The Baskets Queue

Moshe Hoffman[1], Ori Shalev[1], and Nir Shavit[1,2]

[1] The School of Computer Science, Tel Aviv University, Israel
[2] Sun Microsystems Laboratories
moshe.hoffman@gmail.com, orish@post.tau.ac.il,
shanir@post.tau.ac.il

Abstract. FIFO Queues have over the years been the subject of significant research. Such queues are used as buffers both in a variety of applications, and in recent years as a key tool in buffering data in high speed communication networks.

Overall, the most popular dynamic-memory lock-free FIFO queue algorithm in the literature remains the MS-queue algorithm of Michael and Scott. Unfortunately, this algorithm, as well as many others, offers no more parallelism than that provided by allowing concurrent accesses to the head and tail. In this paper we present the Baskets Queue - a new, highly concurrent lock-free linearizable dynamic memory FIFO queue. The Baskets Queue introduces a new form of parallelism among enqueue operations that creates *baskets* of mixed-order items instead of the standard totally ordered list. The operations in different baskets can be executed in parallel. Surprisingly however, the end result is a linearizable FIFO queue, and in fact, we show that a basket queue based on the MS-queue outperforms the original MS-queue algorithm in various benchmarks.

Keywords: CAS, Compare and Swap, Concurrent Data Structures, FIFO queue, Lock-free, Non-blocking, Synchronization.

1 Introduction

First-in-first-out (FIFO) queues are among the most basic and widely used concurrent data structures. They have been studied in a static memory setting [1,2] and in a dynamic one [3,4,5,6,7,8,9,10,11,12,13,14,15,2]. The classical 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 best known concurrent FIFO queue implementation is the lock-free queue of Michael and Scott [16] which is included in the Java^{TM} Concurrency Package [17]. Its key feature is that it maintains, in a lock-free manner, a FIFO ordered list that can be accessed disjointly through head and tail pointers. This allows enqueue operations to execute in parallel with dequeue operations.

A later article by Ladan-Mozes and Shavit [7] presented the *optimistic queue* that in many cases performs better than the MS-queue algorithm. The optimistic doubly-linked list reduces the number of compare-and-swap (CAS) operations necessary to perform an enqueue and replaces them with simple stores. However, neither algorithm allows more parallelism then that allowed by the disjoint head and tail.

E. Tovar, P. Tsigas, and H. Fouchal (Eds.): OPODIS 2007, LNCS 4878, pp. 401–414, 2007.

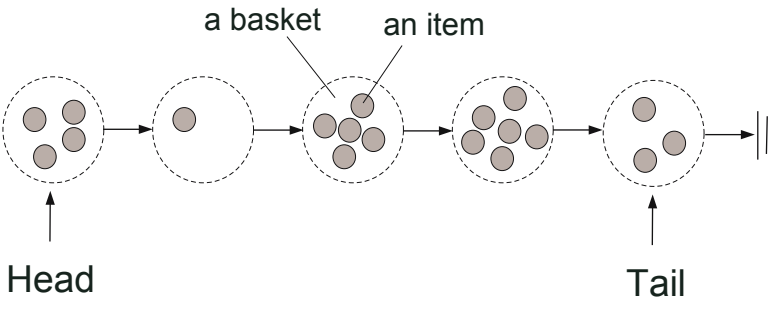

Fig. 1. The abstract Baskets Queue

In an attempt to add more parallelism, Moir et. al [18] showed how one could use elimination as a back-off scheme to allow pairs of concurrent enqueue and dequeue operations to exchange values without accessing the shared queue itself. Unfortunately, in order to keep the correct FIFO queue semantics, the enqueue operation cannot be eliminated unless all previous inserted nodes have been dequeued. Thus, the *elimination backoff queue* is practical only for very short queues.

In this paper we present a new approach that allows added parallelism in the design of concurrent shared queues. Our approach, which we apply to the MS-queue [16], can also be applied to the optimistic queue [7]. In our new "basket" approach, instead of the traditional ordered list of nodes, the queue consists of an ordered list of groups of nodes (baskets). The order of nodes in each basket need not be specified, and in fact, it is easiest to maintain them in LIFO order. Nevertheless, we prove that the end result is a linearizable FIFO queue. The benefit of the basket technique is that, with little overhead, it introduces a new form of parallelism among enqueue operations by allowing insertions into the different baskets to take place in parallel.

1.1 The Baskets Queue

Linearizability was introduced by Herlihy and Wing [4] as a correctness condition for concurrent objects. For a FIFO queue, an execution history is linearizable if we can pick a point within each enqueue or dequeue operation's execution interval so that the sequential history defined by these points maintains the FIFO order.

We notice that the definition of linearizability allows overlapping operations to be reordered arbitrarily. This observation leads to the key idea behind our algorithm: a group of overlapping enqueue operations can be enqueued onto our queue as one group (basket), without the need to specify the order between the nodes. Due to this fact, nodes in the same basket can be dequeued in any order, as the order of enqueue operations can be "fixed" to meet the dequeue operations order. Moreover, nodes from different groups may be inserted in parallel.

A concise abstraction of the new queue is a FIFO-ordered list of baskets where each basket contains one or more nodes (see Fig. 1). The baskets fulfill the following basic rules:

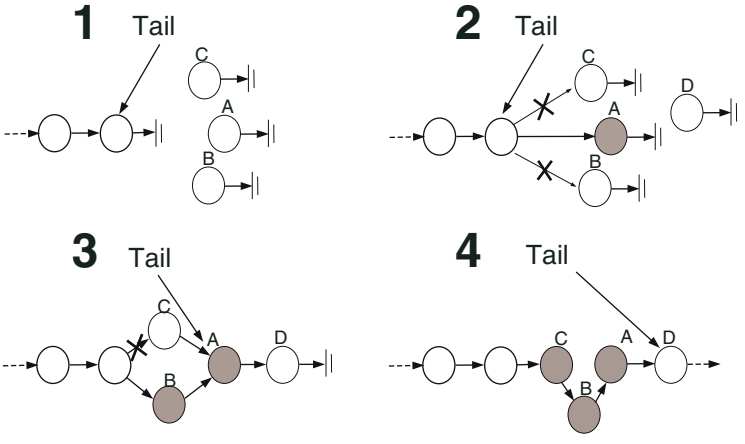

Fig. 2. (1) Each thread checks that the tail-node's next field is null, and tries to atomically change it to point to its new node's address. (2) Thread A succeeds to enqueue the node. Threads B and C fail on the same CAS operation, hence both of them will retry to insert into the basket. (3) Thread B was the first to succeed to enqueue, at the same time thread D calls the enqueue operation, and finishes successfully to enqueue onto the tail. (4) thread C finishes successfully.

1. Each basket has a time interval in which all its nodes' enqueue operations overlap.
2. The baskets are ordered by the order of their respective time intervals.
3. For each basket, its nodes' dequeue operations occur after its time interval.
4. The dequeue operations are performed according to the order of baskets.

Two properties define the FIFO order of nodes:

1. The order of nodes in a basket is not specified.
2. The order of nodes in different baskets is the FIFO-order of their respective baskets.

The basic idea behind these rules is that setting the linearization points of enqueue operations that share an interval according to the order of their respective dequeues, yields a linearizable FIFO-queue.

How do we detect which enqueue operations overlap, and can therefore fall into the same basket? The answer is that in algorithms such as the MS-queue or optimistic queue, threads enqueue items by applying a Compare-and-swap (CAS) operation to the queue's tail pointer, and all the threads that fail on a particular CAS operation (and also the winner of that CAS) overlap in time. In particular, they share the time interval of the CAS operation itself. Hence, all the threads that fail to CAS on the tail-node of the queue may be inserted into the same basket. By integrating the basket-mechanism as the back-off mechanism, the time usually spent on backing-off before trying to link onto the new tail, can now be utilized to insert the failed operations into the basket, allowing enqueues to complete sooner. In the meantime, the next successful CAS operations by enqueues allow new baskets to be formed down the list, and these can be filled concurrently. Moreover, the failed operations don't retry their link attempt on the new tail, lowering the overall contention on it. As we will show, this leads to a queue

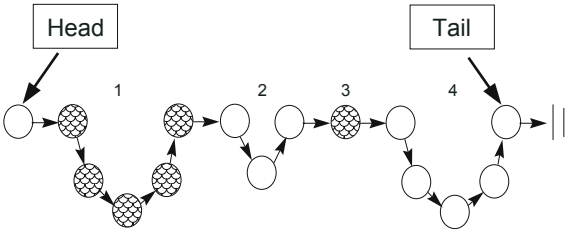

Fig. 3. A queue composed of 4 baskets

algorithm that unlike all former concurrent queue algorithms requires virtually no tuning of the backoff mechanisms to reduce contention, making our algorithm an attractive out-of-the-box queue.

In order to enqueue, just as in MS-Queue, a thread first tries to link the new node to the last node. If it failed to do so, then another thread has already succeeded. Thus it tries to insert the new node into the new basket that was created by the winner thread (see Fig. 2). To dequeue a node, a thread first reads the head of the queue to obtain the oldest basket. It may then dequeue any node in the oldest basket.

As we noted earlier, the implementation of the Baskets Queue we present here is based on Michael and Scott's MS-queue. Our algorithm maintains a linked list of nodes logically divided into baskets (see Fig. 3). Although, as the reader will see, in our implementation the baskets have a stack-like behavior, any concurrent pool object that supports the `add` and the `remove` operations, can serve as a basket. The advantage of such objects is that they can deliver more scalability than the stack-like baskets.

1.2 Performance

We compared our new lock-free queue algorithm to the lock-free MS-queue of Michael and Scott [16], and to the Optimistic-Queue by Ladan-Mozes and Shavit [7]. The algorithms were implemented in the C programming language and were executed on a 16 processors Sun Fire 6800 running the Solaris 9 operating system.

As our empirical results show, the new algorithm scales better under high contention due to the simultaneous successful enqueue operations. We believe that as the number of processors running the basket queue increases, it will be possible to replace the stack-like baskets with more scalable data-structures based on diffracting-trees [19] or counting-networks [20], that will make the Baskets Queue even faster.

2 The Baskets Queue

We now describe our algorithm in detail. Since we employ CAS operations in our algorithm, ABA issues arise [16,15]. In Section 2.2, we describe the `enqueue` and `dequeue` operations ignoring ABA issues. The tagging mechanism we added to overcome the ABA problem is explained in Section A. The code in this section includes this tagging mechanism.

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

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

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

Fig. 4. Types, structures and initialization

Although the nodes of the same basket need not be ordered, we insert and remove them in a stack-like manner, one by one. It is a subject for further research to determine if it feasible to exploit weaker orders to make the queue more scalable.

2.1 Data Structures

Just as in the MS-queue, our queue is implemented as a linked list of nodes with `head` and `tail` pointers. The `tail` points either to a node in the last basket, or in the second to last basket. In contrast to the MS-queue, we use pointer marking [10] to logically delete nodes. The queue's `head` always points to a dummy node, which might be followed by a sequence of logically deleted (marked) nodes.

2.2 The Baskets Queue Operations

The FIFO queue supports two operations: `enqueue` and `dequeue`. The `enqueue` method inserts a value into the queue and the `dequeue` method deletes the oldest value from the queue.

To enqueue a new node into the list, the thread first reads the current `tail`. If the `tail` is the last node (E07) it tries to atomically link the new node to the last node (E09). If the CAS operation succeeded the node was enqueued successfully, and the thread tries to point the queue's `tail` to the new node (E10), and then returns. However, if the thread failed to atomically swap the Null value, it means that the thread overlaps in time with the winner of the CAS operation. Thus, the thread tries to insert the new node to the basket (E12-E18). It re-reads the `next` pointer that points to the first node in the basket, and as long as no node in the basket has been deleted (E13), it tries to insert the node at the same list position. If the `tail` did not point to the last node, the last node is searched (E20-E21), and the queue's `tail` is fixed.

To prevent a late enqueuer from inserting its new node behind the queue's `head`, a node is dequeued by setting the `deleted` bit of its pointer so that a new node can only

```
E01: nd = new_node()
E02: nd->value = val
E03: repeat:
E04:    tail = Q->tail
E05:    next = tail.ptr->next
E06:    if (tail == Q->tail)):
E07:       if (next.ptr == NULL):
E08:          nd->next = <NULL, 0, tail.tag+2>
E09:          if CAS(&tail.ptr->next, next, <nd, 0, tail.tag+1>):
E10:             CAS(&Q->tail, tail, <nd, 0, tail.tag+1>
E11:             return True
E12:          next = tail.ptr->next
E13:          while((next.tag==tail.tag+1) and (not next.deleted)):
E14:             backoff_scheme()
E15:             nd->next = next
E16:             if CAS(&tail.ptr->next, next, <nd, 0, tail.tag+1>):
E17:                return True
E18:             next = tail.ptr->next;
E19:       else:
E20:          while ((next.ptr->next.ptr != NULL) and (Q->tail==tail)):
E21:             next = next.ptr->next;
E22:          CAS(&Q->tail, tail, <next.ptr, 0, tail.tag+1>)
```

Fig. 5. The `enqueue` operation

be inserted adjacently to another unmarked node. As the queue's `head` is only required as a hint to the next unmarked node, the lazy update approach of Tsigas and Zhang [1] can be used to reduce the number of CAS operations needed to update the `head`.

To dequeue a node, a thread reads the current state of the queue (D01-D04) and rechecks it for consistency (D05). If the `head` and `tail` of the list point to the same node (D06), then either the list is empty (D07) or the `tail` lags. In the latter case, the last node is searched (D09-D10) and the `tail` is updated(D11). If the `head` and the `tail` point to different nodes, then the algorithm searches for the first unmarked node between the `head` and the `tail` (D15-D18). If a non-deleted node is found, its value is first read (D24) before trying to logically delete it (D25). If the deletion succeeded the dequeue is completed. Before returning, if the deleted node is far enough from the `head` (D26), the `free_chain` method is performed (D27). If while searching for a non-deleted node the thread reached the `tail` (D21) the queue's `head` is updated (D22). See Fig. 7 for an illustration.

The `free_chain` procedure tries to update the queue's `head` (F01). If it successful, it is safe to reclaim the deleted nodes between the old and the new head (F02-F05).

3 Performance

We compared the performance of our FIFO queue to the best performing dynamic memory FIFO-queue algorithms. The algorithms were compiled in the C programming language with Sun's "CC" compiler 5.8 with the flags "-XO3 -xarch=v8plusa". The

```
const MAX_HOPS = 3 # constant

data_type dequeue(queue_t* Q)

D01: repeat
D02:    head = Q->headf
D03:    tail = Q->tail
D04:    next = head.ptr->next
D05:    if (head == Q->head):
D06:       if (head.ptr == tail.ptr)
D07:          if (next.ptr == NULL):
D08:             return 'empty'
D09:          while ((next.ptr->next.ptr != NULL) and (Q->tail==tail)):
D10:             next = next.ptr->next;
D11:          CAS(&Q->tail, tail, <next.ptr, 0, tail.tag+1)
D12:       else:
D13:          iter = head
D14:          hops = 0
D15:          while ((next.deleted and iter.ptr != tail.ptr) and (Q->head==head)):
D16:             iter = next
D17:             next = iter.ptr->next
D18:             hops++
D19:          if (Q->head != head):
D20:             continue;
D21:          elif (iter.ptr == tail.ptr):
D22:             free_chain(Q, head, iter)
D23:          else:
D24:             value = next.ptr->value
D25:             if CAS(&iter.ptr->next, next, <next.ptr, 1, next.tag+1>):
D26:                if (hops >= MAX_HOPS):
D27:                   free_chain(Q, head, next)
D28:                return value
D29:             backoff-scheme()
```

Fig. 6. The `dequeue` operation

different benchmarks were executed on a 16 processor Sun Fire™ 6800 running the Solaris™ 9 operating system.

3.1 The Benchmarked Algorithms

We compared our FIFO-queue algorithm to the lock-free queue of Michael and Scott [16], and to the Optimistic Queue of Ladan-Mozes and Shavit [7]. To expose the possible effects of our use of logical deletions, a variation of the MS-Queue with logical deletions was added as a control. The set of compared queue implementations was:

1. Baskets Queue - the new algorithm implementation.
2. Optimistic Queue - the pre-backoff version of the Optimistic FIFO-queue.
3. MS-queue - the lock-free version of the Michael and Scott's queue.
4. MS-queue lazy head - This is a variation of MS-Queue where dequeues are performed by logically deleting the dequeued node. Therefore, following Tsigas and Zhang's technique [1], the queue's head may be updated only once for several dequeues.

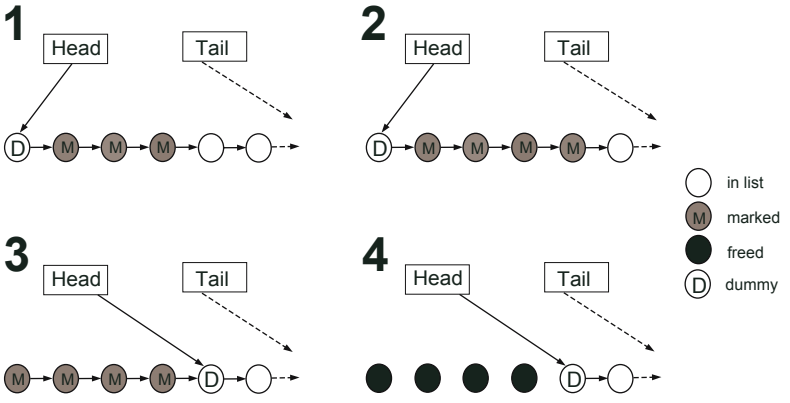

Fig. 7. (1) three nodes are logically deleted. (2) the first non-deleted node is deleted (3) the head is advanced (4) the chain of deleted nodes can be reclaimed.

```
void free_chain(queue_t* q, pointer_t head, pointer_t new_head)

F01: if CAS(&Q->head, head, <new_head.ptr, 0, head.tag+1>):
F02:    while (head.ptr != new_head.ptr):
F03:       next = head.ptr->next
F04:       reclaim_node(head.ptr)
F05:       head = next
```

Fig. 8. The `free_chain` procedure

3.2 The Benchmarks

We chose to use the same benchmarks as in the optimistic queue article [7].

- 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.
- Enqueue-Dequeue Pairs: each process alternately performs an `enqueue` or a `dequeue` operation.
- Grouped Operations: each process picks a random number between 1 and 16, and performs this number of enqueues or dequeues. The process performs enqueues and dequeues alternately as in the Enqueue-Dequeue Pairs benchmark.

The total number of enqueue and dequeue operations is not changed, they are only executed in a different order.

3.3 The Experiments

We ran the specified benchmarks measuring the total time required to perform one million operations as a function of the number of processes. For each benchmark and

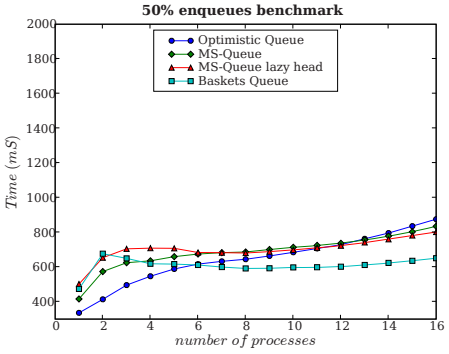

Fig. 9. The 50 % enqueues benchmark

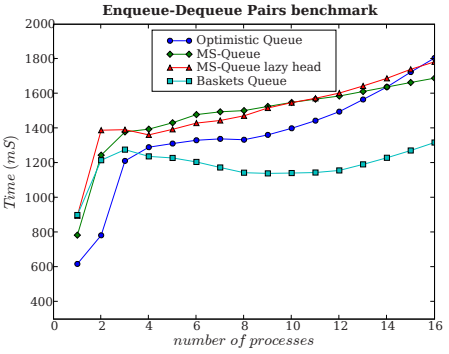

Fig. 10. The Enqueue-Dequeue pairs benchmark

algorithm we chose the exponential backoff delays that optimize the maximal latency (the maximal time required to complete an operation).

To counteract transient startup effects, we synchronized the start of the processes (i.e: no process started before all others finished their initialization phase).

3.4 Empirical Results

Figures 9, 10 and 11 show the results of the three different benchmarks. It can be seen that high levels of concurrency have only moderate effects on the performance of the Baskets Queue. The Baskets Queue is up to 25% faster than the other algorithms. This can be explained by the load on the tail of all the data-structures but the baskets queue, whereas in the baskets queue the contention on the tail is distributed among several baskets. However, at lower concurrency levels, the optimistic approach is superior because the basket-mechanism is triggered upon contention.

When we optimized the exponential backoff delays of the algorithms for each benchmark, we found that for the Basket Queue the optimal backoff delays of all three benchmark is identical. In contrast, for the other algorithms, no single combination of backoff-delays was optimal for all benchmarks. This is due to the fact that the exponential backoff is used only as a secondary backoff scheme when inserting into the baskets, thus it has only a minor effect on the performance.

To further test the robustness of our algorithm to exponential backoff delays, we conducted the same benchmark test without using exponential backoff delays. As seen in figures 12, 13 and 14, in this setting the Baskets Queue outperforms the other algorithms by a large factor. This robustness can be explained by the fact that the basket-mechanism plays the role of the backoff-mechanism by distributing concurrent enqueue operations to different baskets.

To gauge the effectiveness of the basket-mechanism on our 16 processor machine, we took snapshots of the list of baskets. Figure 15 shows a typical snapshot of the Baskets Queue on the 50% enqueues benchmarks. The basket sizes vary from only 1 to 3 nodes. In the average case, an enqueue operation will succeed to enqueue after at

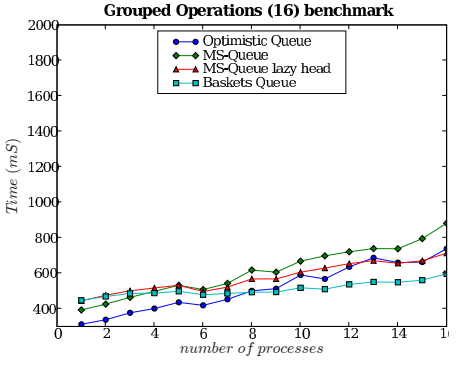

Fig. 11. The grouped operation benchmark

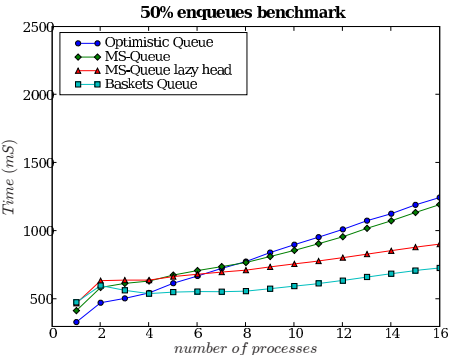

Fig. 12. The 50% enqueues benchmark without backoff

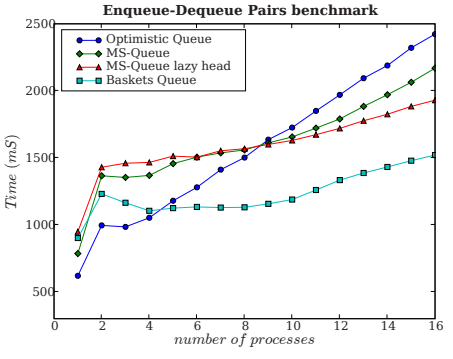

Fig. 13. The Enqueue-Dequeue pairs benchmark without backoff

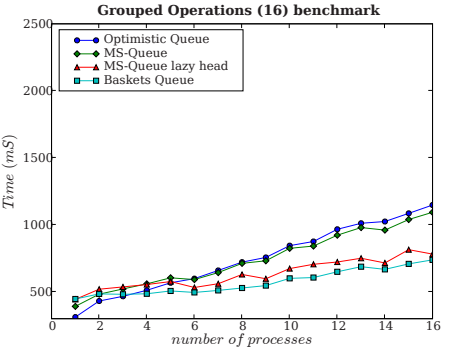

Fig. 14. The Grouped operations benchmark without backoff

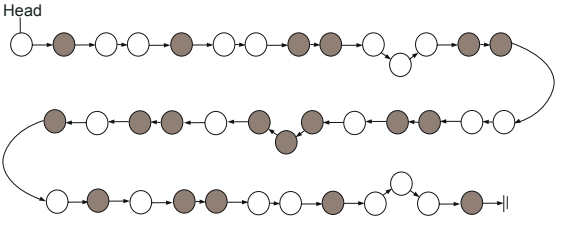

Fig. 15. A typical snapshot of the queue (16 processes)

most 3 failed CAS operations. The baskets sizes are smaller than 8 nodes as one would expect them to be, because the elements are inserted into the baskets one by one. This unnecessary synchronization on the nodes of the same basket imposes a delay on the last nodes to be inserted.

In addition to the robustness to exponential backoff delays, this snapshot confirms that when in use, the backoff-mechanism inside each basket needs only to synchronize at most 3 concurrent enqueues, if any. Therefore, it has only a minor effect on the overall performance. We believe that for machines where exponential backoff techniques are crucial for performance, this robustness makes our algorithm a natural solution as an out-of-the-box queue, to be used without the requirement of fine tuning.

Acknowledgement

We thank Edya Ladan Mozes for useful conversations and suggestions.

References

1. Tsigas, P., Zhang, Y.: A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems. In: Proceedings of the 13th annual ACM symposium on Parallel algorithms and architectures, Crete Island, Greece, pp. 134–143. ACM Press, New York (2001)
2. Valois, J.: Implementing lock-free queues. In: Proceedings of the 7th International Conference on Parallel and Distributed Computing Systems. 64–69 (1994)
3. 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(2), 164–189 (1983)
4. Herlihy, M., Wing, J.: Linearizability: A Correctness Condition for Concurrent Objects. ACM Transactions on Programming Languages and Systems 12(3), 463–492 (1990), doi:10.1145/78969.78972
5. Hwang, K., Briggs, F.A.: Computer Architecture and Parallel Processing. McGraw-Hill, New York (1990)
6. Lamport, L.: Specifying Concurrent Program Modules. ACM Transactions on Programming Languages and Systems 5(2), 190–222 (1983)
7. Ladan-Mozes, E., Shavit, N.: An Optimistic Approach to Lock-Free FIFO Queues. In: In: Proceedings of Distributed computing, Amsterdam, Netherlands, pp. 117–131. Springer, Heidelberg (2004)
8. Mellor-Crummey, J.M.: Concurrent queues: Practical fetch-and-ϕ algorithms. Technical Report Technical Report 229, University of Rochester(November 1987)
9. 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)
10. Prakash, S., Lee, Y.-H., Johnson, T.: A non-blocking algorithm for shared queues using compare-and-swap. IEEE Transactions on Computers 43(5), 548–559 (1994)
11. Sites, R.: Operating Systems and Computer Architecture. In: Stone, H. (ed.) Introduction to Computer Architecture, 2nd edn. (1980)
12. Stone, H.S.: High-performance computer architecture. Addison-Wesley Longman Publishing Co., Inc (1987)
13. 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 Press, Los Alamitos (1990)
14. Stone, J.M.: A Nonblocking Compare-and-Swap Algorithm for a Shared Circular Queue. In: Parallel and Distributed Computing in Engineering Systems, pp. 147–152. Elsevier Science B.V, Amsterdam (1992)

15. Treiber, R.K.: Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center(April 1986)
16. Michael, M., Scott, M.: Nonblocking Algorithms and Preemption-Safe Locking on Multiprogrammed Shared - Memory Multiprocessors. Journal of Parallel and Distributed Computing 51(1), 1–26 (1998)
17. Lea, D.: The java concurrency package (JSR-166), lhttp://gee.cs.oswego.edu/dl/concurrency-interest/index.html
18. Moir, M., Nussbaum, D., Shalev, O., Shavit, N.: Using elimination to implement scalable and lock-free FIFO queues. In: SPAA 2005: Proceedings of the 17th annual ACM symposium on Parallelism in algorithms and architectures, pp. 253–262. ACM Press, New York, NY, USA (2005)
19. Shavit, N., Zemach, A.: Diffracting Trees. ACM Transactions on Computer Systems 14(4), 385–428 (1996)
20. Herlihy, M., Lim, B., Shavit, N.: Scalable Concurrent Counting. ACM Transactions on Computer Systems 13(4), 343–364 (1995)
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. ACM Press, New York (1997)
22. Cormen, T., Leiserson, C., Rivest, R., Stein, C.: Introduction to Algorithms, 2nd edn. MIT Press, Cambridge, MA (2001)

A Memory Management

Our memory manager is similar to the memory manager of the optimistic queue [7]. It consists of a shared pool of nodes. Each thread possesses a list of pointers to nodes in the shared pool, ready for its exclusive allocation. When a thread physically removes a node from the queue, it adds the node's pointer to its list of available allocations.

Although the memory manager is rather simple, the algorithm can be easily adapted to interact with more sophisticated memory schemes as well as garbage collected languages.

A.1 The Tagging Mechanism and the ABA Problem

As our algorithm is based on CAS operations, it suffers from the known ABA problem [16,15]. To overcome it we use the standard tagging-mechanism approach [16,21]. A portion of each pointer address is used to timestamp changes of the pointer, where the pointer and the tag are manipulated atomically using a single CAS operation.

B Correctness Proof

Due to lack of space, the correct set semantics and lock-free proofs are omitted.

B.1 Linearizability of Our Algorithm

If by ordering the operations in the order of their linearization points the queue behaves as the abstract sequential queue, then the queue is linearizable to the abstract FIFO-queue.

Definition 1. *A sequential FIFO queue as defined in [22] is a data structure that supports two operations:* `enqueue` *and* `dequeue`*. The state of the queue is a sequence* $<e_1, ..., e_k>$ *of items. The queue is initially empty. The semantics of* `enqueue` *and* `dequeue` *operations on a given state* $<e_1, ..., e_k>$ *is described as follows:*

- `enqueue(n)` - *inserts n to the end of the queue yielding the new state* $<e_1, ..., e_k, n>$
- `dequeue()` - *if the queue is empty, the operation returns "empty" and does not change the state. Otherwise, it deletes and returns the oldest value from the queue, yielding a new state* $<e_2, ..., e_k>$

Definition 2. *The linearization point of a* `dequeue` *operation that returned a value is the successful pointer marking at line D23.*

Definition 3. *The linearization point of a* `dequeue` *operation that returned "empty" is when reading the* `dummy` *node's next null pointer at line D04.*

Definition 4. *The linearization points of the* `enqueue` *operations of a basket are set inside the basket's shared time interval in the order of their respective dequeues. In other words, the linearization points of the enqueues are determined only once the items are dequeued.*

Lemma 1. *The* `enqueue` *operation of the same basket overlap in time.*

Proof. An `enqueue` operation tries to insert a node into a basket only if it failed to CAS on the `tail` of the list (E09). Before trying to CAS, the `enqueue` operation checks that the `next` pointer of the tail-node is null. Thus, all the failed `enqueue` operations overlap the time interval that starts at some point when the `next` pointer of the tail-node is null, and ends when it points to a new node. The winner of the CAS overlap the same interval too. □

Lemma 2. *The baskets are ordered according to the order of their respective time intervals.*

Proof. A basket is created by a successful `enqueue` operation on the `tail` of the queue. The `enqueue` operations that failed to enqueue, retry to insert their nodes at the same list position. Therefore, the first node of a basket is next to the last node of the previous basket, and the last node of a basket is the winner of the CAS operation. □

Lemma 3. *The linearization points of the* `dequeue` *operations of a basket come after the basket's shared time interval.*

Proof. In order for a `dequeue` operation to complete, the node must be in the list. A basket's first node is linked into the list only after the CAS operation on the `tail` is completed. The completion of this CAS operation is also the end of the shared time interval of the basket. Thus nodes can be marked only after the basket's time interval. □

Lemma 4. *The nodes of a basket are dequeued before the nodes of later (younger) baskets.*

Proof. The nodes are dequeued according to their sequential order in the list (which is logically divided into baskets). In addition, since the nodes are deleted by pointer marking, once all the nodes of a basket are dequeued, no more nodes are allowed to be enqueued into it. □

Lemma 5. *The linearization point of a dequeue operation that returned "empty" comes exactly after an equal number of enqueue and dequeue operations.*

Proof. If the `next` pointer of the dummy node is null then all the enqueued nodes had been removed from the list. Since nodes are removed from the list only after they are marked, the linearization point of an "empty" dequeue comes after equal number of enqueue and dequeue linearization points. □

Theorem 1. *The FIFO-queue is linearizable to a sequential FIFO queue.*

Proof. Ignoring for a moment dequeues that return "empty", from lemmas 2 and 4, the order in which baskets are dequeued is identical to the order in which baskets are enqueued. From lemma 3 the `enqueue` operations of a basket preceed its dequeues. Lemma 1 guarantees that the construction of definition 4 is possible. Thus the order of the `enqueue` operations of a basket is identical to the order of its `dequeue` operations, and the queue is linearizable.

From lemma 5 the queue is linearizable also with respect to dequeue operations that returned "empty". □