An Optimistic Approach to Lock-Free FIFO Queues

Edya Ladan-Mozes[1] and Nir Shavit[2]

[1] Department of Computer Science, Tel-Aviv University, Israel
[2] Sun Microsystems Laboratories and Tel-Aviv University

Abstract. First-in-first-out (FIFO) queues are among the most fundamental and highly studied concurrent data structures. The most effective and practical dynamic-memory concurrent queue implementation in the literature is the lock-free FIFO queue algorithm of Michael and Scott, included in the standard *Java^{TM} Concurrency Package*.
This paper presents a new dynamic-memory lock-free FIFO queue algorithm that performs consistently better than the Michael and Scott queue. The key idea behind our new algorithm is a novel way of replacing the singly-linked list of Michael and Scott, whose pointers are inserted using a costly compare-and-swap (CAS) operation, by an "optimistic" doubly-linked list whose pointers are updated using a simple store, yet can be "fixed" if a bad ordering of events causes them to be inconsistent. We believe it is the first example of such an "optimistic" approach being applied to a real world data structure.

1 Introduction

First-in-first-out (FIFO) queues are among the most fundamental and highly studied concurrent data structures [1–12], and are an essential building block of concurrent data structure libraries such as JSR-166, the Java^{TM} Concurrency Package [13]. A concurrent queue is a linearizable structure that supports enqueue and dequeue operations with the usual FIFO semantics. This paper focuses on queues with dynamic memory allocation.

The most effective and practical dynamic-memory concurrent FIFO queue implementation is the lock-free FIFO queue algorithm of Michael and Scott [14] (Henceforth the *MS-queue*). On shared-memory multiprocessors, this compare-and-swap (CAS) based algorithm is superior to all former dynamic-memory queue implementations including lock-based queues [14], and has been included as part of the Java^{TM} Concurrency Package [13]. Its key feature is that it allows uninterrupted parallel access to the head and tail of the queue.

This paper presents a new dynamic-memory lock-free FIFO queue algorithm that performs consistently better than the MS-queue. It is a practical example of an "optimistic" approach to reduction of synchronization overhead in concurrent data structures. At the core of this approach is the ability to use simple stores instead of CAS operations in common executions, and *fix* the data structure in the uncommon cases when bad executions cause structural inconsistencies.

1.1 The New Queue Algorithm

As with many finely tuned high performance algorithms (see for example CLH [15, 16] vs. MCS [17] locks), the key to our new algorithm's improved performance is in saving a few costly operations along the algorithm's main execution paths.

Figure 1 describes the MS-queue algorithm which is based on concurrent manipulation of a singly-linked list. Its main source of inefficiency is that while its `dequeue` operation requires a single successful CAS in order to complete, the `enqueue` operation requires *two* such successful CASs. This may not seem important, until one realizes that it increases the chances of failed CAS operations, and that on modern multiprocessors [18, 19], even the successful CAS operations cost an order-of-magnitude longer to complete than a load or a store, since they require exclusive ownership and flushing of the processor's write buffers.

The key idea in our new algorithm is to (literally) approach things from a different direction... by logically reversing the direction of `enqueue`s and `dequeue`s to/from the list. If `enqueue`s were to add elements at the beginning of the list, they would require only a single CAS, since one could first direct the new node's `next` pointer to the node at the beginning of the list using only a store operation, and then CAS the `tail` pointer to the new node to complete the insertion. However, this re-direction would leave us with a problem at the end of the list: dequeues would not be able to traverse the list "backwards" to perform a linked-list removal.

Our solution, depicted in Figure 2, is to maintain a doubly-linked list, but to construct the "backwards" direction, the path of `prev` pointers needed by `dequeues`, in an optimistic fashion using only stores (and no memory barriers). This doubly-linked list may seem counter-intuitive given the extensive and complex work of maintaining the doubly-linked lists of lock-free deque algorithms using double-compare-and-swap operations [20]. However, we are able to store and follow the optimistic `prev` pointers in a highly efficient manner.

If a `prev` pointer is found to be inconsistent, we run a `fixList` method along the chain of `next` pointers which is guaranteed to be consistent. Since prev pointers become inconsistent as a result of long delays, not as a result of contention, the frequency of calls to `fixList` is low. The result is a FIFO queue

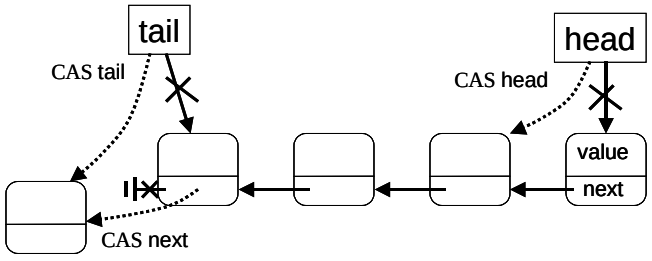

Fig. 1. The single CAS `dequeue` and costly two CAS `enqueue` of the MS-Queue algorithm

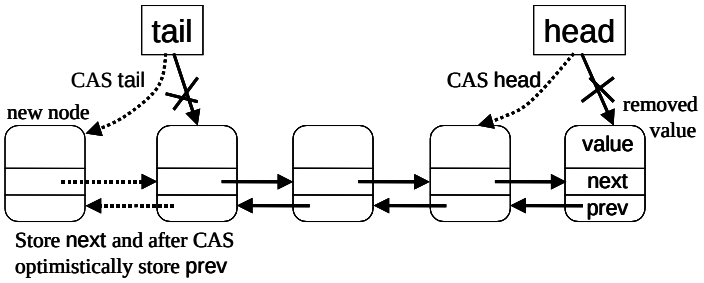

Fig. 2. The Single CAS `enqueue` and `dequeue` of the new algorithm

based on a doubly-linked list where pointers in both directions are set using simple stores, and both `enqueue`s and `dequeue`s require only a single successful CAS operation to complete.

1.2 Optimistic Synchronization

Optimistically replacing CAS with loads/stores was first suggested by Moir et al. [21] who show how one can replace the use of CAS with simple loads in good executions, using CAS only if a bad execution is incurred. However, while they show a general theoretical transformation, we show a practical example of a highly concurrent data structure whose actual performance is enhanced by using the optimistic approach.

Our optimistic approach joins several recent algorithms tailored to the good executions while dealing with the bad ones in a more costly fashion. Among these is the obstruction-freedom methodology of Herlihy et al. [22] and the lock-elision approach by Rajwar and Goodman [23] that use backoff and locking (respectively) to deal with bad cases resulting from contention. Our approach is different in that inconsistencies occur because of long delays, not as a result of contention. We use a special mechanism to fix these inconsistencies, and our resulting algorithm is lock-free.

1.3 Performance

We compared our new lock-free queue algorithm to the most efficient lock-based and lock-free dynamic memory queue implementations in the literature, the two-lock-queue and lock-free MS-queue of Michael and Scott [14]. Our empirical results, presented in Section 4, show that the new algorithm performs consistently better than the MS-queue. This improved performance is not surprising, as our enqueues require fewer costly CAS operations, and as our benchmarks show, generate significantly less failed CAS operations.

The new algorithm can use the same dynamic memory pool structure as the MS-queue. It fits with memory recycling methods such as ROP [24] or SMR

[25], and it can be written in the Java^{TM} programming language without the need for a memory pool or ABA-tags. We thus believe it can serve as a viable practical alternative to the MS-queue.

2 The Algorithm in Detail

The efficiency of our new algorithm rests on implementing a queue using a doubly-linked list, which, as we show, allows `enqueue`s and `dequeue`s to be performed with a single CAS per operation. Our algorithm guarantees that this list is always connected and ordered by the `enqueue` order in one direction. The other direction is optimistic and may be inaccurate at various points of the execution, but can be reconstructed to an accurate state when needed.

Our shared queue data structure (see Figure 3) consists of a `head` pointer, a `tail` pointer, and nodes. Each node added to the queue contains a `value`, a `next` pointer and a `prev` pointer. Initially, a node with a predefined dummy value, hence forth called a `dummy` node, is created and both `head` and `tail` point to it. During the execution, the `tail` always points to the last (youngest) node inserted to the queue, and the `head` points to the first (oldest) node. When the queue becomes empty, both `head` and `tail` point to a `dummy` node. Since our algorithm uses CAS for synchronization, the ABA issue arises [14, 10]. In Section 2.1, we describe the `enqueue` and `dequeue` operations ignoring ABA issues. The tagging mechanism we added to overcome the ABA problem is explained in Section 3. The code in this section includes this tagging mechanism. Initially, the tags of the `tail` and `head` are zero. When a new node is created, the tags of the `next` and `prev` pointers are initiated to a predefined null value.

```
struct pointer_t {
  <ptr, tag>: <node_t *, unsigned integer>
};

struct node_t {
  data_type value;
  pointer_t next;
  pointer_t prev;
};

struct queue_t {
  pointer_t tail;
  pointer_t head;
};
```

Fig. 3. The queue data structures

2.1 The Queue Operations

A FIFO queue supports two operations (or methods): `enqueue` and `dequeue`. The `enqueue` operation inserts a value to the queue and the `dequeue` operation deletes the oldest value in the queue.

The code of the `enqueue` method appears in Figure 4, and the code of the `dequeue` method appears in Figure 5. To insert a value, the `enqueue` method creates a new node that contains the value, and then tries to insert this node to the queue. As seen in Figure 2, the `enqueue` reads the current `tail` of the queue, and sets the new node's `next` pointer to point to that same node. Then it tries to atomically modify the `tail` to point to its new node using a CAS operation. If the CAS succeeded, the new node was inserted into the queue. Otherwise the enqueue retries.

```
void enqueue(queue_t* q, data_type val)
E01: pointer_t tail
E02: node_t* nd = new_node()                          # Allocate a new node
E03: nd->value = val                                  # Set enqueued value
E04: while(TRUE){                                     # Do till success
E05:   tail = q->tail                                 # Read the tail
E06:   nd->next = <tail.ptr, tail.tag+1>              # Set node's next ptr
E07:   if CAS(&(q->tail), tail, <nd, tail.tag+1>){ # Try to CAS the tail
E08:     (tail.ptr)->prev = <nd, tail.tag>            # Success, write prev
E09:     break                                        # Enqueue done!
E10:   }
E11: }
```

Fig. 4. The `enqueue` operation

To delete a node, a `dequeue` method reads the current `head` and `tail` of the queue, and the `prev` pointer of the node pointed by the `head`. It then tries to CAS the `head` to point to the node as that pointed by the `prev` pointer. If it succeeded, then the node previously pointed by the `head` was deleted. If it failed, it repeats the above steps. If the queue is empty then `NULL` is returned.

We now explain how we update the `prev` pointers of the nodes in a consistent and lock-free manner, assuming there is no ABA problem. The `prev` pointers are modified in two stages. The first stage is performed optimistically immediately after the successful insertion of a new node. An `enqueue` method that succeeded in atomically modifying the `tail` using a CAS, updates the `prev` pointer of the node previously pointed by the `tail` to point to the new node. This is done using a simple store operation. Once this write is completed, the `prev` pointer points to its preceding node in the list. Thus the order of operations to perform an enqueue is a write of the `next` in the new node, then a CAS of the `tail`, and finally a write of the `prev` pointer of the node pointed to by the `next` pointer. This ordering will prove crucial in guaranteeing the correctness of our algorithm.

Unfortunately, the storing of the `prev` pointer by an `enqueue` might be delayed for various reasons, and a dequeuing method might not see the necessary `prev` pointer. The second stage is intended to fix this situation. In order to fix the `prev` pointer, we use the fact that the `next` pointer of each node is set only by the `enqueue` method that inserted that node, and never changes until the node

```
data_type dequeue(queue_t* q)
D01: pointer_t tail, head, firstNodePrev
D02: node_t* nd_dummy
D03: data_type val
D04: while(TRUE){                          # Try till success or empty
D05:   head = q->head                              # Read the head
D06:   tail = q->tail                              # Read the tail
D07:   firstNodePrev = (head.ptr)->prev            # Read first node prev
D08:   val = (head.ptr)->value                     # Read first node val
D09:   if (head == q->head){                       # Check consistency
D10:     if (val != dummy_val){                    # Head val is dummy?
D11:       if (tail != head){                      # More than 1 node?
D12:         if (firstNodePrev.tag != head.tag){  # Tags not equal?
D13:           fixList(q, tail, head)              # Call fixList
D14:           continue                            # Re-iterate (D04)
D15:         }
D16:       }
D17:       else{                                   # Last node in queue
D18:         nd_dummy = new_node()                 # Create a new node
D19:         nd_dummy->value = dummy_val           # Set it's val to dummy
D20:         nd_dummy->next = <tail.ptr, tail.tag+1>   # Set its next ptr
D21:         if CAS(&(q->tail), tail ,<nd_dummy, tail.tag+1>){# CAS tail
D22:           (head.ptr).prev = <nd_dummy, tail.tag>         # Write prev
D23:         }
D24:         else{                                 # CAS failed
D25:           free(nd_dummy)                      # free nd_dummy
D26:         }
D27:         continue;                             # Re-iterate (D04)
D28:       }
D29:       if CAS(&(q->head), head, <firstNodePrev.ptr,head.tag+1>){# CAS
D30:         free (head.ptr)                       # Free the dequeued node
D31:         return val                            # Dequeue done!
D32:       }
D33:     }
D34:     else {                                    # Head points to dummy
D35:       if (tail.ptr == head.ptr){              # Tail points to dummy?
D36:         return NULL;                          # Empty queue, done!
D37:       }
D38:       else{                                   # Need to skip dummy
D39:         if (firstNodePrev.tag != head.tag){ # Tags not equal?
D40:           fixList(q, tail, head);             # Call fixList
D41:           continue;                           # Re-iterate (D04)
D42:         }
D43:         CAS(&(q->head),head,<firstNodePrev.ptr,head.tag+1>)#Skip dummy
D44:       }
D45:     }
D46:   }
D47: }
```

Fig. 5. The `dequeue` operation

is dequeued. Thus, if ABA problems resulting from node recycling are ignored, this order is invariant. The fixing mechanism walks through the entire list from the `tail` to the `head` along the chain of `next` pointers, and corrects the `prev` pointers accordingly. Figure 7 provides the code of the `fixList` procedure. As can be seen, the fixing mechanism requires only simple load and store operations.

There are two special cases we need to take care of: when the last node is being deleted and when the the `dummy` node needs to be skipped.

- The situation in which there is only one node in the queue is encountered when the `tail` and `head` point to the same node, which is not a `dummy` node. Deleting this node requires three steps and two CAS operations, as seen in Figure 6 Part A. First, a new node with a dummy value is created, and its next pointer is set to point to the last node. Second, the `tail` is atomically modified using a CAS to point to this `dummy` node, and then, the `head` is atomically modified using a CAS to also point to this `dummy` node. The intermediate state in which the `tail` points to a `dummy` node and the `head` points to another node is special, and occurs only in the above situation. This sequence of operations ensures that the algorithm is not blocked even if a `dequeue` method modified the `tail` to point to a `dummy` node and then stopped. We can detect the situation in which the `tail` points to a `dummy` node and the `head` does not, and continue the execution of the `dequeue` method. In addition, enqueuing methods can continue to insert new nodes to the queue, even in the intermediate state.
- In our algorithm, a `dummy` node is a special node with a `dummy` value. It is created and inserted to the queue when it becomes empty as explained above. Since a `dummy` node does not contain a real value, it must be skipped when nodes are deleted from the queue. The steps for skipping a `dummy` node are similar to those of a regular dequeue, except that no value is returned. When a `dequeue` method identifies that the `head` points to a `dummy` node and the `tail` does not, as in Figure 6 Part B, it modifies the `head` using a CAS to point to the node pointed by the `prev` pointer of this `dummy` node. Then it can continue to `dequeue` nodes.

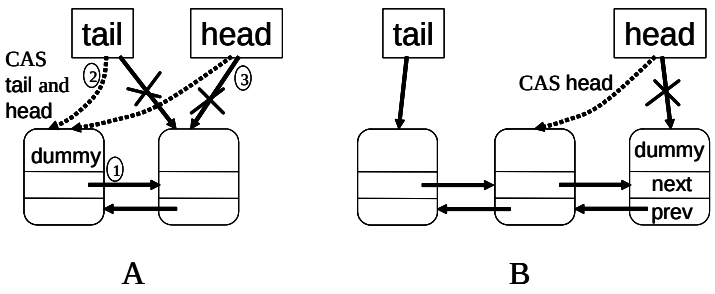

Fig. 6. A - A `dequeue` of the last node, B - Skipping the dummy node

```
F01:void fixList(queue_t* q, pointer_t tail, pointer_t head)
F02: pointer_t curNode , curNodeNext, nextNodePrev
F03: curNode = tail                                    # Set curNode to tail
F04: while((head == q->head) && (curNode != head)){  # While not at head
F05:   curNodeNext = (curNode.ptr)->next               # Read curNode next
F06:   if (curNodeNext.tag !=  curNode.tag){           # Tags don't equal?
F07:     return;                                       # ABA, return!
F08:   }
F09:   nextNodePrev = (curNodeNext.ptr)->prev          # Read next node prev
F10:   if (nextNodePrev != <curNode.ptr, curNode.tag-1>){#Ptr don't equal?
F11:     (curNodeNext.ptr)->prev = <curNode.ptr, curNode.tag-1>; # Fix
F12:   }
F13:   curNode = <curNodeNext.ptr, curNode.tag-1>    # Advance curNode
F14: }
```

Fig. 7. The `fixList` procedure

3 Solving the ABA Problem

An ABA situation [10, 14] can occur when a process read some part of the shared memory in a given state and then was suspended for a while. When it wakes up the part it read could be in an identical state, however many insertions and deletions could have happened in the interim. The process may incorrectly succeed in performing a CAS operation, bringing the data structure to an inconsistent state. To identify such situation and eliminate ABA, we use the standard tagging-mechanism approach [26, 14].

In the tagging-mechanism, each pointer (`tail`, `head`, `next`, and `prev`) is added a `tag`. The `tags` of the `tail` and `head` are initiated to zero. When a new node is created, the `next` and `prev` tags are initiated to a predefined null value. The `tag` of each pointer is atomically modified with the pointer itself when a CAS operation is performed on the pointer.

Each time the `tail` or `head` is modified, its `tag` is incremented, also in the special cases of deleting the last node and skipping the `dummy` node. If the `head` and `tail` point to the same node, their `tags` must be equal. Assume that an `enqueue` method executed by a process P read that the `tail` points to node A and then was suspended. By the time it woke up, A was deleted, B was inserted and A was inserted again. The `tag` attached to the `tail` pointing to A will now be different (incremented twice) from the `tag` originally read by P. Hence P's `enqueue` will fail when attempting to CAS the `tail`.

The ABA problem can also occur while modifying the `prev` pointers. The `tag` of the `next` pointer is set by the enqueuing process to equal the `tag` of the new `tail` it tries to CAS. The tag of the `prev` pointer is set to equal the `tag` of the `next` pointer in the same node. Thus consecutive nodes in the queue have consecutive `tags` in the `next` and `prev` pointers. Assume that an `enqueue` method executed by process P inserted a node to the queue, and stopped before

it modified the `prev` pointer of the consecutive node A (see Section 2.1). Then A was deleted and inserted again. When P woke up, it wrote its pointer and the `tag` to the `prev` pointer of A. Though the pointer is incorrect, the tag indicates this since it is smaller than the one expected. A `dequeue` method verifies that the `tag` of the `prev` pointer of the node it is deleting equals the `tag` of the `head` pointer it read. If the `tags` are different, it concludes that an ABA problem occurred, and calls a method to fix the `prev` pointer.

The fixing of the `prev` pointer after it was corrupted by an ABA situation is performed in the `fixList` procedure (Figure 7), in combination with the second stage of modifying the `prev` pointers, as explained in Section 2.1. In addition to using the fact that the `next` pointers are set locally by the `enqueue` method and never change, we use the fact that consecutive nodes must have consecutive `tags` attached to the `next` and `prev` pointers. The fixing mechanism walks through the entire list from the `tail` to the `head` along the `next` pointers of the nodes, correcting `prev` pointers if their `tags` are not consistent.

Finally we note that in garbage-collected languages such as the Java^{TM} programming language, ABA does not occur and the `tags` are not needed. When creating a new instance of a node, its `prev` pointer is set to NULL. Based on this, the fixing mechanism is invoked if the `prev` pointer points to NULL (instead of checking that the tags are equal). In this way we can detect the case in which an `enqueue` did not succeed in its optimistic update of the `prev` pointer of the consecutive node,and fix the list according to the `next` pointers.

4 Performance

We evaluated the performance of our FIFO queue algorithm relative to other known methods by running a collection of synthetic benchmarks on a 16 processor Sun Enterprise™ 6500, an SMP machine formed from 8 boards of two 400MHz UltraSparc® processors, connected by a crossbar UPA switch, and running a Solaris™ 9 operating system. Our C code was compiled by a Sun *cc* compiler 5.3, with flags `-xO5 -xarch=v8plusa`.

4.1 The Benchmarks

We compare our algorithm to the two-lock queue and to MS-queue of Michael and Scott [14]. We believe these algorithms to be the most efficient known lock-based and lock-free dynamic-memory queue algorithms in the literature. We used Michael and Scott's code (referenced in [14]).

The original Michael and Scott paper [14] showed only an *enqueue-dequeue pairs* benchmark where a process repeatedly alternated between enqueuing and dequeuing. This tests a rather limited type of behavior. In order to simulate additional patterns, we implemented an internal memory management mechanism. As in Micheal and Scott's benchmark, we use an array of nodes that are allocated in advance. Each process has its own pool with an equal share of these nodes. Each process performs a series of `enqueues` on its pool of nodes and `dequeues`

from the queue. A dequeued node is placed in dequeuing process pool for reuse. If there are no more nodes in its local pool, a process must first `dequeue` at least one node, and can then continue to `enqueue`. Similarly, a process cannot `dequeue` nodes if its pool is full. To guarantee fairness, we used the same mechanism for all the algorithms. We tested several benchmarks of which two are presented here:

- enqueue-dequeue pairs: each process alternately performed `enqueue` or `dequeue` operation.
- 50% enqueues: each process chooses uniformly at random whether to perform an enqueue or a dequeue, creating a random pattern of 50% `enqueue` and 50% dequeue operations.

4.2 The Experiments

We repeated the above benchmarks delaying each process a random amount of time between operations to mimic local work usually performed by processes (in the range of 0 to 1000 increment operations in a loop).

We measured latency (in milliseconds) as a function of the number of processes: the amount of time that elapsed until the completion of a total of a million operations divided equally among processes. To counteract transient startup effects, we synchronized the start of the processes (i.e., no process can start before all others finished their initialization phase).

We pre-tested the algorithms on the given benchmarks by running hundreds of combinations of exponential backoff delays. The results we present were taken from the best combination of backoff values for each algorithm in each benchmark (although we found, similarly to Michael and Scott, that the exact choice of backoff did not cause a significant change in performance). Each of the presented data points in our graphs is the average of eight runs.

4.3 Empirical Results

As can be seen in Figure 8, the new algorithm consistently outperforms the MS-queue in both the 50% and the enqueue-dequeue pairs benchmarks when there are more than two processes. From the enqueue-dequeue pairs benchmark one can also see that the lock-based algorithm is consistently worst than the lock-free algorithms, and deteriorates when there is multiprogramming, that is, when there are 32 processes on 16 processors. Hence, in the rest of this section, we concentrate on the performance of the MS-queue and our new algorithm.

Figure 8 shows that the results in both enqueue-dequeue pairs and 50% enqueues benchmarks were very similar, except in the case of one or two processes. To explain this, let us consider the overhead of an empty queue, the number of calls to the `fixList` procedure as it appears in the left side of Figure 9, and the number failed CAS operations as it appears in the right side of Figure 9.

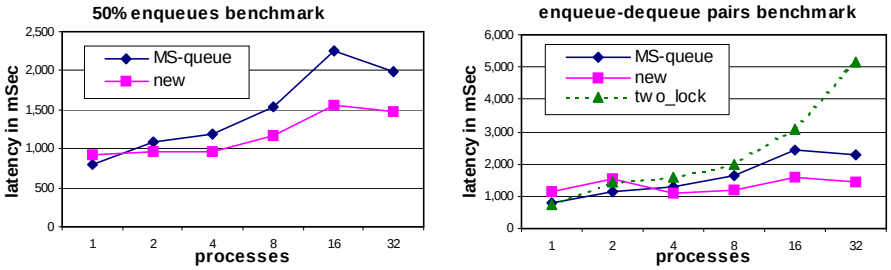

Fig. 8. Results of enqueue-dequeue pairs and 50% benchmarks

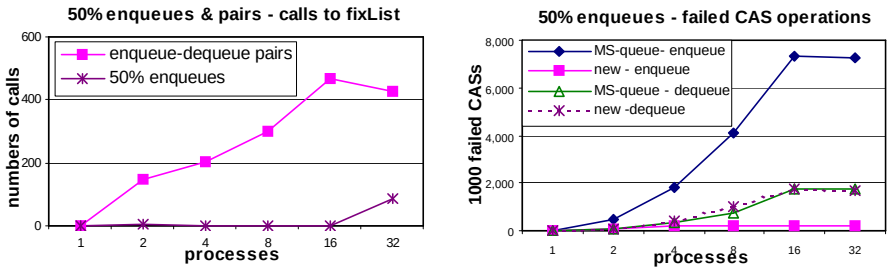

Fig. 9. The number of failed CASs and calls to fixList

– As described in Section 2.1 (see also Figure 6), additional successful CASs are required by the new algorithm when the queue becomes empty. As the number of concurrent processes increases, their scheduling causes the queue to become empty less frequently, thus incurring less of the overhead of an empty queue. A benchmark which we do not present here shows that this phenomena can be eliminated if the enqueue-dequeue pairs benchmark is initiated with a non-empty queue. In the 50% enqueues benchmark, due to its random characteristics, the overhead of an empty queue is eliminated even in low concurrency levels.
– Overall, there were a negligible number of calls to `fixlist` in both benchmarks, no more than 450 calls for a million operations. This makes a strong argument in favor of the optimistic approach.
  Recall that the `fixList` procedure is called when a process tries to `dequeue` a node before the `enqueuing` process completed the optimistic update of the `prev` pointer of the consecutive node. This happens more frequently in the enqueue-dequeue pairs benchmark due to its alternating nature. In the 50% enqueues benchmark, due to its more random patterns, there are almost no calls to `fixList` when the concurrency level is low, and about 85 when there are 32 processes.

– The righthanded side of Figure 9 shows the number of failed CAS operations in the `enqueue` and `dequeue` methods. These numbers expose one of the key performance benefits of the new algorithm. Though the number of failed CASs in the `dequeue` operations in both algorithms is approximately the same, the number of failed CASes in the `enqueue` of MS-queue is about 20 to 40 times greater than in our new algorithm. This is a result of the additional CAS operation required by MS-queue's `enqueue` method, and is the main advantage allowed by our new optimistic doubly-linked list structure.

We conclude that in our tested benchmarks, our new algorithm outperforms the MS-queue. The MS-queue's latency is increased by the failed CASs in the `enqueue` operation, while the latency of our new algorithm is influenced by the additional CASs when the queue is empty. We note again that in our presented benchmarks we did not add initial nodes to soften the effect of encountering an empty queue.

5 Correctness Proof

This section contains a sketch of the formal proof that our algorithm has the desired properties of a concurrent FIFO queue. A sequential FIFO queue as defined in [27] is a data structure that supports two operations: `enqueue` and `dequeue`. The `enqueue` operation takes a value as an argument, inserts it to the queue, and does not return a value. The `dequeue` operation does not take an argument, deletes and returns the oldest value from the queue.

We prove that our concurrent queue is linearizable to the sequential FIFO queue, and that it is lock-free. We treat basic read/write (load/store) and CAS operations as atomic actions, and can thus take the standard approach of viewing them as if they occurred one after the other in sequence [28].

In the following we explain the FIFO queue semantics and define the linearization points for each `enqueue` and `dequeue` operation. We then define the insertion order of elements to the queue. The correctness proof and the lock freedom property proof are only briefly described out of space limitations.

5.1 Correct FIFO Queue Semantics

The queue in our implementation is represented by a `head` and a `tail` pointers, and uses a `dummy` node. Each node in the queue contains a value, a `next` pointer and a `prev` pointer. All pointers, `head`, `tail`, `next` and `prev`, are attached with a `tag`. Initially, all tags are zero and the `head` and `tail` point to the `dummy` node.

The Compare-And-Swap (CAS) operation used in our algorithm takes a register, an *old* value, and a *new* value. If the register's current content equals *old*, then it is replaced by *new*, otherwise the register remains unchanged [29]. A successful CAS operation is an operation that modified the register.

The successfulness of the `enqueue` and `dequeue` operations depends on the successfulness of CAS operations performed in the execution. For any process,

the `enqueue` operation is always successful. The operation ends when a process successfully performed the CAS operation in line E07. A successful `dequeue` operation is one that successfully performed the CAS in D22. If the queue is empty, the `dequeue` operation is considered unsuccessful and it returns null.

Definition 1. *The linearization points of* `enqueue` *and* `dequeue` *operations are:*

- `Enqueue` *operations are linearized at the successful CAS in line E07.*
- *Successful* `dequeue` *operations are linearized at the successful CAS in line D29.*
- *Unsuccessful* `dequeue` *operations are linearized in line D06.*

Definition 2. *In any state of the queue, the insertion order of nodes to the queue is the reverse order of the nodes starting from the tail, linked by the* `next` *pointers, to the head.*

If the `dummy` *node is linked before the* `head` *is reached, then the insertion order is the same from the* `tail` *to the* `dummy` *node, the* `dummy` *node is excluded, and the node pointed by the* `head` *is attached instead of the* `dummy` *node. If the* `head` *points to the* `dummy` *node then the* `dummy` *node is excluded.*

5.2 The Proof Structure

In the full paper we show that the insertion order is consistent with the linearization order on the `enqueue` operations. We do that by showing that the `next` pointer of a linearized `enqueue` operation always points to the node inserted by the previous linearized `enqueue` operation, and that the `next` pointers never change during the execution. We then show that the correctness of the `prev` pointers can be verified using the `tags`, and fixed if needed by the `fixList` procedure. We also prove that in any state of the queue there is at most one node with a dummy value in the queue, and that the queue is empty if both `head` and `tail` point to a `dummy` node.

To finish the proof we show that the deletion order of nodes from the queue is consistent with the insertion order. This is done by proving that we can detect the case in which the optimistic update of the `prev` pointer did not occur (and also the case of an ABA situation) and fix it using the `tags` and the `fixList` procedure. We then show that when a `dequeue` operation takes place, the `prev` pointer of the node pointed by the `head`, always point to the consecutive node as dictated by the `next` pointers.

From the above we can conclude that our concurrent implementation implements a FIFO queue.

5.3 Lock Freedom

In order to prove that our algorithm is lock-free we need to show that if one process fails in executing an `enqueue` or `dequeue` operation, then another process have modified the `tail` or the `head`, and thus the system as whole made progress. We also need to show that the `fixList` procedure eventually ends. These properties are fairly easy to conclude from the code.

5.4 Complexity

It can be seen from the code that each `enqueue` and `dequeue` operation takes a constant number of steps in the uncontended case. The `fixList` procedure, in a specific state of the queue, requires all the running `dequeue` processes to go over all the nodes in the queue in order to fix the list. However, once this part of the queue is fixed, when ABA does not occur, all the nodes in this part can be dequeued without the need to fix the list again.

6 Conclusion

In this paper we presented a new dynamic-memory lock-free FIFO queue. Our queue is based on an optimistic assumption of good ordering of operations in the common case, and on the ability to fix the data structure if needed. It requires only one CAS operation for each enqueue and dequeue operation and performs constantly better than the MS-queue. We believe that our new algorithm can serve as a viable alternative to the MS-queue for implementing linearizable FIFO queues.

References

1. Gottlieb, A., Lubachevsky, B.D., Rudolph, L.: Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. ACM Trans. Program. Lang. Syst. **5** (1983) 164–189
2. Herlihy, M., Wing, J.: Linearizability: A correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems **12** (1990) 463–492
3. Hwang, K., Briggs, F.A.: Computer Architecture and Parallel Processing. McGraw-Hill, Inc. (1990)
4. Lamport, L.: Specifying concurrent program modules. ACM Transactions on Programming Languages and Systems **5** (1983) 190–222
5. Mellor-Crummey, J.M.: Concurrent queues: Practical fetch-and-ϕ algorithms. Technical Report Technical Report 229, University of Rochester (1987)
6. Prakash, S., Lee, Y.H., Johnson, T.: Non-blocking algorithms for concurrent data structures. Technical Report 91–002, Department of Information Sciences, University of Florida (1991)
7. Prakash, S., Lee, Y.H., Johnson, T.: A non-blocking algorithm for shared queues using compare-and-swap. IEEE Transactions on Computers **43** (1994) 548–559
8. Stone, H.S.: High-performance computer architecture. Addison-Wesley Longman Publishing Co., Inc. (1987)
9. Stone, J.: A simple and correct shared-queue algorithm using compare-and-swap. In: Proceedings of the 1990 conference on Supercomputing, IEEE Computer Society Press (1990) 495–504
10. Treiber, R.K.: Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center (1986)
11. Tsigas, P., Zhang, Y.: A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems. In: Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures, ACM Press (2001) 134–143

12. Valois, J.: Implementing lock-free queues. In: Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems. (1994) 64–69
13. Lea, D.: (The java concurrency package (JSR-166)) `http://gee.cs.oswego.edu/dl/concurrency-interest/index.html`.
14. Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC '96), New York, USA, ACM (1996) 267–275
15. Craig, T.: Building FIFO and priority-queueing spin locks from atomic swap. Technical Report TR 93-02-02, University of Washington, Department of Computer Science (1993)
16. Magnussen, P., Landin, A., Hagersten, E.: Queue locks on cache coherent multiprocessors. In: Proceedings of the 8th International Symposium on Parallel Processing (IPPS), IEEE Computer Society (1994) 165–171
17. Mellor-Crummey, J., Scott, M.: Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Transactions on Computer Systems **9** (1991) 21–65
18. Weaver, D., (Editors), T.G.: The SPARC Architecture Manual (Version 9). PTR Prentice Hall, Englewood Cliffs, NJ) (1994)
19. Intel: Pentium Processor Family User's Manual: Vol 3, Architecture and Programming Manual. (1994)
20. Agesen, O., Detlefs, D., Flood, C., Garthwaite, A., Martin, P., Moir, M., Shavit, N., Steele, G.: DCAS-based concurrent deques. Theory of Computing Systems **35** (2002) 349–386
21. Luchangco, V., Moir, M., Shavit, N.: On the uncontended complexity of consensus. In: Proceedings of Distributed Computing. (2003)
22. Herlihy, M., Luchangco, V., Moir, M.: Obstruction-free synchronization: Double-ended queues as an example. In: Proceedings of the 23rd International Conference on Distributed Computing Systems, IEEE (2003) 522–529
23. Rajwar, R., Goodman, J.: Speculative lock elision: Enabling highly concurrent multithreaded execution. In: Proceedings of the 34th Annual International Symposium on Microarchitecture. (2001) 294–305
24. Herlihy, M., Luchangco, V., Moir, M.: The repeat offender problem: A mechanism for supporting lock-free dynamic-sized data structures. In: Proceedings of the 16th International Symposium on DIStributed Computing. Volume 2508., Springer-Verlag Heidelberg (2002) 339–353 A improved version of this paper is in preparation for journal submission; please contact authors.
25. Michael, M.: Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In: The 21st Annual ACM Symposium on Principles of Distributed Computing, ACM Press (2002) 21–30
26. Moir, M.: Practical implementations of non-blocking synchronization primitives. In: Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing. (1997) 219–228
27. Cormen, T., Leiserson, C., Rivest, R., Stein, C.: Introduction to Algorithms. Second edition edn. MIT Press, Cambridge, MA (2001)
28. Afek, Y., Attiya, H., Dolev, D., Gafni, E., Merritt, M., Shavit, N.: Atomic snapshots of shared memory. In Dwork, C., ed.: Proceedings of the 9th Annual ACM Symposium on Principles of Distribted Computing, Québec City, Québec, Canada, ACM Press (1990) 1–14
29. Herlihy, M.: Wait-free synchronization. ACM Transactions on Programming Languages and Systems (TOPLAS) **13** (1991) 124–149