Distrib. Comput. (2008) 20:323–341
DOI 10.1007/s00446-007-0050-0

An optimistic approach to lock-free FIFO queues

Edya Ladan-Mozes · Nir Shavit

Received: 17 November 2004 / Accepted: 12 July 2007 / Published online: 30 November 2007

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 work presents a new dynamic-memory concurrent lock-free FIFO queue algorithm that in a variety of circumstances performs 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.

Keywords CAS · Compare and swap · Concurrent data structures · FIFO queue · Lock-free · Non-blocking · Synchronization

A preliminary version of this paper appeared in the proceedings of the 18th International Conference on Distributed Computing (DISC) 2004, pages 117–131.

E. Ladan-Mozes (✉)
CSAIL-MIT, 32 Vassar st, Cambridge, MA 02139, USA
e-mail: edya@MIT.EDU

N. Shavit
School of Computer Science, Tel-Aviv University,
69978 Tel Aviv, Israel

N. Shavit
Sun Microsystems Laboratories,
Burlington MA, USA

1 Introduction

First-in-first-out (FIFO) queues are among the most fundamental and highly studied concurrent data structures [5, 9,10,12,18,19,23,24,26–31], and are an essential building block of concurrent data structure libraries such as JSR-166, the Java Concurrency Package [13]. A concurrent queue is a linearizable structure that supports enqueue and dequeue operations with the usual FIFO semantics. This work 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 [19] (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 [19], and has been included as part of the Java 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 in a variety of benchmarks performs better than the MS-queue. It is perhaps the first practical example of the "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 optimistic queue algorithm

As with many finely tuned high performance algorithms (see for example CLH [4,16] Versus MCS [17] locks), the key to

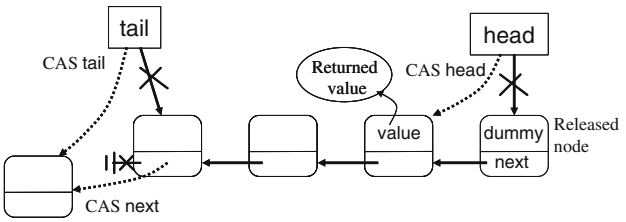

Fig. 1 The single CAS `dequeue` and costly two CAS `enqueue` of the MS-Queue algorithm

our new algorithm's performance is in saving a few costly operations along the commonly used 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 [11,33], even the successful CAS operations take an order-of-magnitude longer to complete than a load or a store, since they are implemented by taking exclusive ownership of a cache line and a flushing of the processor's write buffer.

The key idea in our new algorithm is to (literally) approach things from a different direction... by logically reversing the direction of `enqueues` and `dequeues` to/from the list. If `enqueues` 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 Fig. 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

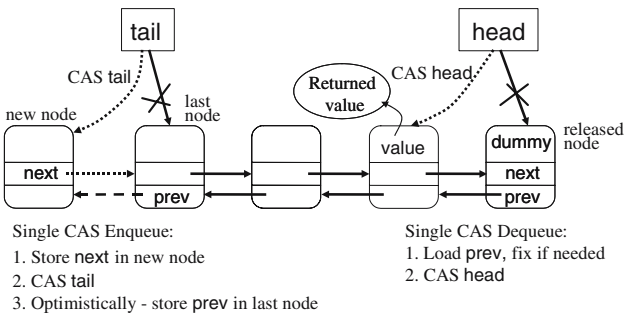

Fig. 2 The Single CAS `enqueue` and `dequeue` of the new algorithm

lock-free deque algorithms using double-compare-and-swap operations [2]. 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 based on a doubly linked list where pointers in both directions are set using simple stores, and both `enqueues` and `dequeues` 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. [15] 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. [8] and the lock-elision approach by Rajwar and Goodman [25] 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.

Independently of our work, Lea [14] has recently used an optimistic approach to implement the "successor pointers" direction of the linked list in a CLH lock [4,16]. A short survey of related work can be found in Sect. 7.

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 [19]. We used Michael and Scott's `C` code and compared it to our new FIFO queue algorithm on a 16-processors shared-memory machine using a collection of benchmarks, including the key benchmark used by Michael and Scott [19]. Our empirical results, presented in Sect. 4, show that in our benchmarks the new algorithm performs better than the MS-queue. This improved performance on our tested architecture is not surprising, as our enqueues require fewer costly CAS operations, and as our benchmarks show, generate significantly fewer failed CAS operations. We also found that our algorithm performs

better when pre-backoff and validation are performed on the `head` pointer *before* it is CASed in the `dequeue` operation.

The new algorithm uses the same dynamic memory pool structure as the MS-queue. It fits with memory recycling methods such as ROP [7] or Hazard Pointers [20], and it can be written in garbage-collected programming languages without the need for a memory pool or ABA-tags (see Sect. 3.6). We thus believe it can serve as a viable alternative to the MS-queue on certain architectures. We note however that our algorithm is significantly more sensitive to tuning parameters than the MS-queue, and is thus not the right solution for users who are looking for an out-of-the-box solution. Finally, we note that our benchmarking was conducted on a specific multiprocessor architecture, and one may well find that on other architectures such as new multi-core machines with their reduced cost for CAS operations, the practical value in using our new algorithm may be diminished. This question remains to be answered.

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

struct node_t {                          struct queue_t {
  data_type value;                         pointer_t tail;
  pointer_t next;                          pointer_t head;
  pointer_t prev;                        };
};

void init_queue(queue_t* q)
I01: node_t* nd = new_node()      # Allocate a new node
I02: nd->next = <null, 0>         # next points to null with tag 0
I03: q->tail = <nd, 0>;           # tail points to nd with tag 0
I04: q->head = <nd, 0>;           # head points to nd with tag 0
```

Fig. 3 Types, structures and initialization

```
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

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` and `dequeue` operations to be performed using a single CAS per operation. Our algorithm guarantees that this list is always connected and ordered by the order of `enqueue` operations in one direction. The other direction is optimistic and may be inaccurate at various points of the execution, but can be returned to an accurate state when needed.

In our algorithm we employ CAS synchronization operations in addition to normal *load* and *store* operations. The CAS operation, `CAS(`a`,`p`,`n`)`, atomically compares the value at memory location a to p and if they are the same it writes n to a and returns `true`, otherwise it returns `false`. Since our algorithm uses CAS for synchronization, ABA issues arise [19,29]. In Sect. 2.2, we describe the `enqueue` and `dequeue` operations ignoring ABA issues. The tagging mechanism we added to overcome the ABA problem is similar to the one used in [19] and is explained in more detail in Sect. 3. The code in this section includes this tagging mechanism.

2.1 The optimistic queue data structure

Our shared queue data structure (see Fig. 3) consists of a `head` pointer, a `tail` pointer, and nodes. Each node contains a `value`, a `next` pointer and a `prev` pointer. When a new node is created, its `next` and `prev` pointers are initialized to a null value. When the queue is first initialized, a `dummy` node, a node whose value is not counted as a value in the queue, is created and both `head` and `tail` are set to point to it. During the execution, the `tail` always points to the last (youngest) node inserted to the queue and the `head` always points to a dummy node. The node that contains the first (oldest) value in the queue is the node whose `next` pointer points to that dummy node. When the queue becomes empty, both `head` and `tail` point to the same node now considered to be a dummy.

2.2 The optimistic queue operations

A FIFO queue supports two operations (which we will alternately call *methods*, especially when discussing their implementation code): `enqueue` and `dequeue`. The `enqueue` method inserts a value to the queue and the `dequeue` method deletes the oldest value from the queue.

The code of the `enqueue` method appears in Fig. 4, and the code of the `dequeue` method appears in Fig. 5. To insert a value, the `enqueue` method creates a new node that contains the value (E02–E03), and then tries to insert this node into the queue. The `enqueue` reads the current `tail` of the queue (E05), and sets the new node's `next` pointer to point to that same node (E06). Then it tries to atomically modify the `tail` to point to this new node using a CAS operation (E07). If the CAS succeeded, the new node was inserted into the queue, and the `enqueue` process updates the `prev` pointer of the node pointed-to by the `tail` read in E05, and exits (E08–E09). Otherwise the enqueue retries.

To delete a node, a `dequeue` method reads the current `head` and `tail` of the queue (D04–D05), and the `prev` pointer of the node pointed-to by the `head` (D06). To verify that these `head` and `tail` existed together in some state of

```
data_type dequeue(queue_t* q)
D01: pointer_t tail, head, firstNodePrev
D02: data_type val
D03: while(TRUE){                          # Try till success or empty
D04:   head = q->head                      # Read the head
D05:   tail = q->tail                      # Read the tail
D06:   firstNodePrev = (head.ptr)->prev    # Read first node prev
D07:   if (head == q->head){               # Check consistency
D08:     if (tail != head){                # Queue not empty?
D09:       if (firstNodePrev.tag != head.tag){  # Tags not equal?
D10:         fixList(q, tail, head)        # Call fixList
D11:         continue                      # Re-iterate (D04)
D12:       }
D13:       val = (firstNodePrev.ptr)->value     # Read the value to return
D14:       if CAS(&(q->head), head, <firstNodePrev.ptr,head.tag+1>){# CAS
D15:         free (head.ptr)               # Free the node at head
D16:         return val                    # Dequeue done!
D17:       }
D18:     }
D19:     else{                             # Only one node
D20:       return NULL                     # Empty queue, done!
D21:     }
D22:   }
D23: }
```

Fig. 5 The `dequeue` operation

the queue, the `dequeue` method checks that the `head` was not changed since it was last read in D05 (D07).

In Lines D09–D11 the `dequeue` method verifies that the `prev` pointer is correctly set, as will be explained in Sects. 2.3 and 3. If this condition is satisfied, then the `dequeue` method reads the value stored in the node pointed-to by the `prev` pointer (D13) and then tries to CAS the `head` to point to that same node (D14). If it succeeded, then the node previously pointed by the `head` is freed (D15) and the value read in Line D13 is returned (D16). If it failed, it repeats the above steps.

When the queue is empty both `head` and `tail` point the same node. In this case the condition in Line D08 fails and a null is returned (D20).

As can now be seen from the code, the `enqueue` operation requires only one CAS to insert a node to the queue, a win over the MS-queue.

2.3 Updating the `prev` pointer

We now explain how we update the `prev` pointers of the nodes in a consistent and lock-free manner using only simple store operations. In this section we ignore ABA problems and memory integrity issues. These issues are discussed in Sect. 3.

The `prev` pointer of a node is optimistically stored immediately after the successful insertion of a new node (E08). In Line E08, an `enqueue` method updates the `prev` pointer of the node previously pointed by the `tail` to point to the new node it inserted in Line E07. Once this write is completed, the `prev` pointer points to its preceding node in the list.

Unfortunately, the storing of the `prev` pointer by an inserting process might be delayed for various reasons. Thus a dequeuing method that tries to remove a node from the queue might see a null value in the `prev` pointer instead of a correct value.

```
F01:void fixList(queue_t* q, pointer_t tail, pointer_t head)
F02: pointer_t curNode , curNodeNext
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:   (curNodeNext.ptr)->prev = <curNode.ptr, curNode.tag-1>; # Fix
F07:   curNode = <curNodeNext.ptr, curNode.tag-1>    # Advance curNode
F08: }
```

Fig. 6 The `fixList` procedure

To fix the `prev` pointer and to enable the `dequeue` method to proceed, the `dequeuing` process invokes the `fixList` procedure in Line D10. In the `fixList` procedure we use the fact that the `next` pointer of each node is set only by the method that inserted that node, and never changes until the node is dequeued. Thus it is possible to reconstruct the necessary `prev` pointer from the sequence of the `next` pointers. Figure 6 provides the code of the `fixList` procedure. The fixing mechanism walks through the entire list from the `tail` to the `head` along the chain of `next` pointers (F04–F05, F07), and corrects the `prev` pointers accordingly (F06). As can be seen, the fixing mechanism requires only simple load and store operations. Though the fixing mechanism traverses the whole list, our experiments show that this linear complexity is mitigated by the fact that it is rarely called and is inexpensive in its per node overhead. For efficiency, the `fixList` procedure stops if the `head` was changed (F04), which means that another process already completed the `fixList` procedure.

3 The ABA problem and memory integrity

An ABA situation [19,29] 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 location it read could be in an identical state, though many insertions and deletions could have changed the state of the queue in the interim period. The process may then incorrectly succeed in performing a CAS operation, bringing the data structure to an inconsistent state. In our algorithm, another type of an ABA problem can occur due to long delays: a slow process can write incorrect information to the `prev` pointer of a node that was already deleted from the queue, or one that was inserted again. To identify such situations and eliminate ABA, we use a standard tagging-mechanism approach [19, 21] and extend its use as explained in Sects. 3.1, 3.2 and 3.3. In Sects. 3.4 and 3.5 we discuss the roll-over problem and memory integrity issues that arise from the need to handle tags and ABA situations.

3.1 The tagging mechanism

In our tagging-mechanism, each pointer (`tail`, `head`, `next`, and `prev`) is added a `tag` field, a part of the address

that is used to timestamp the reference stored in the variable. The size of this tag, as we discuss shortly, is architecture dependent. When the queue is initialized, the tags of the tail and head pointers are initialized to zero. The tag of the next pointer of the dummy node that the tail and head are initially set to point to is also initialized to zero. When a new node is created, the next and prev tags are initialized to a predefined null value.

As can be seen from the code, the tag of each pointer is atomically read and/or modified by any operation (load, store, or CAS) on a reference pointer.

We apply the following tag manipulation rules:

- When the tail or head is CASed, its tag is incremented by 1.
- When a new node is inserted into the queue, the tag of its next pointer is set by the enqueueing process to be one greater than the current tag of the tail. Thus, once this process successfully CAS the tail, these tags will be equal.
- When the prev is stored by an enqueueing process, its tag is set to equal the tag of the next pointer in the same node.

These modifications of the tags prevent a situation in which a CAS on the tail or head succeeds incorrectly. We now explain how this is achived.

3.2 Preventing the ABA problem in the tail and head pointers

Consider a situation in which 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. Without the tagging mechanism, process *P* would have incorrectly succeeded in performing the CAS, since it had no way to distinguish the two states of the data structure from one another. The tagging mechanism, however, will cause the tag of the tail pointing to *A* to differ (because it was incremented twice) from the tag originally read by *P*. Hence, *P*'s incorrect enqueue will fail when attempting to CAS the tail.

3.3 Detecting and fixing the ABA problem in the prev pointer

Adding a tag to the tail and head pointers and incrementing it each time these pointers are CASed, readily prevents ABA situations resulting from incorrect successful CAS operations. This is not the case in ABA situations that result from out-of-date stores to prev pointers. Since the prev pointers and their tags are only stored, and can be stored by any process (for example, a process that executes the fixList method), the ABA problem can also occur while modifying the prev pointers. Recall that according to our tagging rules stated above the tags of the next pointer and the prev pointer in the same node are set to be equal. Thus, if there is a situation in which they are not equal, then this is an inconsistency.

The following example illustrates this type of ABA situation concerning the prev pointer. Assume that an enqueue method executed by process *P* inserted a node into the queue, and stopped before it modified the prev pointer of the successor node *A* (see Sect. 2.2). Then node *A* was deleted and inserted again. When *P* wakes up, it writes the pointer and the tag to the prev pointer of *A*. The tag value exposes this inconsistency since it is smaller than the one expected.

The detection and fixing of an incorrect prev pointer is based of the following properties that are maintained by the tagging mechanism:

Propoerty 1 The tag of the next pointer should be equal to the tag of the prev pointer in the same node.

Propoerty 2 The tag of the head pointer equals the tag of the next pointer of the node it points to.

If these properties are maintained, it follows that in a correct state, the tag of the head should equal the tag of the prev pointer of the node the head pointed-to. If these tags are different, we can conclude that an ABA problem occurred, and call the fixList method to fix the prev pointer.

These properties follow from the use of the tagging mechanism given a proper initialization of the queue as described in Sect. 3.1, together with the fact that the next pointer of a node, including its tag, is never changed during the execution until this node is deleted. Note that for the same reasons, Property 2 applies also to the tail pointer and the node it points to, and that if the head and tail point to the same node, their tags are equal (Lemma 13).

Property 1 will be proved in Sect. 5 by Lemma 11 and Property 2 will be proved in Lemma 12.

As can be seen from the code, the fixing of a prev pointer after it was corrupted is performed in the fixList procedure (Fig. 6). 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. This fixing mechanism further extends the use of the tagging mechanism, making use of the fact that (1) the next pointers are set locally by the enqueue method, (2) they never change until the node is deleted from the queue, and (3) that consecutive nodes must have consecutive tags for the next and prev pointers.

3.4 The rolling-over of tags

Most of the current computer architectures support atomic operations such as CAS only on a single computer word. To allow the pointer and its tag to be modified atomically they must reside in the same word. Thus, in our implementation as well as in [19], the computer's words in which the `tail`, `head`, `next`, and `prev` pointers are stored are logically divided into two equal parts. The first part is used to store a pointer, and the second is used to store the tag.

Since the pointer occupies only half of the computer word, it cannot store a real memory location. Rather, (as in [19]), our implementation uses a pre-allocated pool of nodes. The first part of each pointer points to a node in this pool. The pool forms a closed system of nodes which, as we will prove, allows us to maintain memory integrity.

Inconsistencies can result from tags "wrapping around." First, if by the time a slow process wrote to a `prev` pointer the tags rolled-over in such a way that the `tag` of the `prev` pointer it wrote equals the `tag` of the `next` pointer in the same node (as in a correct setting), then the tag is correct but the pointer points incorrectly, and we cannot detect that. Secondly, it is possible that a slow process reads a pointer and its tag before being suspended. By the time it wakes up, the tags have rolled-over so that the same pointer and the same tag it read are encountered again, then the queue data structure becomes inconsistent.

In practice, on today's 64-bits architectures, there are $2^{32}-1$ numbers that can be used for tag values, and assuming one of the operations is performed each microsecond, a thread must be delayed for more than an hour in order for the tags to roll-over. We can have the operating system abort such unreasonably delayed threads after an appropriate time limit. This will prevent any process from seeing the same value twice due to tags roll-over.

3.5 Preserving memory integrity

Memory integrity issues arise in our algorithm since any process can write to the `prev` pointer of any node it encountered during its execution using a simple store operation, even if at the actual writing time this node is no longer in the queue.

The external pool in which nodes are pre-allocated, as described in Sect. 3.4, is a standard approach to preserving memory integrity. The queue and external pool form a closed system in which non-interference is kept. That is, when a delayed process writes to a `prev` pointer of some node, it might corrupt this prev pointer (and in Sect. 3.3 we explained how the algorithm overcomes such corruption). However, as we show, at that point the node containing this pointer will be either in the queue, or in the pool and out of use, and overwriting it will not break the memory integrity.

Note that since a pre-allocated pool of nodes is used, the size of the `value` field stored in each node is fixed. In our implementation, as in [19], the value field is limited to the size of a word. In order to use the queue for other (even different) data types, the value field can store a pointer to the memory location of that data.

3.6 Garbage collected languages — a simple solution

In garbage-collected languages such as the Java programming language, the ABA problem does not occur and memory integrity is preserved, altogether eliminating the need to use tags.

In garbage-collected languages a node will not be collected as long as some process has a pointer to it implying that the same memory space cannot be simultaneously re-allocated to both the queue or to some other application. That means that if any process has a pointer to a node, this node will not be re-inserted to the queue or used by any other application until the last process releases the pointer to that node. Thus, garbage collection actually provides us the property of non-interference and eliminates the memory integrity problem as described in Sect. 3.5

Let us examine the ABA problem that occurred in the `head` and `tail` pointer in non-garbage collected languages as explained in Sect. 3.2. In that case, a process read some location in the shared memory in a given state and then was suspended for a while. When it woke up, the location it read was in an identical state, however many insertions and deletions already happened in the interim. In garbage-collected languages such situation cannot occur. Until the process that was suspended releases the pointer to the node, the memory space of this node cannot be re-used, and therefore, an identical state of the data structure cannot occur.

The same "solution" applies to the ABA problem in the `prev` pointer as explained in Sect. 3.3. If an `enqueue` process did not update the `prev` pointer of the node pointed by the new node it just inserted to the queue, then this `prev` pointer is null. When a `dequeue` process encounters this null value (instead of checking tags for equality in Line D10), it calls the `fixList` method. Since this node was not re-used before all processes released their pointer to it in the previous use of this memory location, the processes that currently can have a pointer to it are:

- The `enqueue` process that inserted that node.
- The `enqueue` process that inserted the successive node.
- All `dequeuing` processes that started to run the `fixList` method after this node was inserted to the queue.

Note that the process that inserted this node does not try to update its prev pointer. The process that inserted the

successive node tries to store a reference to the node it just inserted in this `prev` pointer. The `fixList` method, as before, traverses the chain of `next` pointers, which cannot be changed after a node was inserted to the queue, and updates the `prev` pointers accordingly. Therefore all these processes attempt to write the exact same value to this pointer, and this value points to the node inserted after it, as required. Even after the node is deleted from the queue, its memory is not re-used as long as any of the above processes did not release a pointer to it.

It follows that garbage collection eliminates the need for the tagging mechanism and for the external pool of pre-allocated nodes, allowing us to use a full word to store a pointer to a node. We note that using memory recycling methods such as [7,20] in languages that are not garbage-collected will also eliminate the memory integrity problem and thus the need for ABA `tags`.

A concise version of Java code for the queue data structure and the `enqueue` and `dequeue` methods appears in Sect. 6. Our Java based version of the algorithm follows that of the `ConcurrentLinkedQueue` class in the Java concurrency package, which implements the MS-queue algorithm. Following the notation used in `ConcurrentLinkedQueue`, the enqueue operation is called `offer` and the dequeue operation is called `poll`.

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 Enterprize™ 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 compared our algorithm to the *two-lock queue* and to *MS-queue* of Michael and Scott [19]. We believe these algorithms to be the most efficient known lock-based and lock-free dynamic-memory queue algorithms in the literature. In our benchmarks we used Michael and Scott's code (referenced in [19]).

The original paper of Michael and Scott [19] evaluated the lock-based and lock-free FIFO queue only on the *enqueue-dequeue pairs* benchmark, in which a process repeatedly alternated between enqueuing and dequeuing. This tests a rather limited type of behavior. In order to simulate additional patterns, we extended the memory management mechanism, implemented as an external pool of nodes. As in Michael 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. An `enqueue` operation takes a node from the process's pool and inserts it to the queue. The `dequeue` operation places the dequeued node in the dequeuing process's pool. If there are no nodes left in its local pool, a process must first `dequeue` at least one node, and then it can continue to `enqueue`. Similarly, a process cannot `dequeue` nodes if its pool is full. To guarantee fairness, we used this extended memory management mechanism for all the algorithms we tested. We ran several benchmarks:

- 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.
- grouped operations: each process picks a random number between 1 and 20, and performs this number of enqueues or dequeues. The process decides to perform enqueues or dequeues either uniformly as in the 50% benchmark or alternately as in the enqueue–dequeue pairs benchmark. Note that the total number of enqueue and dequeue operations is not changed, they are only distributed differently along the execution.

4.2 The experiments

We repeated the above benchmarks with and without "work" delays between operations. When delays are introduced, each process is delayed a random amount of time between operations to mimic local work usually performed by processes (quantified by the variable `work`)

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 on enqueues and dequeues. The results we present were taken from the best combination of backoff values for each algorithm in each benchmark. Similarly to Michael and Scott, we found that for MS-queue algorithm the choice of backoff did not cause a significant change in performance. For the algorithm presented in this paper, however, we found that backoff does effect the performance. Therefore we present two implementation of our algorithm, in the first one (denoted `new - no pre-backoff`) backoff is called only after a CAS operation fails. In the second one (denoted `new - with pre-backoff`), a process both calls

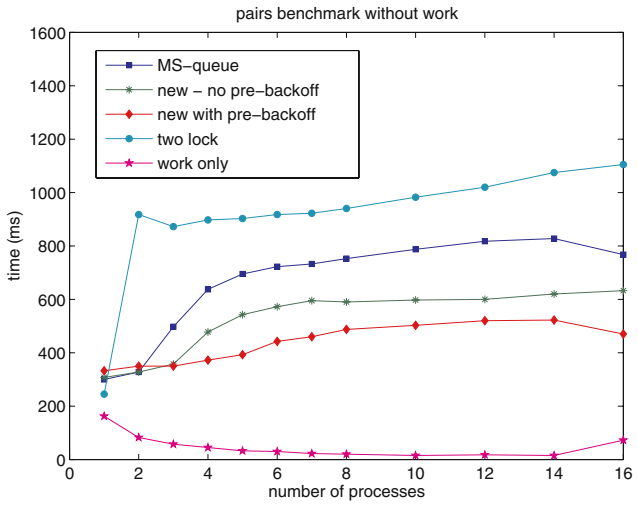

Fig. 7 Results of enqueue–dequeue pairs benchmark without work

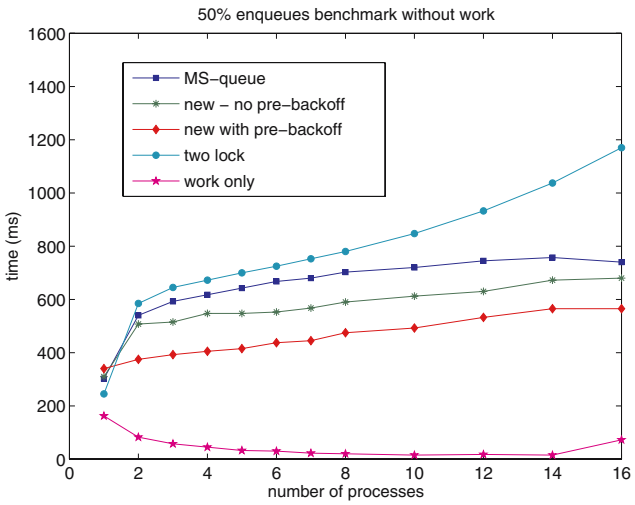

Fig. 9 Results of 50% enqueues benchmark without work

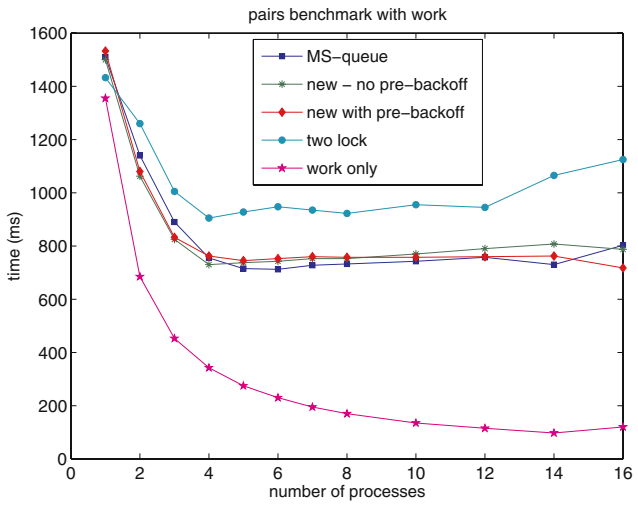

Fig. 8 Results of enqueue–dequeue pairs benchmark with work between operations

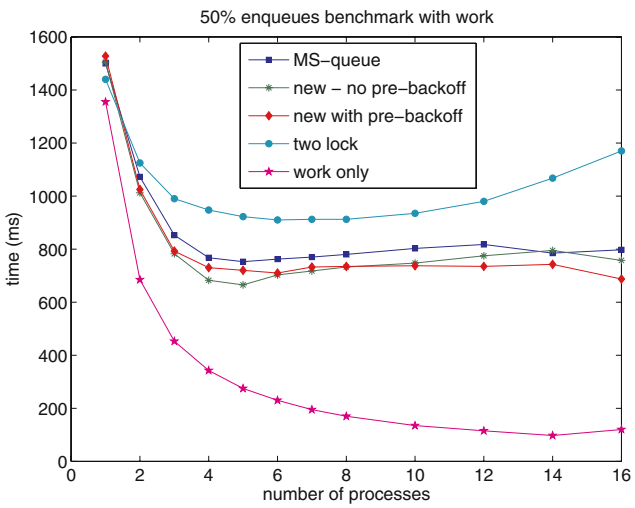

Fig. 10 Results of 50% enqueues benchmark with work between operations

backoff and validates the `head` pointer *before* it tries to perform the CAS in the `dequeue` operation (Line D14). If the validation fails then the `dequeue` process skips the CAS operation and restarts the `dequeue` operation. Each of the presented data points in our graphs is the average of four runs.

4.3 Empirical results

Figures 7 and 8 depict the performance results for the enqueue–dequeue pairs benchmark without and with work between operations. In Fig. 7 the `work only` line measures the time it takes to initiate the queue and run through the loop of operations, without actually performing enqueues and dequeues. The `work only` line in Fig. 8 adds to that the work being done by the processes between enqueue and dequeue operations.

Figures 9 and 10 depict the performance results for the 50% enqueues benchmark without and with work between operations, where the `work only` lines have the same meaning as in Figs. 7 and 8.

As can be seen, when no work between operation is introduced, our new algorithm is consistently better than MS-queue in both benchmarks. When we use pre-backoff and validation in the `dequeue` operation, the performance gap between MS-queue and our algorithm is even bigger. When work is introduced, the performance of the two algorithms in the enqueue–dequeue pairs benchmark is very similar, and in the 50% enqueues benchmark there is a small gap.

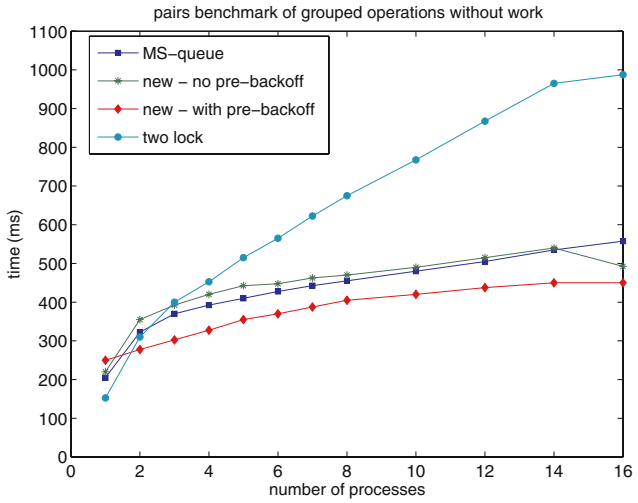

Fig. 11 Results of grouped operation using enqueue–dequeue pairs benchmark without work

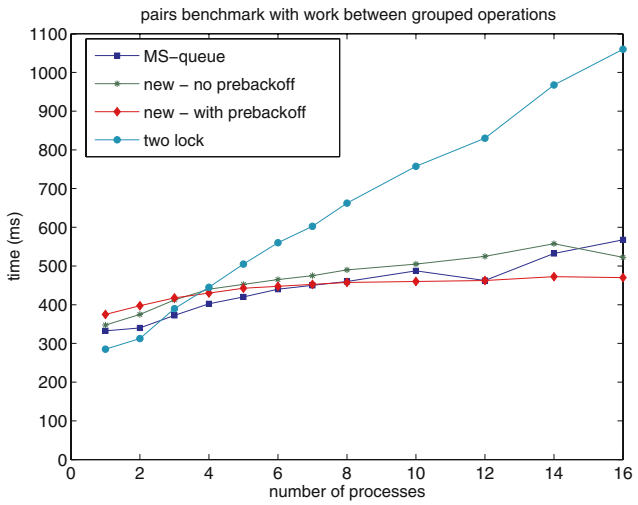

Fig. 12 Results of grouped operation using enqueue–dequeue pairs benchmark with work between operations

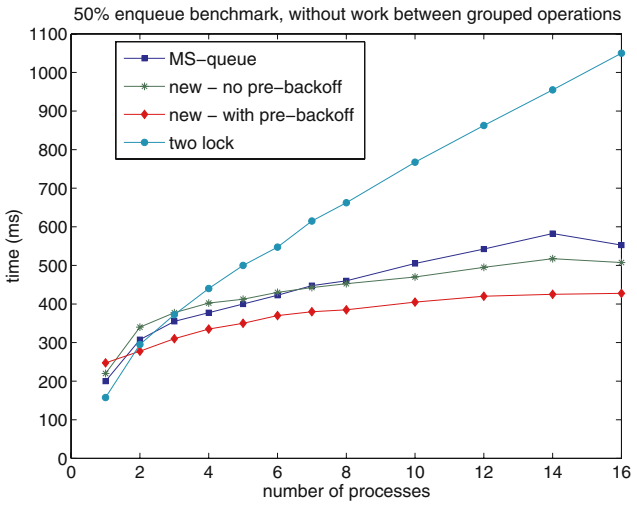

Fig. 13 Results of grouped operation using 50% enqueues benchmark without work

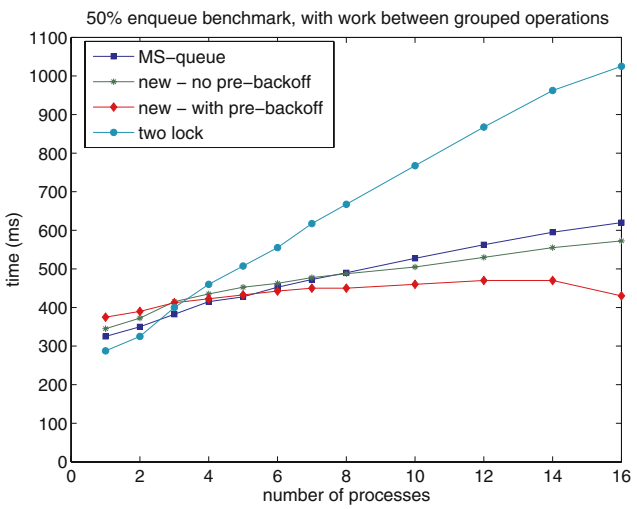

Fig. 14 Results of grouped operation using 50% enqueues benchmark with work between operations

In all four graphs, the blocking algorithm (two-lock) is worse than the non-blocking algorithms, and its performance deteriorates as the number of processes increases.

Figures 11, 12, 13 and 14 present the performance results for grouped operations using both the enqueue–dequeue pairs benchmark and the 50% enqueues benchmark (without and with work between operations). In these benchmarks each process performs a group of either enqueue or dequeue operations at a time. The graphs show a similar behavior to the one observed when the operations were not grouped, but with gaps between the non-blocking algorithms is smaller. The blocking algorithm, however, deteriorates faster and the gap between it and the non-blocking algorithms is bigger.

There are three main reasons for the performance gap between the non-blocking algorithms. First, there were a negligible number of calls to `fixList` in both benchmarks — no more than 5 calls for a million operations. From this we can conclude that almost all `enqueue` operations were able to store the correct `prev` pointer after the CAS operation in Line E08. This makes a strong argument in favor of the optimistic approach.

The second reason is that the `fixList` procedure runs fast - even though each process traverses the entire list, it requires only load and store operations. We also ran experiments in which we initialized the queue with different number of nodes, connected only by their next pointers; the running time r emained the same. This is due to the fact that once the `fixList` procedure is called, it traverses and fixes the entire list. Though many such processes can run concurrently, they complete quickly, and perform the fixing for all following dequeuing processes.

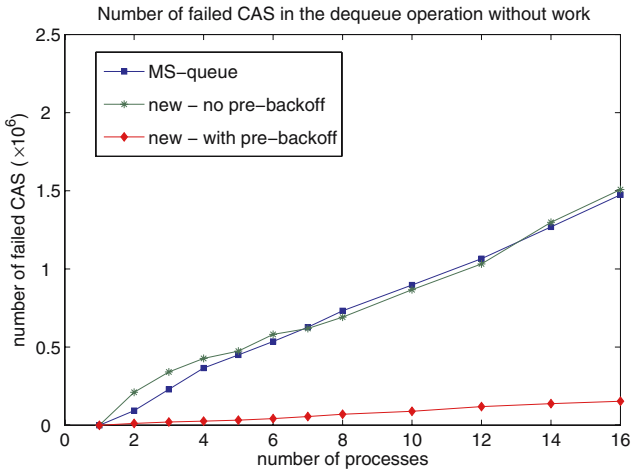

Fig. 15 Number of failed CAS in the dequeue operation without work

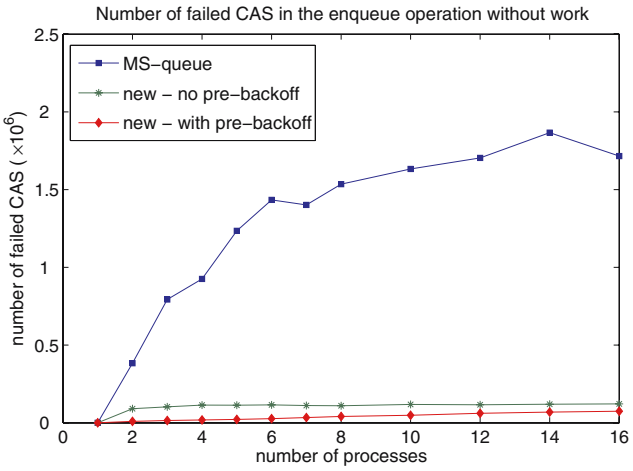

Fig. 17 Number of failed CAS in the enqueue operation without work

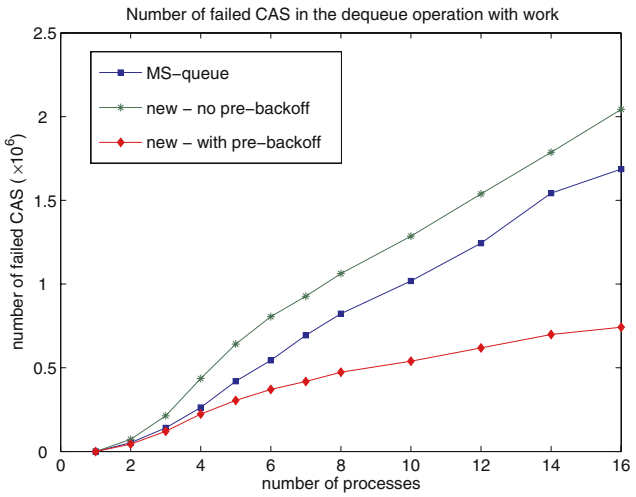

Fig. 16 Number of failed CAS in the dequeue operation with work

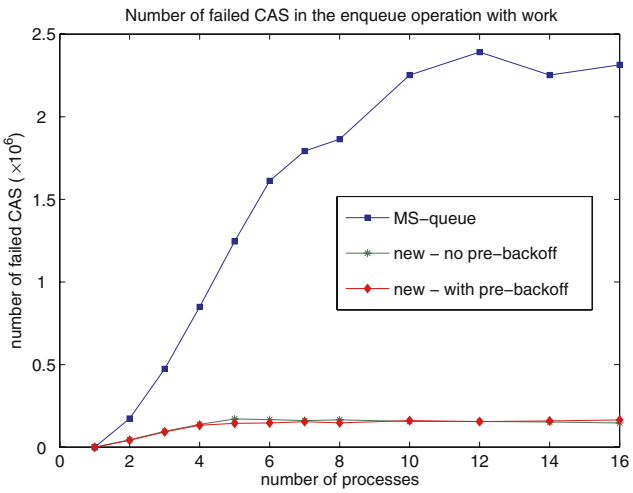

Fig. 18 Number of failed CAS in the enqueue operation with work

The third reason for the performance gap is the number of failed CAS operations performed by the algorithms. Figures 15 and 16 show the number of failed CAS operations in the dequeue method without and with work. Figures 17 and 18 show the number of failed CAS operations in the enqueue method without and with work. The numbers for the enqueue–dequeue pairs benchmark and the 50% benchmark are similar.

These graphs show the main advantage of our algorithm, and the importance of the pre-backoff and validation for our algorithm. When no work is introduced (Fig. 15), the number of failed CAS operations in the dequeue method is similar for MS-queue and for our algorithm without pre-backoff. Adding pre-backoff and validation reduces the number of failed CASes by an order of magnitude. The introduction of work (Fig. 16) causes more failed CAS operations in our algorithm, though this number is still much smaller when pre-backoffing and validation are used. Adding work imposes different scheduling patterns on the operations and has an affect similar to backoffing. As mentioned before, for MS-queue, we observed similar results with and without backoffs. Our algorithm is more sensitive to the tuning of the backoff mechnism, and therefore introducing work between operations causes more contention and hence more failed CAS operations.

Figures 17 and 18 show the importance of having only one CAS operation in the enqueue method, instead of two as in MS-queue. These graphs show that there are an order of magnitude less failed CAS operations in the enqueue method in our new algorithm, both with and without pre-backoff, and with work and without work between operations. This is a result of the additional CAS operation required by the MS-queue `enqueue` method, and is the main advantage allowed by our new optimistic doubly - linked list structure.

One can observe that in our tested benchmarks, the difference in the number of failed CAS operations is much bigger

than the performance gained by the reduction of this number. This fact suggests that reducing the number of CAS operations performed (both successful and unsuccessful) is only one component in gaining performance. The better performance achieved by our algorithm using pre-backoff and validation emphasis the importance of these techniques.

Based on the above benchmarks, we believe that our new algorithm can serve as a viable alternative to the MS-queue for implementing linearizable FIFO queues on architectures in which the cost of a CAS or similar universal synchronization operation is high.

5 Correctness proof

This section contains a proof that our algorithm has the desired properties of a lock-free FIFO queue.

5.1 Modelling shared memory

Our model of multiprocessor computation follows [9], though for brevity, we will use operational style arguments. We take the standard approach and view memory as being linearizable [9,12]. This means that we treat basic read/write (load/store) and CAS operations as atomic actions. Linearizability assures us, among other properties, that when two memory locations x and y are initialized to 0, and processor A runs a program which writes 1 to x and then 1 to y, then if processor B reads y and sees that it is 1, a subsequent read of x will return 1 as well.

Since we treat basic read/write (load/store) and CAS operations as atomic actions, we can take the standard approach of viewing them as if they occurred one after the other in sequence [1], and denote the union of all shared and local memory after each such operation as the *system state*. Henceforth, all our arguments will be on such sequences of operations and states in a given execution.

5.2 Memory management

Our memory management module is based on an external shared pool of nodes, from which new nodes are taken and into which dequeued nodes are freed, in the same style used by [19]. This external pool is managed by a different management layer. A process is granted a new node from the pool by invoking a `newNode()` method. We assume that the `next` and `prev` pointers of this node are initialized to null, and once a process is granted a new node from the pool, this node is no longer in the pool and it cannot be given to another process until it is returned to the pool. A node is returned to the pool by invoking a `free()` method on it.

5.3 A concurrent FIFO queue specification

A sequential FIFO queue as defined in [3] 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, or returns null if the queue is empty. In our implementation we encapsulate the values inside nodes. Thus, the `enqueue` method allocates a new node from the pool of nodes in which the value passed to it is placed, and the `dequeue` method releases a node back to that pool. We prove that our concurrent queue implementation is lock-free and that it is linearizable to the sequential FIFO queue specification.

5.4 Linearizability proof

To show that our algorithm is linearizable to a sequential FIFO queue, we define linearization points for each `enqueue` and `dequeue` operation, and then show that any execution of our implementation with these linearization points is equivalent to an execution of the sequential FIFO queue.

Let a *successful* CAS operation be one that modified the memory location it accessed.

Definition 1 The linearization points of the `enqueue` and `dequeue` methods are:

- `enqueue` methods are always successful and are linearized at the successful CAS in Line E07,
- successful `dequeue` methods, that is, ones returning a non-null value, are linearized at the successful CAS in Line D14, and
- unsuccessful `dequeue` methods, ones that return null because the queue is empty, are linearized in Line D05.

Definition 2 A new node created in Line E02 is considered to be *in the queue* after the successful CAS on the `tail` in Line E07 which sets the `tail` to point to this node, and until a successful CAS on the `head` in Line D14 redirects the `head` pointer from it. The *state of the queue* consists of all the nodes that are *in the queue*. In the *empty state of the queue* there is only one node in the queue, and it is considered to be a dummy node.

Note that both the nodes pointed by the `tail` and the `head` are in the queue, however the value in the node pointed by the `head` itself is always considered a dummy value.

The above definition defines only which nodes are in the queue, but does not assume any order between them. This order will be argued later on when we prove the main theorem, Theorem 1. In Theorem 1 we will show that the nodes in

the queue are ordered according to the linearization order of the `enqueue` operations. We will then be able to conclude that our concurrent algorithm is linearizable to a sequential implementation of a FIFO queue.

We first prove some basic properties of the states of the queue, properties that follow directly from the code. Based on these properties we will then prove that Theorem 1 holds in any state of the queue. We use the following technical definitions in the proofs:

Definition 3 Consider two nodes *A* and *B* in the queue, where the `next` pointer of node *B* points to node *A*. Node *A* is called the *successor* of node *B,* and node *B* is called the *predecessor* of node *A*.

Definition 4 Consider a successful CAS on the *tail* (*head*). The old value replaced by this CAS operation is denoted as the *replaced tail* (*replaced head*), and the new value of the *tail* (*head*) immediately after this successful CAS is denoted as the *modified tail* (*modified head*).

Definition 5 Define the state of the queue when the queue is first initialized as the *initial state*.

Lemma 1 *In the initial state the queue is empty. In addition, the* `tag` *of the* `tail`*, the* `tag` *of the* `head` *and the* `tag` *of the* `next` *pointer of the dummy node are all equal.*

Proof From the code of the `init_queue` method, a new node is created in Line I01 and both `head` and `tail` point to it in Lines I03 and I04. Thus by Definition 2 the queue is empty when it is first initialized and this node is considered dummy. Also from the code, in lines I02 the `tag` of the `next` pointer of this dummy node is set to zero, and in Lines I03 and I04 the `tags` of the `tail` and the `head` are also set to zero. □

Lemma 2 *In the state immediately following a successful CAS on the* `tail` *in Line E07, the following hold:*

- *The* `modified tail` *points to the new node created by the process performing the CAS, and its tag is one greater than the* `replaced tail's tag`.
- *The* `next` *pointer of the new node pointed by the* `modified tail` *points to the node pointed by the* `replaced tail`*, and its* `tag` *is one greater than the* `replaced tail's tag`.
- *The* `tail's tag` *equals the* `tag` *of the* `next` *pointer of the node it points to.*

Proof From Lemma 1, in the initial state the `tail's tag` equals the `tag` of the `next` pointer of the node it points to.

From the code, the `tail` can only be modified in the CAS in Line E07, and thus no other operation can change the `tail's` value. If an `enqueue` process successfully performed the CAS in Line E07, then, by the code, the `modified tail` points to the new node created in Line E02 and its `tag` is one greater than the `tag` of the `replaced tail`.

The `next` pointer of the new node created in Line E02 is written by the `enqueuing` process in Line E06, before the successful CAS in Line E07, but after the execution of Line E05. Line E06 sets the `next` pointer of this new node to point to the same node as pointed-to by the `replaced tail` while its `tag` is set to be one greater than the `tag` of the `replaced tail`.

From all the above, it follows that in any state of the queue, the `tail's tag` equals the `tag` of the `next` pointer of the node it points to. □

Lemma 3 *In the states following a true evaluation of the condition in Line D07, and until the dequeue method leaves the condition block in Line D22, the* `head` *and* `tail` *values read in the last D04 and D05 existed together in any state of the queue in which the the* `tail` *was read in D05.*

Proof The condition in Line D07 evaluates to true if the `head` read in Line D04 equals the `head` read in Line D07. That is, the `head` did not change in this interval. The `tail` is read in Line D05 within this interval, therefore the lemma follows. □

Lemma 4 *In the state in which a node is freed and returned to the pool of nodes in Line D15, the node is not in the queue.*

Proof From the code, a node is freed in Line D15, after a successful CAS on the `head` in Line D14. The node that is freed in Line D15 is the one pointed-to by the `replaced head`. From Definition 2, the node pointed-to by the `replaced head` is no longer in the queue after the successful CAS in Line D14. □

Lemma 5 *In any state, only the successful CAS operation in Line E07 can modify the* `tail` *pointer and only the successful CAS operation in Line D14 can modify the* `head` *pointer. In addition, in any state, the* `next` *pointer of a node* in the queue *never changes.*

Proof Examining the code of all methods, the only instruction that can write the `tail` is the CAS operation in Line E07, and the only instruction that can write the `head` is the CAS operation in Line D14. Also from the code, the `next` pointer of a node allocated in Line E02 is written in Line E06, before the CAS in Line E07. From the properties of the memory management module (Sect. 5.2) and from Lemma 4, this new node was not in the queue when it was allocated, and it is thus not in the queue when its `next` pointer is written (Line E06). By Definition 2, after the successful CAS in Line E07 the new node is in the queue. Examining the code of all the methods reveals that this is the only instruction that writes

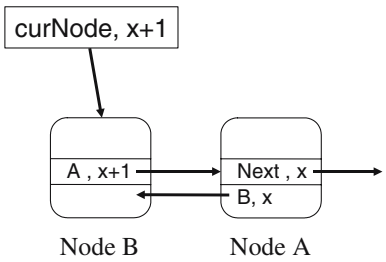

Fig. 19 A correct setting of the `prev` pointer between two nodes

the `next` pointer of any node, and thus the `next` pointer of a node in the queue is never changed. □

Lemma 6 *An unsuccessful dequeue method does not perform any CAS operations and does not change any next pointers.*

Proof By Definition 1 and Lemma 3, the linearization point of an unsuccessful `dequeue` method is the last read of the `tail` in Line D05, in which both the `head` and the `tail` co-existed in the state in which the null value was returned in Line D20. An unsuccessful dequeue method fails the condition in Line D08 and immediately returns null in Line D20. Thus, no CAS operations were performed and no `next` pointers were modified. □

Lemma 7 *In any state, if node A is the successor of node B and both A and B are in the queue, then the* `tag` *of the* `next` *pointer of node B is one greater then the* `tag` *of the* `next` *pointer of A.*

Proof Follows from Definition 3, and Lemmas 1, 2, and 5. □

In the following Lemmas we show that the "backwards" direction of the doubly-linked list is constructed correctly. That is, if the `tags` of the `prev` pointer and the `next` pointer in the same node are equal, then the `prev` pointer of that node points to its predecessor node — the node whose `next` pointer points to it. This is crucial to the correctness of our algorithm since the `dequeue` method decides which node to `dequeue` based on the order defined by the `prev` pointers. We will use these lemmas to prove in Theorem 1 that our queue is linearizable. A correct setting of the `prev` pointers is depicted in Fig. 19.

Lemma 8 *In any state, when* `curNode` *is set in Lines F03 or F07 to point to a node A that is in the queue, then its* `tag` *is set to equal the* `tag` *of the* `next` *pointer of A.*

Proof From the code, `curNode` is set to equal the tail passed to the `fixList` procedure in Line F03. The `fixList` procedure is called from the `dequeue` method in Line D10, and thus, by Lemma 3, the `tail` and `head` pointers passed to it co-existed in the state the `tail` was read in Line D05. By Lemma 2, the `tag` of this `tail` equals the `tag` of the `next` pointer of the node it points to. In Line F05 the `next` pointer of the node pointed by `curNode` is read. By Lemmas 2, 5, and 7, it follows that the `tag` of this `next` pointer is one less than the `tag` of `curNode`. In Line F07, when `curNode` is set, it is set to point the same node pointed by this `next` pointer and its `tag` is set to be one less than its current `tag`. Hence the Lemma follows. □

Lemma 9 *In any state, when the* `prev` *pointer of node A that is in the queue is written by the* `fixList` *procedure in Line F06, it is set to point to A's predecessor node, and its* `tag` *is set to equal A's* `next` *pointer tag, as they existed in the state before the last execution of Line F05.*

Proof The `fixList` procedure is called from the `dequeue` method in Line D10, and thus, by Lemma 3, the `tail` and `head` pointers passed to it co-existed in the state the `tail` was read in Line D05. The `fixList` execution traverses the nodes that were in the queue by the time it was called. For each node encountered in the traversal, pointed by `curNode`, the `fixList` procedure reads its `next` pointer (F05). Denote the node pointed by `curNode` as node *B* and its successor read in Line F05 as node *A*. By Lemma 2 and 8, for any node *B* pointed to by `curNode`, the `tag` of `curNode` equals the `tag` of the `next` pointer of *B*.

In Line F06, the `fixList` procedure writes the `prev` pointer of node *A*. The `prev` pointer is set to point to node *B* and its `tag` is set to be one less than the `tag` of `curNode`, as they existed in the last execution of Line F05. If by the time the `prev` pointer is written in Line F06 node *A* is still in the queue, then from Lemmas 5 and 7, the `tag` of the `prev` pointer of *A* written in Line F06 equals the `tag` of *A*'s `next` pointer. Note that by Definition 2 and Lemma 5, *A* is in the queue until the `head` is CASed from it to another node, and if *A* is in the queue then so is its predecessor *B*.

The `fixList` procedure traverses the nodes until it reaches the node pointed-to by the `head` passed to it or until this `head` is no longer equal to the `head` pointer of the queue (F04). Thus, the `fixList` stops only if it traversed all the nodes that were in the queue at the last read of Line D05, or if some `dequeue` process that read the same `head` as passed to the `fixList` procedure successfully performed the CAS in Line D14. Note that a change to the `tail` pointer does not affect the fixList procedure. □

Lemma 10 *In any state, when the* `prev` *pointer of node A that is in the queue is written by an* `enqueue` *process in Line E08, it is set to point to A's predecessor node, and its* `tag` *is set to equal A's* `next` *pointer tag, as they existed at the time the* `tail` *was CASed by this* `enqueue`.

Proof From the code, the `prev` pointer of a node can only be written by an `enqueue`ing process in Line E08, after

it succeeded in the CAS in Line E07. This successful enqueueing process can only write the prev pointer of the node pointed-to by the replaced tail it read in Line E05. Let *A* be the node pointed to by the replaced tail and denote the new node inserted to the queue by this enqueue process as node *B*.

Note that among all concurrent processes that read the same replaced tail, only the process that successfully CASed it can write *A*'s prev pointer. In Line E08 the prev pointer of node *A* is set to point to node *B* and its tag is set to equal the replaced tail's tag. Since by Lemma 2 the replaced tail's tag equals the tag of the next pointer of *A*, *A*'s prev pointer tag equals *A*'s next pointer tag. □

Lemma 11 *Consider two nodes A and B in the queue, where node A is the successor of node B. In any state in which A's* prev *pointer* tag *equals A's* next *pointer* tag, *A's* prev *pointer points to node B.*

Proof From the code, the prev pointer of node *A* is only modified in the following places:

- In Line E08, by an enqueue method after a successful CAS on the tail that inserted *A*'s predecessor to the queue.
- In Line F06, by the fixList procedure.

From Lemmas 9 and 10 the claim follows. □

Lemma 12 *In any state, the* head's tag *equals the* tag *of the* next *pointer of the node it points to* (*that is in the queue by Definition 2*), *and in the state immediately following a successful CAS on the* head *in Line D14, the* modified head *points to the predecessor of the node pointed-to by the* replaced head.

Proof By Definition 5, when the queue is initialized both the tag of the head and the tag of the next pointer of the dummy node are set to zero, therefore the claim holds in the initial empty state. Assume the claim holds in some state of the queue.

The head can only be CASed by a dequeueing process in Line D14. Denote the node pointed-to by the head read by a dequeueing process in the last execution of line D05 as node *A*. In Line D06 the dequeue process reads the prev pointer of *A*. In Lines D09, before the head can be CASed in Line D14, the process checks that the head's tag equals *A*'s prev pointer tag read in Line D06. If they are, then by Lemmas 5, 7, and 11, the prev pointer of *A* points to *A*'s predecessor, and the tag of the next pointer of the predecessor node is one greater than the tag of *A*'s next pointer.

The successful CAS in Line D14 modified the head to point to *A*'s predecessor with a tag which is one greater than the replaced head's tag. It thus equals the tag of the next pointer of the node it points to. □

Lemma 13 *If the* tail's tag *equals the* head's tag *as they were read by a* dequeue *method in the last executions of Lines D04 and D05, then both point to the same node.*

Proof From Definitions 2 and 5, and Lemma 1, this claim is true in the initial, empty state of the queue. From Lemma 3, the head and tail co-existed in the last execution of Line D05. From Lemma 2, the tail's tag equals the tag of the next pointer of the node it points to, and from Lemma 12 the head's tag also equals the tag of the next pointer of the node it points to. Thus, when the tail and head tags are equal, they must point to the same node. □

We now prove Theorem 1 which is the main theorem of the linearizability proof. It defines the order between the nodes in the queue, and establishes the relation between this order and the linearization order of the enqueue and dequeue methods.

Theorem 1 *The order induced by the* next *pointers of nodes in the queue from* tail *to* head, *is the inverse of the linearization order of the enqueue operations that inserted these nodes to the queue.*

Proof We prove the theorem by induction on the length of the execution, examining all system states and the enqueue and dequeue methods that can be applied in them. We start by proving that Theorem 1 holds in the empty state of the queue.

Lemma 14 *Theorem 1 holds in the* initial empty state *of the queue.*

Proof By Definitions 2 and 5 and Lemma 1, the initial state of the queue is the empty state. There is one dummy node in the queue and the tail and head pointers both point to it. Thus, no order is induced by the next pointers. Therefore Theorem 1 holds. □

We now show by induction that Theorem 1 holds in any system state. By Lemma 5, in any state of the queue, the only operations that can change the state of the queue (i.e. the nodes that are in the queue) are the CAS operations on the tail in Line E07 and on the head in Line D14. By Lemma 6, an unsuccessful dequeue method does not perform any CAS operations. Therefore, in this proof we only need to consider successful enqueue and dequeue methods, which are linearized at these CAS operations.

1. Assume that the system is in the *initial empty state* and that Theorem 1 holds true.

(a) From Lemma 1, in this state the `tail` equals the `head`. From the code, a `dequeue` operation in this state will fail the condition in Line D08 and thus will be unsuccessful. By Lemma 6, no CAS operations were performed and no `next` pointers were changed. Hence, there is no change in the state of the queue and Theorem 1 still holds.

(b) Consider the first successful CAS on the `tail` in Line E07. By Definition 2 and Lemma 2, after this successful CAS, the `modified tail` points to the new node created by the `enqueuing` process in Line E02, and this node is in the queue. By Lemmas 1 and 2, the `next` pointer of the new node points to the dummy node. By Lemma 5, this `next` pointer is not changed as long as this node is in the queue. Thus, the order of the nodes currently in the queue from `tail` to `head` as induced by the `next` pointers is first the new node and then the dummy node. Hence Theorem 1 holds.

2. Assume that in some system state the queue contains one or more non-dummy nodes and that Theorem 1 holds true. By Theorem 1, the nodes in the queue are ordered according to their `next` pointers and this order is the inverse of the linearization order of the `enqueue` operations that inserted these nodes.

(a) Consider a successful CAS in Line E07 in this state, performed by an `enqueue` operation. According to the properties of the memory management module described in Sect. 5.2 and by Lemma 4, the new node created by this `enqueue` operation in Line E02 is not in the queue before this successful CAS. By the induction assumption, before this successful CAS, the first node in the order induced by the `next` pointers of nodes in the queue was pointed-to by the `replaced tail` and was inserted by the last linearized `enqueue` operation.
By Definition 2 and Lemma 2, after this successful CAS, the `modified tail` points to the new node, and the `next` pointer of the new node points to the same node pointed-to by the `replaced tail`.
From all the above it follows that (1) this linearized `enqueue` operation is now the last in the linearization order, (2) the new node it inserted is now the first in the order of nodes in the queue, and (3) the `next` pointer of this node points to the first node in the order of nodes that were in the queue before this operation (before the linearization point of this operation).
Hence Theorem 1 still holds.

(b) Consider a successful CAS on the `head` in Line D14 in this state, performed by a successful `dequeue` operation. Denote the node pointed-to by the `replaced head` as A and the node pointed by the `modified head` as node B.
By the induction assumption, before this successful CAS, A is the last ("oldest") node in the order induced by the `next` pointers of nodes in the queue. The `enqueue` operation that inserted A is linearized before all other `enqueue` operations that inserted nodes that are in the queue in this state.
By Definition 2 and Lemma 12, after this successful CAS the `modified head` points to predecessor of node A (denoted B) and node A is no longer in the queue. Since the `modified head` points to B, B is now considered a dummy node.
From all the above it follows that: (1) this linearized `dequeue` operation removed the last node in the order induced by the `next` pointers of nodes in the queue at the linearization time, and (2) the `modified head` points to the current last node in the order induced by the `next` pointers of nodes in the queue immediately after the `dequeue`'s linearization point.
If this CAS causes the `head`'s `tag` to be equal to the `tail`'s `tag`, then by Lemma 13, the `tail` also points to node B. By the induction assumption and by Definition 2, the queue is now empty. In this empty state, the `tail` and `head` point to the same node and their `tags`, as well as B's `next` pointer's `tag`, are equal.
Hence Theorem 1 holds in the case in which there are still nodes in the queue after the CAS as well as in the case in which this CAS causes the queue to become empty. □

Corollary 1 *The concurrent FIFO queue algorithm is linearizable to the sequential FIFO queue.*

Proof From Theorem 1 it follows that in any state, all the nodes in the queue are ordered by their next pointers. By Definition 1, the linearization point of a dequeue operation is the CAS on the `head` in Line D14. By these, and by Lemma 12 it must be that in each state, a linearized successful `dequeue` operation deletes the "oldest" node in the queue in this state. Thus, the order of values inserted to the queue by the linearized `enqueue` operations is the same as the order of values deleted by the linearized `dequeue` operations. □

5.5 Lock-freedom proof

In this section we prove that our concurrent implementation is lock-free. To do so we need to show that if one process fails in executing an `enqueue` or a `dequeue`, then other processes must be succeeding infinitely often, and the system as a whole is making progress.

Lemma 15 *If an* `enqueue` *method failed to insert its node in Line E07, then another* `enqueue` *method succeeded.*

Proof By Definition 2 and Lemma 2, a node is inserted to the queue at the successful CAS in line E07 and by Lemma 5 this CAS is the only place in which the `tail` pointer is modified. If an `enqueue` method failed in E07, then the value of the `tail` is no longer the same as read by this method in Line E05. The value of the `tail` can only be changed if another `enqueue` method succeeded in performing the CAS in Line E07, and thus another node was inserted to the queue. □

Lemma 16 *Any execution of* `fixList` *eventually terminates.*

Proof The `fixList` procedure exits its loop if it either reaches the node pointed-to by the `head` or if the `head` is changed. The `fixList` procedure is called with `head` and `tail` values that, by Lemma 3, existed in the state in which the `tail` value was read. By Theorem 1 and Lemmas 5 and 8, the nodes in the queue are ordered by their `next` pointers, this order never changes, and the fixList traverses the nodes in the queue by the `next` pointers order. Therefore the number of nodes traversed by `fixList` cannot increase from their number in the state in which the `fixList` was called. By the code, any successful `dequeue` operation will change the `head` pointer. Thus, `fixList` may stop before it accessed all the nodes that existed in the queue in the state from which it was called. Hence, within a finite number of steps, any `fixList` execution will end.

Lemma 17 *If a* `dequeue` *method failed to delete a node when executing the CAS in Line D14, then another* `dequeue` *method must have succeeded to do so.*

Proof In order for the `dequeue` operation to be lock-free, the `fixList` procedure called from the `dequeue` operation in Line D11 must eventually end, and the `head`'s `tag` must eventually be equal to the `prev` pointer's `tag` of the node it points to, denoted *A*, so that the condition in Line D09 will fail and a value can be returned. Lemma 16 proves that the `fixList` procedure eventually ends. Lemmas 9, 10 and 11 prove that any operation that sets the `prev` pointer of a node in the queue, sets it such that its `tag` equals the `next` pointer `tag` in the same node. By Lemma 12, when *A*'s `prev` pointer is set, its `tag` equals the `head`'s `tag`.

Note that all concurrent processes that try to modify *A*'s `prev` pointer while *A* is in the queue, will try to write the same value. These processes can be:

- A slow `enqueue` process that inserted *A*'s predecessor node (Line E08).
- Calls to the `fixList` procedure made by other dequeuing processes.

By Lemma 10, if *A* is in the queue then this `enqueue` operation sets the `prev` pointer of *A* such that its `tag` equals *A*'s `next` pointer `tag` and thus, by Lemma 12, equals the `head's tag`.

If *A* is still in the queue and pointed-to by the `head`, all concurrent `dequeue` operations will read the same value of the `head` in Line D04, and thus will call the `fixList` procedure with the same `head` value. The exact value of the `tail` is not important, as long as it is possible to reach *A* form the `tail`, and by Theorem'1 this must be possible.

Since by Lemma 16 eventually all `fixList` procedures called by the concurrent `dequeue` operations with the same `head` value will end, then eventually *A*'s `prev` pointer `tag` will equal the `tag` of the `head`.

From Lemma 5, the `head` can only be modified in the CAS in Line D14. Thus when *A*'s `prev` pointer `tag` equals the `head's tag`, then if a dequeuing process failed to CAS the `head` in Line D14 it must be that the `head` was changed by another successful `dequeuing` process.

From the Lemmas 15, 16, and 17 it follows that:

Corollary 2 *The concurrent FIFO queue algorithm is lock-free.*

6 Code in the Java programming language of our new algorithm

This section presents a short version of the Java code for the queue data structure and the enqueue and dequeue methods. The Java Concurrency package implements the MS-queue algorithm in the class ConcurrentLinkedQueue. We followed this implementation and modified it to support our new algorithm. Following the notation used in the Java programming language, the enqueue operation is called `offer` and the dequeue operation is called `poll`.

```
public class OptimisticLinkedQueue<E> extends AbstractQueue<E>
   implements Queue<E>, java.io.Serializable {
    private static class Node<E> {
        private volatile E item;
        private volatile Node<E> next;
        private volatile Node<E> prev;

        Node(E x) { item = x; next = null; prev = null; }

        Node(E x, Node<E> n) { item = x; next = n; prev = null; }

        E getItem() {
            return item;
        }
        void setItem(E val) {
            this.item = val;
        }
        Node<E> getNext() {
            return next;
        }
        void setNext(Node<E> val) {
            next = val;
        }
        Node<E> getPrev() {
            return prev;
        }
        void setPrev(Node<E> val) {
```

```
        prev =  val;
    }
}
private static final
    AtomicReferenceFieldUpdater<OptimisticLinkedQueue, Node>
    tailUpdater =
    AtomicReferenceFieldUpdater.newUpdater
    (OptimisticLinkedQueue.class, Node.class, "tail");
private static final
    AtomicReferenceFieldUpdater<OptimisticLinkedQueue, Node>
    headUpdater =
    AtomicReferenceFieldUpdater.newUpdater
    (OptimisticLinkedQueue.class,  Node.class, "head");
private boolean casTail(Node<E> cmp, Node<E> val) {
    return tailUpdater.compareAndSet(this, cmp, val);
}

private boolean casHead(Node<E> cmp, Node<E> val) {
    return headUpdater.compareAndSet(this, cmp, val);
}
/**
 * Pointer to the head node, initialized to a dummy node. The first
 * actual node is at head.getPrev().
 */
private transient volatile Node<E> head = new Node<E>(null, null);

/** Pointer to last node on list **/
private transient volatile Node<E> tail = head;

/**
 * Creates a <tt>ConcurrentLinkedQueue</tt> that is initially empty.
  */
 public OptimisticLinkedQueue() {}

 /**
  * Enqueues the specified element at the tail of this queue.
  */
 public boolean offer(E e) {
     if (e == null) throw new NullPointerException();
     Node<E> n = new Node<E>(e, null);
     for (;;) {
         Node<E> t = tail;
         n.setNext(t);
         if (casTail(t, n)) {
            t.setPrev(n);
            return true;
         }
     }
 }

 /**
  * Dequeues an element from the queue. After a successful
  casHead, the prev and next pointers of the dequeued node are
  set to null to allow garbage collection.
  */
 public E poll() {
     for (;;) {
         Node<E> h = head;
         Node<E> t = tail;
         Node<E> first = h.getPrev();
         if (h == head) {
             if (h != t) {
                 if (first == null){
                     fixList(t,h);
                     continue;
                 }
                 E item = first.getItem();
                 if (casHead(h,first)) {
                     h.setNext(null);
                     h.setPrev(null);
                     return item;
                 }
             }
             else
                 return null;
         }
     }
 }
 /**
  * Fixing the backwords pointers when needed
  */
 private void fixList(Node<E> t, Node<E> h){
     Node<E> curNodeNext;
     Node<E> curNode = t;
     while (h == this.head && curNode != h){
         curNodeNext = curNode.getNext();
         curNodeNext.setPrev(curNode);
         curNode = curNode.getNext();
     }
 }
}
```

7 Related work

There are many linearizable [9] concurrent FIFO queue algorithms in the literature. A survey can be found in [22]. These algorithms can be categorized according to whether they are blocking or non-blocking; and whether they are based on static allocation of a circular array, or on a dynamic allocation of nodes in a linked-list. Our linearizable algorithm belongs to the group of non-blocking dynamically allocated FIFO queues.

In [5], Gottlieb, Lubachevsky, and Rudolph present a statically-allocated blocking FIFO queue that does not use locks. A process must wait until its turn to insert/delete a value to/from the queue, and a faster process can be delayed by a slower one indefinitely.

Treiber [29] presents several dynamically-allocated implementations of FIFO queues that are non-blocking. The drawback of his solutions is that they require a dequeue process to traverse the entire list in order to delete a node.

Stone [28] presents an algorithm that requires only one CAS per enqueue or dequeue operation. When a node is enqueued, the tail is set to point to the new node, and the *link* of the previous node (pointed-to by the old tail), is set to point to the new node. To dequeue a node, a process follows the chain of *links*, and tries to CAS the head accordingly. However, this implementation is blocking since an enqueuer can block dequeuers if its update of the previous link is delayed. It is also non-linearizable, since a slow enqueuer can observe an empty queue, even though a faster enqueuer already inserted a node into the queue.

In [24], Prakash, Lee, and Johnson present a FIFO queue algorithm that is non-blocking, uses two CAS operations for an enqueue and one for a dequeue. It requires a snapshot of the current state of the queue on two variables in both enqueue and dequeue operations. To be non-blocking, in cases when more than one CAS is needed, an intermediate state is detected, and faster processes can complete the work of slower ones.

Valois [32] presents two new non-blocking algorithms, one using dynamic allocation and the other using static allocation. The dynamic allocation algorithm eliminates the need for the snapshots used in [24] and solves the synchronization and contention problems when the queue becomes empty by setting the head to point to the last node dequeued and not to the node currently at the front of the queue. As in [24], this algorithm also requires two CAS operations to perform an enqueue, and one CAS operation to perform a dequeue. In this algorithm, the tail can lag behind the head, implying that

dequeued nodes cannot be simply freed, rather, a reference counting mechanism is needed. As described by Michael and Scott [19], this mechanism causes fairly quick memory consumption, hence it is impractical. Valois also presents two other alternatives: the first one in which the tail does not always point to the last or second to last node, and the second one in which only a successful enqueue process updates the tail. The first alternative causes processes to spend time on traversing the list. The second alternative causes the algorithm to become blocking. The static allocation array implementation described is simple but requires unaligned CAS operations and is thus impractical on real-world multiprocessor machines.

The linearizable shared FIFO queue presented by Michael and Scott in [19] is the most popular lock-free (and hence non-blocking) dynamically-allocated queue known in the literature. It is based on a single-linked list, where nodes are inserted at the tail and deleted from the head. An enqueue operation requires two CAS operations, and a dequeue requires one CAS operation (as in [24,32]). As in the Prakash, Lee, and Johnson algorithm, this algorithm uses a snapshot. However, it is a simpler one, applied only to one variable. As in Valois algorithm, it requires a dummy node that is always kept at the head of the queue. Thus, the value that is dequeued from the queue is always the one in the node followed by the node pointed-to by the head.

Tsigas and Zhang present in [30] another implementation of a statically-allocated non-blocking queue. By letting the tail and head lag at most m nodes behind the actual last/first node, each of their enqueue operations requires $1+1/m$ CAS operations on average and a dequeue requires only one CAS operation.

Scherer and Scott [26] present a dual-data-structure version of a FIFO queue, in which performance improvements are derived by splitting the dequeue operation into a request operation and a follow-up operation, which are linearized separately, while preserving the correct FIFO order. This dual-queue presents different semantics since dequeue operations wait for the queue to become non-empty and the queue determines the order in which pending requests are granted.

Blocking FIFO queues are discussed in several papers regarding spin locks. The most well known are the MCS-lock [17] and CLH-lock [4,16] algorithms. Though these algorithms are blocking, there is an interesting point in which our algorithm is similar to the CLH-lock algorithm. All queue algorithms involve two operations — setting a tail pointer and setting a node-to-node pointer. In the CLH-lock, as in our new algorithm, each thread reads the shared tail pointer (or lock pointer) without modifying it, and then "optimistically" points its private node's pointer to the proposed predecessor. Only then does it compete with others to insert its node into the queue, making it public. The single CAS operation is needed only for setting the tail pointer and not for the node-to-node pointer which is set privately. On the other hand, in the MS-queue, the new node is first inserted into the queue and made public (as in the MCS-lock), and only then is the second pointer updated. This implies, for the MS-queue, that a strong synchronization operation must be used for setting both the a tail pointer and the node-to-node pointer.

All dynamically-allocated non-blocking algorithms face the problem of allocating and releasing nodes, and with it the ABA problem [7,20]. Valois [32] has used reference counting to solve the problem. Herlihy et al. [7] and Michael [20] present generic memory management mechanisms that can be applied in the implementation of many non-blocking data structures. In [6], Herlihy et al. present two ways in which the ROP technique can be applied to the MS-queue.

In our algorithm we optimistically build a doubly-linked list. The design of such doubly-linked lists is discussed in the literature in the context of double-ended queues. In [2], a double-ended queue implementation is presented. This implementation uses a DCAS operation, currently not supported on real-world multiprocessor machines.

In [8], an obstruction-free double-ended queue is presented, based only on CAS operations. The obstruction-free requirement (a non-blocking condition that is weaker than lock-freedom) optimistically assumes that each process will have enough time in which it executes in isolation, and has a mechanism to overcome interference if it occurs. By weakening the non-blocking progress requirement, they are able to present a simple and practical implementation of double-ended queue.

Acknowledgements We wish to thank Doug Lea for a helpful discussion. We would also like to thank the anonymous *DISC2004* referees for their many helpful comments.

References

1. Afek, Y., Attiya, H., Dolev, D., Gafni, E., Merritt, M., Shavit, N.: Atomic snapshots of shared memory. J. ACM. **40**(4), 873–890 (1993)
2. Agesen, O., Detlefs, D., Flood, C., Garthwaite, A., Martin, P., Moir, M., Shavit, N., Steele, G.: DCAS-based concurrent deques. Theo. Comput. Syst. **35**(3), 349–386 (2002)
3. Cormen, T., Leiserson, C., Rivest, R., Stein, C.: Introduction to Algorithms, 2nd edn. MIT Press, Cambridge (2001)
4. Craig, T.: Building FIFO and priority-queueing spin locks from atomic swap. Technical Report TR 93-02-02, Department of Computer Science, University of Washington (1993)
5. Gottlieb, A., Lubachevsky, B.D., Rudolph, L.: Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. ACM Trans. Progr. Lang. Syst. **5**(2), 164–189 (1983)
6. Herlihy, M., Luchangco, V., Martin, P., Moir, M.: Dynamic-sized lock-free data structures. In: Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing, pp. 131–131. ACM, New York (2002)

7. 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, vol. 2508, pp. 339–353. Springer, Heidelberg (2002). A improved version of this paper is in preparation for journal submission
8. 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, pp. 522–529. IEEE (2003)
9. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Progr. Lang. Syst. (TOPLAS) **12**(3), 463–492 (1990)
10. Hwang, K., Briggs, F.A.: Computer Architecture and Parallel Processing. McGraw-Hill, New York (1990)
11. Intel. Pentium Processor Family User's Manual: vol 3. In: Architecture and Programming Manual, 1994
12. Lamport, L.: Specifying concurrent program modules. ACM Trans. Progr. Lang. Syst. **5**(2), 190–222 (1983)
13. Lea, D.: The java concurrency package (JSR-166). http://gee.cs.oswego.edu/dl/concurrency-interest/index.html
14. Lea, D.: The java.util.concurrent synchronizer framework. In: Workshop on Concurrency and Synchronization in Java Programs, pp. 1–9 (2004)
15. Luchangco, V., Moir, M., Shavit, N.: On the uncontended complexity of consensus. In: Proceedings of the 17th International Conference on Distributed Computing, pp. 45–59 (2003)
16. Magnussen, P., Landin, A., Hagersten, E.: Queue locks on cache coherent multiprocessors. In: Proceedings of the 8th International Symposium on Parallel Processing (IPPS), pp. 165–171. IEEE Computer Society (1994)
17. Mellor-Crummey, J., Scott, M.: Algorithms for scalable synchronization on shared—memory multiprocessors. ACM Trans. Comput. Syst. **9**(1), 21–65 (1991)
18. Mellor-Crummey, J.M.: Concurrent queues: Practical fetch-and-ϕ algorithms. Technical Report 229, University of Rochester (1987)
19. Michael, M., Scott, M.: Nonblocking algorithms and preemption-safe locking on multiprogrammed shared—memory multiprocessors. J. Parallel Distrib. Comput. **51**(1), 1–26 (1998)
20. Michael, M.M.: Hazard pointers: safe memory reclamation for lock-free objects. IEEE Trans. Parallel Distrib. Syst. **15**(6), 491–504 (2004)
21. Moir, M.: Practical implementations of non-blocking synchronization primitives. In: Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing, pp. 219–228 (1997)
22. Moir, M., Shavit, N.: Chapter 47 – Concurrent Data Structures – Handbook of Data Structures and Applications, 1st edn. Chapman and Hall/CRC, London (2004)
23. 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)
24. Prakash, S., Lee, Y.-H., Johnson, T.: A non-blocking algorithm for shared queues using compare-and-swap. IEEE Trans. Comput. **43**(5), 548–559 (1994)
25. Rajwar, R., Goodman, J.: Speculative lock elision: enabling highly concurrent multithreaded execution. In: Proceedings of the 34th Annual International Symposium on Microarchitecture, pp. 294–305 (2001)
26. Scherer, W.N., Scott, M.L.: Nonblocking concurrent data structures with condition synchronization. In: Proceedings of the 18th International Symposium on Distributed Computing, pp. 174–187. Springer, Berlin (2004)
27. Stone, H.S.: High-performance Computer Architecture. Addison-Wesley Longman, Reading (1987)
28. Stone, J.: A simple and correct shared-queue algorithm using compare-and-swap. In:Proceedings of the 1990 Conference on Supercomputing, pp. 495–504. IEEE Computer Society
29. Treiber, R.K.: Systems programming: coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center (1986)
30. 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, pp. 134–143. ACM, New york (2001)
31. Valois, J.: Implementing lock-free queues. In: Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems, pp. 64–69 (1994)
32. Valois, J.D.: Lock-free linked lists using compare-and-swap. In: Symposium on Principles of Distributed Computing, pp. 214–222 (1995)
33. Weaver, D., Germond, T.: The SPARC Architecture Manual Version 9. Prentice Hall, Englewood Cliffs (1994)