setting the `nextCheck` counter to the `HELPING_DELAY` value (cf. Figure 2).

Enqueue operation The details of the `enqueue` operation executed by threads running on the fast path are provided in Figure 6. A thread t_i starts with a call to the `help_if_needed()` method (Line 59), which was explained above. Next, t_i initializes the `trials` counter and tries to append a new node to the end of the internal linked-list. The number of such trials is limited by the `MAX_FAILURES` parameter, which controls when t_i will give up and switch to the slow path. Each trial is similar to the one performed in the MS-queue, that is, t_i identifies the last node in the list by reading `tail` (Line 63) and tries to append a new node after that last node with a CAS (Line 67). The only difference from the MS-queue is in the way `tail` is fixed when t_i identifies that some `enqueue` operation is in progress (i.e., `tail` references some node whose `next` reference is different from `null`). In the MS-queue, the fix is done by simply updating `tail` to refer to the new last node. In our case, the fix is carried out by the `fixTail()` method, which is aware of the existence of the slow path. In particular, it checks whether the `enqTid` field of the last node in the list is set to a default value -1 (Line 79). If so, it means that the last node was inserted into the list from the fast path, and thus `tail` can simply be updated to refer to it (as done in the MS-queue). Otherwise, the last node was inserted from the slow path. Thus, we need first to update the entry in the `state` array corresponding to the thread whose ID = `enqTid` (clearing its `pending` flag), and only after that can we update `tail`. (Without updating `state`, we may create a race between threads trying to help the same `enqueue` operation on the slow path). This functionality is carried out by `help_finish_enq()`, explained in Section 4.1.3.

Finally, if t_i fails to append a new node within `MAX_FAILURES` trials, it calls the `wf_enq()` method (Line 76), which transfers the execution of the `enqueue` operation to the slow path.

Dequeue operation Figure 7 provides the details of the `dequeue` operation executed by threads running on the fast path. Similarly to `enqueue`, the `dequeue` operation starts with calling `help_if_needed()` (Line 86) and initializing the `trials` counter (Line 87). In the **while** loop (Lines 88–108), a thread tries to update `head` to refer to the next element in the queue (or throw an exception if it finds the queue empty). If the thread fails to complete this in `MAX_FAILURES` trials, it switches to the slow path, calling the `wf_deq()` method (Line 109).

We use a slightly modified version of the **while** loop of the `dequeue` operation in the MS-queue [20]: as in the MS-queue, when a queue is found to be empty (Line 93) but there is an `enqueue` operation in progress (Line 97), the dequeuing thread helps to complete that `enqueue` first. In our case, however, the dequeuing thread needs to be aware of the path on which the `enqueue` operation is progressing. For this purpose, we use the `fixTail()` method described above.

We also modify the original MS-queue implementation with regard to how the removal of the first element of the queue is handled. As in the case of `enqueue`, we need to synchronize between con-

```
58 void enq(int value) {
59   help_if_needed(); // check first if help is needed
60   Node node = new Node(value);
61   int trials = 0; // init the trials counter
62   while (trials++ < MAX_FAILURES) {
63     Node last = tail.get();
64     Node next = last.next.get();
65     if (last == tail.get()) {
66       if (next == null) { // enqueue can be applied
67         if (last.next.compareAndSet(next, node)) {
68           tail.compareAndSet(last, node);
69           return;
70         }
71       } else { // some enqueue operation is in progress
72         fixTail(last, next); // fix tail, then retry
73       }
74     }
75   }
76   wf_enq(node); // switch to the slow path
77 }

78 void fixTail(Node last, Node next) {
79   if (next.enqTid == -1) { // next node was appended on the fast path
80     tail.compareAndSet(last, next);
81   } else { // next node was appended on the slow path
82     help_finish_enq();
83   }
84 }
```

Figure 6. The enqueue operation on the fast path.

```
85 int deq() throws EmptyException {
86   help_if_needed(); // check first if help is needed
87   int trials = 0; // init the trials counter
88   while (trials++ < MAX_FAILURES) {
89     Node first = head.getReference();
90     Node last = tail.get();
91     Node next = first.next.get();
92     if (first == head.getReference()) {
93       if (first == last) { // queue might be empty
94         if (next == null) { // queue is empty
95           throw new EmptyException();
96         }
97         fixTail(last, next); // some enqueue operation is in progress
98       } else if (head.getStamp() == -1) {
99         // no dequeue was linearized on the slow path
100        int value = next.value;
101        if (head.compareAndSet(first, next, -1, -1)) {
102          return value;
103        }
104      } else { // some dequeue was linearized on the slow path
105        help_finish_deq(); // help it to complete, then retry
106      }
107    }
108  }
109  return wf_deq(); // switch to the slow path
110 }
```

Figure 7. The dequeue operation on the fast path.

current dequeue operations running on the fast and slow paths. For this purpose, we use the stamp of the head reference. As explained in Section 4.1.3, the dequeue operation run by a thread t_j on the slow path is linearized when t_j writes the ID of the thread for which the dequeue is performed (i.e., j or the ID of the thread helped by t_j) into the head's stamp. Thus, in the fast path, a thread t_i checks first if the head's stamp holds a default (-1) value (Line 98). If so, then as in the original MS-queue, t_i tries to update head to refer to the next node in the underlying list (Line 101). Otherwise, t_i realizes that the first element of the queue has been removed by a dequeue operation that runs on the slow path. Thus, t_i helps first to complete that dequeue operation (Line 105) by updating the state of the thread whose ID is written in the head's stamp and swinging head to refer to the next node in the linked-list. This is done by the help_finish_deq() method, which is explained in Section 4.1.3.

4.1.3 Slow path

Enqueue operation The enqueue operation run by a thread t_i on the slow path is given in Figure 8. It starts from writing a new operation descriptor into the entry of t_i in the state array. The new descriptor is created with a phase number increased by 1 from the value that appears in the previous descriptor of t_i. Essentially, the phase number counts the number of operations run by a thread on the slow path. Also, t_i updates the enqTid field of the new node it tries to insert into the linked-list, with its ID (i in this example). Afterwards, t_i calls the help_enq() method.

The help_enq() method is executed either by a thread that tries to enqueue a new element on the slow path, or by a thread that starts a fast path and has decided to help another thread (this is the method called from help_if_needed()). Its implementation is similar to that of the KP-queue. A thread tries to append a new node to the end of the linked-list as long as the corresponding enqueue operation remains pending, i.e., as long as it was not completed by another concurrently running and helping thread. When the thread succeeds to append the new node (by swinging the next reference of the last node in the list to refer to the new node), it calls help_finish_enq(), where it clears the pending flag in the entry in state of the thread for which it executed help_enq() and updates tail to refer to the newly appended node.

Dequeue operation Similarly to enqueue, the dequeue operation run by threads on the slow path starts with writing a new operation descriptor into the state array (cf. Figure 9). Afterwards, the help_deq() method is called. This method is also called by threads that decide to help another dequeue operation in help_if_needed().

In help_deq(), assuming the queue is not empty, a thread t_i tries to write its ID (or the ID of the thread that it helps in help_if_needed()) into the stamp in head. Once t_i succeeds in doing so, the dequeue operation of t_i (or the thread helped by t_i) has been linearized. This explains why we actually need head to be stamped. If, for instance, we used a deqTid field in list nodes and linearize an operation on the slow path at a successful update of this field (as done in the KP-queue), a race between dequeue operations run on the fast and slow paths might ensue: While the operation on the fast path is linearized at the instant the head reference is updated, the operation on the slow path would be linearized when the deqTid field of the first node in the queue is modified.

Following the write of thread's ID into the stamp, t_i calls help_finish_deq(), where the pending flag of the linearized dequeue operation is cleared and head is fixed to refer to the next node in the underlying linked-list.

4.2 Correctness

In the short version of our paper, we provide only the highlights of the proof that the given implementation of the concurrent queue is correct and wait-free, deferring the further details to the longer version. In particular, we review the computation model, define linearization points for queue operations, and explain why our algorithm guarantees wait-freedom.

4.2.1 Computation model

Our model of a concurrent multithreaded system follows the linearizability model defined in [14]. In particular, we assume that programs are run by n deterministic threads, which communicate by executing atomic operations on shared variables from some pre-

```
111 void wf_enq(Node node) {
112   long phase = state.get(TID).phase + 1; // increase the phase counter
113   node.enqTid = TID;
114   // announce enqueue
115   state.set(TID, new OpDesc(phase, true, true, node));
116   help_enq(TID, phase); // help your own operation to complete
117   help_finish_enq(); // make sure tail is properly updated[15]
118 }

119 void help_enq(int tid, long phase) {
120   while (isStillPending(tid, phase)) {
121     Node last = tail.get();
122     Node next = last.next.get();
123     if (last == tail.get()) {
124       if (next == null) { // enqueue can be applied
125         // avoid racing with other helping threads [15]
126         if (isStillPending(tid, phase)) {
127           // try to append the new node to the list
128           if (last.next.compareAndSet(next,state.get(tid).node)) {
129             help_finish_enq();
130             return;
131           }
132         }
133       } else { // some enqueue operation is in progress
134         help_finish_enq();     // help it first, then retry
135       }
136     }
137   }
138 }

139 void help_finish_enq() {
140   Node last = tail.get();
141   Node next = last.next.get();
142   if (next != null) {
143     // read the enqTid field of the last node in the list
144     int tid = next.enqTid;
145     if (tid != −1) { // last node was appended on the slow path
146       OpDesc curDesc = state.get(tid);
147       if (last == tail.get() && state.get(tid).node == next) {
148         // switch the pending flag off
149         OpDesc newDesc =
              new OpDesc(state.get(tid).phase, false, true, next);
150         state.compareAndSet(tid, curDesc, newDesc);
151         tail.compareAndSet(last, next); // update tail
152       }
153     } else { // last node was appended on the fast path
154       tail.compareAndSet(last, next); // update tail
155     }
156   }
157 }
```

Figure 8. The enqueue operation on the slow path.

defined, finite set. Threads are run on computing cores, or processors, and the decision which thread will run when and on which processor is made solely by a scheduler. Normally, the number of processors is much smaller than the number of threads. Each thread is assumed to have an ID in a range between 0 and $n-1$. In fact, this assumption can be easily relaxed by means of a wait-free renaming algorithm (e.g., [1]). We also assume that each thread can access its ID and the value of n.

When scheduled to run, a thread performs a sequence of computation steps. Each step is either a local computation or an atomic operation on at most one shared variable. We assume that the shared memory supports atomic reads, writes, and compare-and-swap operations. The latter, abbreviated as CAS, is defined with the following semantics: CAS(v, *old*, *new*) changes the value of the shared variable v to *new* (and returns *true*) if and only if its value just before CAS is applied is equal to *old*. We refer to such CAS operations as *successful*. Otherwise, the value of v is unchanged, *false* is returned, and we refer to such CAS operations as *unsuccessful*. Note that we do not require any special operations to support stamped shared variables. Such variables, required by our implementation, can be implemented with atomic operations men-

```
158 int wf_deq() throws EmptyException {
159   long phase = state.get(TID).phase + 1; // increase the phase counter
160   // announce dequeue
161   state.set(TID, new OpDesc(phase, true, false, null));
162   help_deq(TID, phase); // help your own operation to complete
163   help_finish_deq(); // make sure head is properly updated [15]

164   Node node = state.get(TID).node; // check the node recorded in state
165   if (node == null) { // dequeue was linearized on the empty queue
166     throw new EmptyException();
167   }
168   // return the value of the first non−dummy node [15]
169   return node.next.get().value;
170 }

171 void help_deq(int tid, long phase) {
172   while (isStillPending(tid, phase)) {
173     Node first = head.getReference();
174     Node last = tail.get();
175     Node next = first.next.get();
176     if (first == head.getReference()) {
177       if (first == last) { // queue might be empty
178         if (next == null) { // queue is empty
179           OpDesc curDesc = state.get(tid);
180           if (last == tail.get() && isStillPending(tid, phase)) {
181             // record null in the node field,
182             // indicating that the queue is empty
183             OpDesc newDesc =
                    new OpDesc(state.get(tid).phase, false, false, null);
184             state.compareAndSet(tid, curDesc, newDesc);
185           }
186         } else { // some enqueue operation is in progress}
187           help_finish_enq(); // help it first, then retry
188         }
189       } else {
190         OpDesc curDesc = state.get(tid);
191         Node node = curDesc.node;
192         // avoid racing with other helping threads [15]
193         if (!isStillPending(tid, phase)) break;
194         if (first == head.getReference() && node != first) {
195           OpDesc newDesc =
                  new OpDesc(state.get(tid).phase, true, false, first);
196           // try to record a reference to the first node in the list
197           if (!state.compareAndSet(tid, curDesc, newDesc)) {
198             continue;
199           }
200         }
201         head.compareAndSet(first, first, −1, tid); // try to stamp head
202         help_finish_deq(); // help thread that won the stamp to complete
203       }
204     }
205   }
206 }

207 void help_finish_deq() {
208   Node first = head.getReference();
209   Node next = first.next.get();
210   int tid = head.getStamp(); // read the stamp on head
211   if (tid != −1) { // last dequeue was linearized on the slow path
212     OpDesc curDesc = state.get(tid);
213     if (first == head.getReference() && next != null) {
214       // switch the pending flag off
215       OpDesc newDesc =
            new OpDesc(state.get(tid).phase, false, false, state.get(tid).node);
216       state.compareAndSet(tid, curDesc, newDesc);
217       head.compareAndSet(first, next, tid, −1); // update head
218     }
219   } // last dequeue was linearized on the fast path − nothing to do here
220 }

221 boolean isStillPending(int tid, long ph) {
222   return state.get(tid).pending && state.get(tid).phase <= ph;
223 }
```

Figure 9. The dequeue operation on the slow path.

tioned above either by introducing a level of indirection (this is how `AtomicStampedReference` in Java is implemented) or by stealing a few bits from the value of the variable [13].

A concurrent queue is a data structure with operations linearizable [14] to those of a sequential queue. The latter supports two operations: `enqueue` and `dequeue`. The first operation accepts an element as an argument and inserts it into the queue. The second operation does not accept any argument, and removes and returns the oldest element from the queue. If the queue is empty, the `dequeue` operation returns a special value (or throws an exception).

4.2.2 Linearizability

The operations of the concurrent queue presented above are composed of the fast and slow paths. Thus, each operation has linearization points on each of the two paths, where the points on the slow path can be reached only if the points on the fast path are not reached. Given an operation executed by a thread t_i, note that the source lines corresponding to the linearization points on the fast path can be executed only by t_i, while the lines corresponding to the linearization points on the slow path can be executed either by t_i or by any other thread t_j that tries to help t_i by running `help_enq()` or `help_deq()` from `help_if_needed()`. In the definition below, we refer to a `dequeue` operation that returns a value as *successful*, while a `dequeue` that ends by throwing an exception is referred to as *unsuccessful*. An `enqueue` operation is always successful.

Definition 1. *The linearization points for operations applied on the fast path are as follows:*

- *An `enqueue` operation is linearized at the successful CAS in Line 67.*
- *A successful `dequeue` operation is linearized at the successful CAS in Line 101.*
- *An unsuccessful `dequeue` operation is linearized in Line 90.*

The linearization points for operations applied on the slow path are as follows:

- *An `enqueue` operation is linearized at the successful CAS in Line 128.*
- *A successful `dequeue` operation is linearized at the successful CAS in Line 201.*
- *An unsuccessful `dequeue` operation is linearized in Line 174.*

Note that the linearization points on the fast path correspond to the linearization points of the MS-queue [20]. Similarly, the linearization points for operations applied on the slow path correspond to the points of the KP-queue [15]. The proof of correctness of the linearization points defined above is far beyond the scope of this short paper. Our full proof is composed of two parts. First, we show that nodes corresponding to queue elements are inserted and removed to/from the queue according to the FIFO semantics. In the second part of the proof, we show that each operation executed on the slow path is linearized exactly once.

4.2.3 Wait-freedom

We provide an overview of the proof of the wait-free progress guarantee. This proof also has two parts. First, we show that the algorithm is lock-free. For this purpose, we prove that every time a thread fails to linearize an operation, either by applying an unsuccessful CAS or by failing to pass one of the verification conditions in the **if**-statements (e.g., in Lines 65, 66, 92, etc.), some other thread makes progress and does succeed in linearizing an operation. The proof is based on a straightforward inspection of code lines that modify the underlying linked-list, in a way similar to the lock-freedom proof in [20].

The second part of our proof shows that the number of steps taken by a thread before its pending operation is linearized, is limited. The following lemma is at the heart of this part. For brevity, we refer to the `HELPING_DELAY` and `MAX_FAILURES` constants by D and F, respectively.

Lemma 1. *The number of steps required for a thread to complete an operation on the queue is bounded by $O(F + D \cdot n^2)$.*

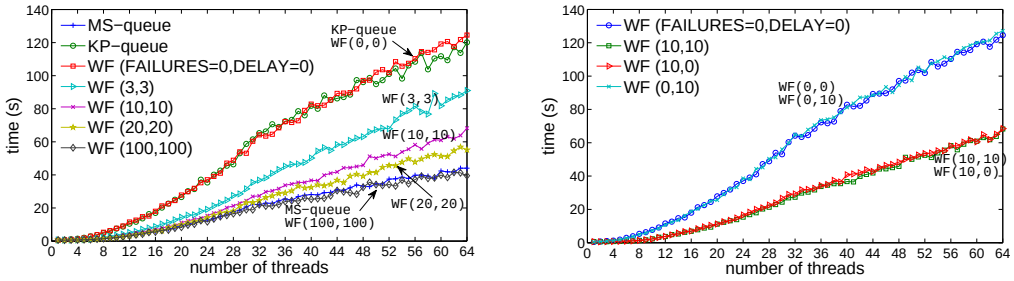
Sketch of proof: Consider a pending operation run by a thread t_i on the slow path. In the worst case, this operation will remain pending until all other threads in the system decide to help it to complete (in Line 52 or Line 53). A thread t_j will decide to help t_i after it completes at most $O(D \cdot n)$ operations. This is because t_j might complete up to D operations before it decides to help a thread whose ID is written in t_j's helping record, and t_j might help all other $n-1$ threads before it gets to t_i. However, once t_j starts helping t_i, it will not stop helping it until t_i actually makes progress. Thus, the number of operations that might linearize before *all* other threads decide to help t_i is $O(D \cdot n^2)$. Furthermore, since the algorithm is lock-free, it is guaranteed that once all threads are helping t_i, the operation of t_i will be completed in a constant number of steps.

Now consider the thread t_i when it starts its operation on the queue on the fast path. It may realize in `help_if_needed()` that it needs to help another thread t_k before attempting to execute its own operation. Following similar arguments, the number of operations that might linearize before t_i returns from `help_if_needed()` is bounded by $O(D \cdot n^2)$. Afterwards, t_i makes several attempts to linearize its operation on the fast path. Given that the algorithm is lock-free, it might fail if during its attempts other operations succeed to linearize. As the number of such attempts is limited by `MAX_FAILURES`, after at most $O(F)$ steps t_i will switch into the slow path, and by the above argument will complete its operation in at most $O(D \cdot n^2)$ steps. Thus, after at most $O(F + D \cdot n^2)$ steps in total, t_i will complete its operation. □

4.3 Performance

In our performance study, we compare our fast wait-free queue with the lock-free MS-queue and the wait-free KP-queue. For the MS-queue, we use the Java implementation given in [13]. The KP-queue has several versions presented in [15]; for our study, we use the optimized one. We employ several versions of the fast wait-free queue, each configured with different `MAX_FAILURES` and `HELPING_DELAY` parameters. We denote by WF(x,y) the version with `MAX_FAILURES` set to x and `HELPING_DELAY` set to y.

The study was carried out on a machine featuring a shared-memory NUMA server with 8 quadcore AMD 2.3GHz processors (32 physical cores in total), operating under Ubuntu 10.10 and installed with 16GB RAM attached to each processor. All tests were run in OpenJDK Runtime version 1.6.0 update 20, using the 64-Bit Server VM, with -Xmx16G -Xms16G flags. In our tests, we varied the number of threads between 1 and 64. Starting with an empty queue, we performed two common queue benchmarks [15, 16, 20]. First, we ran the `enqueue-dequeue` benchmark, where each thread performs numerous (100000, in our case) iterations of an `enqueue` operation followed by a `dequeue` operation. Second, we ran the 50%-`enqueue` benchmark, where on each iteration every thread randomly chooses which operation to perform with equal chances to `enqueue` and `dequeue` (with 100000 iterations per thread). Note that the queue in both benchmarks remains short or even empty, maximizing the contention between threads performing `enqueue` and `dequeue`. To mimic local computations performed by threads after accessing the queue, we inserted a small and random delay after each operation on the queue [16, 20]. The delay was achieved by running some simple calculation in a loop with a randomly chosen number of iterations [16]. We note that the results in tests without local computations were qualitatively the same.



(a) (b)

Figure 10. Total completion time for various queues and the `enqueue-dequeue` benchmark.

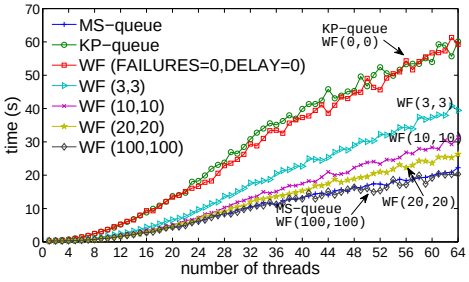


Figure 11. Total completion time for various queues and the 50%-`enqueue` benchmark.

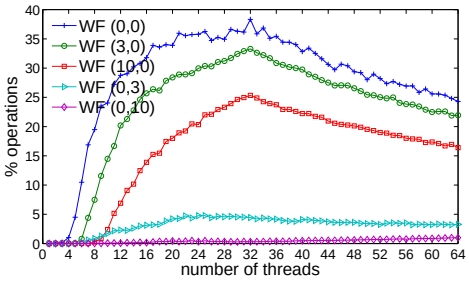


Figure 12. Percentage of operations that help other operations (`enqueue-dequeue` benchmark).

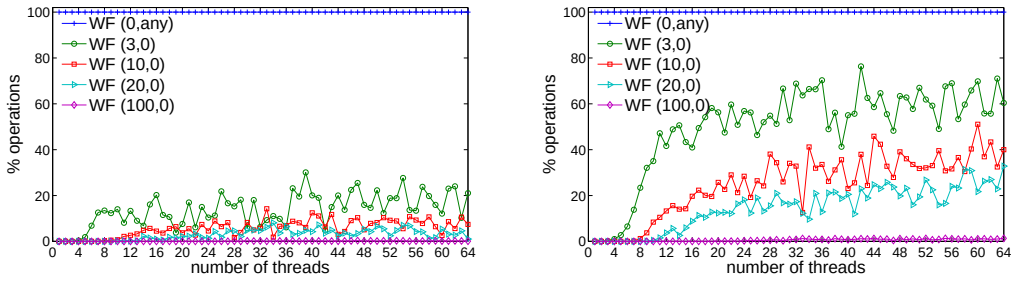
We report the total completion time for each of the tested queues, as well as some interesting statistics on the number of operations that actually use the slow path and those that actually decide to help. Each reported point is an average of ten experiments run with the same parameters. We note that the standard deviation (not shown for the sake of readability) for the total completion time figures was up to 15%, with the majority of results having deviation around 8% and below. (In our tests on other machines with fewer cores, the standard deviation was smaller than that). Other results, concerning the statistics of the executions, were much noisier, as they depended on the actual execution interleaving created by the system scheduler. Nonetheless, every reported performance phenomenon was reproduced in several repeated runs of our benchmarks.

Figure 10 summarizes the total completion time numbers for various queue implementations and the `enqueue-dequeue` benchmark. In particular, Figure 10a shows that the WF(0,0) version exhibits performance very close to that of the KP-queue, while both queues perform roughly 3 times worse than the MS-queue. This happens since all operations in WF(0,0) use only the slow path, which is far less efficient then the MS-queue. Increasing the `MAX_FAILURES` and `HELPING_DELAY` parameters reduces the number of operations that require the slow path. In fact, when both parameters are set to 100, the fast wait-free queue steadily achieves performance similar to that of the MS-queue. Consequently, these results demonstrate that our wait-free design can achieve the performance and scalability of a lock-free algorithm.

Our other experiments (as shown in Figure 10b) suggest that giving threads the chance to complete their operation on the fast path (i.e., increasing the number of trials) is highly effective (e.g., the performance of WF(10,0) is similar to WF(10,10)). On the other hand, reducing their helping work (by increasing the delay in making an attempt to help other threads) does not change the performance significantly (e.g., the performance of WF(0,10) is similar to WF(0,0)). Note that even when a thread t_i provides help on each operation (i.e., when `HELPING_DELAY`=0), it only helps another thread t_j that did not make progress since t_i's previous operation. As shown below, this small delay already ensures that the amount of work put into the helping effort is limited.

Figure 11 shows the performance results for the 50%-`enqueue` benchmark. In general, all queue implementations exhibit similar behavior as in the `enqueue-dequeue` benchmarks, but the total completion time numbers are 2 time smaller. This is because in the 50%-`enqueue` benchmark, each thread performs half of the operations of the first benchmark. We note that all measured queues produced similar relative behavior in both benchmarks. Thus, due to lack of space, we will focus only on the `enqueue-dequeue` benchmark for the rest of this section.

Figure 12 shows the percentage of queue operations that decide to help other operations in the `help_if_needed()` method. Here, increasing the `MAX_FAILURES` parameter decreases the number of times threads help other operations to complete. As we will see below, this is a side-effect of the fact that, with a higher `MAX_FAILURES` parameter, fewer operations enter the slow path and request help. At the same time, increasing the `HELPING_DELAY` parameter has a much greater effect even though all operations having `MAX_FAILURES` set to 0 enter the slow path. This shows that the delayed helping can potentially reduce the amount of redundant work created by threads helping each other. It is interesting also to note that the percentage of helping operations for all queues increases until the number of threads reaches the number of available cores (32) and then slightly, but continuously decreases. The first phenomenon is explained by increased contention, i.e., more operations are repeatedly delayed due to contention and thus helped by others. When the number of threads goes beyond the number



(a) `dequeue` (b) `enqueue`

Figure 13. Percentage of `enqueue` (a) and `dequeue` (b) operations completing on the slow path (`enqueue-dequeue` benchmark).

of cores, at every moment of execution some threads are switched off. Attempts to help them either return immediately because these threads have no active pending operations, or require one help operation. Once this help is done, no further help operations are executed until the thread regains execution. Thus, a smaller percentage of the total number of operations actually help other operations.

Finally, Figures 13a and 13b show the percentage of `dequeue` and `enqueue` operations that enter the slow path, respectively. For the former, 3 trials are almost always sufficient to ensure that only 20–30% of the operations complete on the slow path. At the same time, for the latter, in order to avoid many operations from entering the slow path, a higher value of `MAX_FAILURES` is required. This might be because `dequeue` is more efficient, requiring just one CAS. Thus, the chances are greater that a `dequeue` operation running on the fast path will complete with just a few trials. Note that for both operations, setting `MAX_FAILURES` to 100 almost eliminates the use of the slow path. Essentially, this number depends on the maximal contention level in the system. (In our tests on other machines with fewer cores, not reported here due to space limits, this number was significantly lower).

Given these results, one may ask why not just always use large values for both parameters so that the slow path will not be used? The proof sketch of the wait-freedom property in Section 4.2 shows that these parameters govern the bound on the worst-case time required to complete each operation on the queue. Thus, they control the tradeoff between the practical performance and the theoretical bound on the worst-case completion time of each operation.

4.4 Memory management

The implementation of our queue is provided in Java, a garbage-collected language, which simplifies the memory management and avoids problems related to it, such as the ABA problem [13]. We have also implemented our queue in C using the Hazard Pointers technique [19] for memory management. Although only minor modifications to the algorithm are required, their precise details are beyond the scope of this short paper.

5. Summary

We presented the fast-path-slow-path methodology for creating fast and scalable wait-free data structures. The key feature in this methodology is designing each operation as a combination of the fast path and the slow path. Good performance is achieved when the fast path is extensively utilized and due to the fact that concurrent operations can proceed on both paths in parallel. Our measurements show that most operations can complete without facing too many failed attempts, even in a highly concurrent environment. Thus, operations almost always complete using the fast path only, and the execution is fast. The application of our methodology is demonstrated in this paper by constructing a new wait-free queue. The performance evaluation of the obtained queue shows that it can be as fast as the efficient lock-free queue of Michael and Scott [20]. Subsequent work [23] utilizes our methodology to improve the performance of a wait-free linked-list.

Acknowledgments

We thank Idit Keidar and Dmitri Perelman for letting us use their multi-core machine for the experiments reported above. Also, we thank the anonymous reviewers for their valuable comments.

References

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

[2] J. H. Anderson and Y.-J. Kim. A new fast-path mechanism for mutual exclusion. *Distrib. Comput.*, 14:17–29, 2001.

[3] 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.

[4] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel. Non-blocking binary search trees. In *Proc. ACM PODC*, pages 131–140, 2010.

[5] P. Fatourou and N. D. Kallimanis. A highly-efficient wait-free universal construction. In *Proc. ACM Symposium on Parallel Algorithms (SPAA)*, pages 325–334, 2011.

[6] F. E. Fich, V. Luchangco, M. Moir, and N. Shavit. Obstruction-free algorithms can be practically wait-free. In *Proc. Int. Conference on Distributed Computing (DISC)*, pages 78–92, 2005.

[7] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proc. Int. Conference on Distributed Computing (DISC)*, pages 300–314, 2001.

[8] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proc. ACM Symposium on Parallel Algorithms (SPAA)*, pages 355–364, 2010.

[9] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. *J. Parallel Distrib. Comput.*, 70(1):1–12, 2010.

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

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

[12] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proc. Conf. on Distributed Computing Systems (ICDCS)*, pages 522–529, 2003.

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

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

[15] A. Kogan and E. Petrank. Wait-free queues with multiple enqueuers and dequeuers. In *Proc. ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 223–234, 2011.

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

[17] L. Lamport. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 5:1–11, 1987.

[18] Y. Levanoni and E. Petrank. An on-the-fly reference-counting garbage collector for java. *ACM TOPLAS*, 28(1):1–69, 2006.

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

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

[21] M. Moir. Transparent support for wait-free transactions. In *Proc. Conf. on Distributed Computing (DISC)*, 1998.

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

[23] S. Timnat, A. Braginsky, A. Kogan, and E. Petrank. Poster paper: Wait-free linked-lists. To appear in *Proc. ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2012.

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

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

[26] J.-H. Yang and J. H. Anderson. A fast, scalable mutual exclusion algorithm. *Distributed Computing*, 9:1–9, 1994.

[27] P.-C. Yew, N.-F. Tzeng, and D. H. Lawrie. Distributing hot-spot addressing in large-scale multiprocessors. *IEEE Trans. Comput.*, 36:388–395, 1987.

A. Building blocks

In our work we show a design methodology, which, when applied on a lock-free and a wait-free version of a data structure, can derive a new wait-free algorithm that will be as fast and scalable as the given lock-free algorithm. In this paper, the application of this methodology is exemplified by a construction of a new and fast wait-free queue algorithm. In order to provide the required background for understanding the design of this construction, we briefly review the principle ideas behind the lock-free queue by Michael and Scott [20] and the recent wait-free queue by Kogan and Petrank [15]. These two queues are referred to in the paper as the MS-queue and the KP-queue respectively, and serve as a basis for our new wait-free queue construction.

A.1 The lock-free MS-queue

The MS-queue [20] utilizes an internal singly-linked list of nodes, which keeps elements of the queue. The algorithm maintains two references to the head and tail of the list with corresponding names: `head` and `tail`. In order to enqueue a new element, a thread creates a new node with this element and verifies that `tail` indeed refers the last node of the list (i.e., a node with the `next` reference set to null). If not, the thread fixes `tail` first. Otherwise, the thread tries to swing the `next` reference of that last node to refer to the new node (using a CAS operation). If it succeeds, the thread fixes `tail` to refer to the new node (using another CAS). Otherwise, it realizes that some concurrently running thread has changed the list, and restarts the operation.

The `dequeue` operation is even simpler. A thread reads the value of the first node in the list and tries to swing `head` to refer to the second node in the list (using CAS). If it succeeds, the thread returns the value it has read; otherwise, it restarts the operation. Special care is given to the case of an empty queue: If a thread finds both `head` and `tail` referencing the same node and there is no `enqueue` in progress (i.e., the `next` reference of that node is `null`), an exception is thrown (or some other action is taken, such as returning a special $\perp$ value). If `head` and `tail` reference the same node, but the `next` reference of that node is not `null`, the dequeuing thread realizes that there is some `enqueue` in progress. Thus, it first helps this `enqueue` to complete (by fixing `tail`) and then reattempts its `dequeue` operation.

A.2 The wait-free KP-queue

The KP-queue [15] extends the ideas of the MS-queue in two main directions. First, to guarantee wait-free progress, the KP-queue utilizes a dynamic priority-based helping mechanism where younger threads (having lower logical priority) help older threads to apply their operations. The mechanism is implemented by means of (1) an auxiliary `state` array, in which each thread writes details on its current operation, and (2) phase numbers chosen by threads in monotonically increasing order before they try to apply their operations on the queue. Thus, when a thread t_i wants to apply an operation on the queue, it chooses a phase number that is higher than phases of threads that have previously chosen phase numbers for their operations. Then t_i helps all threads with the phase number smaller than or equal to t_i's phase (including itself) to apply their pending operations. It learns about pending operations from the `state` array.

Second, to ensure correctness, and in particular, to avoid applying the same operation more than once, the authors propose a three-step scheme used to design each of the two queue operations, `enqueue` and `dequeue`. This scheme requires two new fields, `enqTid` and `deqTid`, to be added to each node in the underlying linked-list. These fields hold the ID of a thread that tries to insert (or remove) a node to (from, respectively) the list. They are used by other concurrently running threads to identify the thread that is in the middle of an operation on the queue, and help it to complete its operation in progress.

In more detail, in order to enqueue a new element, a thread t_i creates a new node with the `enqTid` field set to i. Then it writes a reference to this node along with the chosen phase number into the `state` array. Afterwards, t_i proceeds by helping all threads with a phase smaller than or equal to t_i's phase. When it reaches its own operation, and provided its own operation was not completed yet by another concurrently running and helping thread, t_i tries to put its new element at the end of the underlying list (using CAS). If it succeeds, t_i marks its operation as linearized (i.e., clears the `pending` flag) in `state` (using another CAS) and attempts to fix the `tail` reference (using a third CAS). If t_i finds a node X behind the node referred by `tail`, it reads the `enqTid` field stored in X, marks the operation of the corresponding thread as having been linearized, and only then fixes the `tail` reference and tries again to insert its own node.

In order to dequeue an element from the queue, a thread t_j passes through the following four stages (after writing a new phase in `state` and helping other threads, if necessary, and provided the queue is not empty): (1) stores a reference to the first node of the list in its entry in `state`; (2) tries to write its ID, j, into the `deqTid` field of the first node in the list; (3) if successful, marks its operation as linearized in its entry in `state`, and (4) swings the `head` reference to the next element in the list. Each of the stages corresponds to a CAS operation. Similarly to `enqueue`, if t_j fails in Stage (2), it reads the `deqTid` field stored in the first node of the list, marks the operation of the corresponding thread as linearized, fixes the `head` reference and tries again to remove another node from the list.