FIFO Queue Synchronization

by

Moshe Hoffman

A Thesis submitted for the degree

Master of Computer Science

Supervised by

Professor Nir Shavit

School of Computer Science

Tel Aviv University

July 2008

CONTENTS

1. Introduction 1
1.1 A Statement of the Problem 2
1.2 Terms and Definitions 2
1.3 Literature review 3

2. The Baskets Queue 5
2.1 The Abstract Baskets Queue 5
2.2 The Algorithm 6
2.3 Data Structures 6
2.4 The Baskets Queue methods 7
2.5 Solving the ABA problem 8

3. Correctness 12
3.1 Correct Set Semantics 12
3.2 Linearizability of the algorithm 14
3.3 Lock-freedom 15

4. Evaluation 18
4.1 The Benchmarked Algorithms 18
4.2 The benchmarks 18
4.3 The experiments 19
4.4 Empirical results 19

5. Conclusions 26
5.1 Results 26
5.2 Future Work 26

LIST OF FIGURES

2.1 The abstract Baskets Queue 6
2.2 Concurrent enqueues 7
2.3 A queue composed of 4 baskets 8
2.4 Types, structures and initialization 8
2.5 The `enqueue` method 9
2.6 The `dequeue` method 10
2.7 Dequeue illustration 11
2.8 The `free_chain` procedure 11

4.1 The 50 % enqueues benchmark 19
4.2 The Enqueue-Dequeue pairs benchmark 20
4.3 The grouped method benchmark 21
4.4 The 50% enqueues benchmark without backoff 22
4.5 The Enqueue-Dequeue pairs benchmark without backoff 23
4.6 The Grouped method calls benchmark without backoff 24
4.7 A typical snapshot of the queue (16 processes) 25

ACKNOWLEDGEMENTS

I would like to thank all those who supported me during my study.

My thesis advisor Prof. Nir Shavit for accepting me, for his expertise, patience and support throughout my research. Dr. Ori Shalev for his personal guidance during and before the research, and for introducing me to world of multiprocessor synchronization. Special gratitude to Jenny Shalev who gave me the opportunity to complete my studies. I wouldn't be performing this work without her help. I thank my father for encouraging me to complete my studies and for the English grammar lessons. Finally my mother for the love and support all along the way.

ABSTRACT

FIFO Queues are among the most highly used concurrent data structures, and over the years have been the subject of a significant body of research. Such queues are used as buffers in a variety of applications, and in recent years as key tools 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. This thesis describes 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` method calls 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, it is shown that a basket queue based on the MS-queue outperforms the original MS-queue algorithm in various benchmarks.

1. INTRODUCTION

Multiprocessors use data-structure in their shared memory to coordinate multiple execution threads. In order to communicate, the threads manipulate the contents of the shared memory by applying methods to the shared objects. The concurrent objects are required to be appear "atomic" yet be fast and scalable. This seemingly contradictory requirement poses a great challenge in the design and implementation of concurrent data-structures.

The conventional method for achieving atomicity is to use spin locks. Each thread must acquire the shared lock before accessing its critical section, and release it afterwards. The lock semantics guarantees that only one thread at a time can acquire it, thus fulfilling the atomicity property. However, locks are blocking, all other threads that fail to acquire the lock have to wait until the lock is released. This is an undesired property, as the threads are subject to the scheduling of the operating system.

To overcome the disadvantages of using locks, researchers have introduced lock-free and wait-free data-structure. The lock-freedom property guarantees that every step of the threads assures progress in the system. Thus, in a finite number of steps, some thread will successfully access the shared resource. Wait-free algorithms guarantee that any thread will complete its operation given enough execution time, regardless the execution of other threads.

First-in-first-out (FIFO) queues are important data-structures for thread synchronization. They are used everywhere, from operating system cores to user applications, and in recent years also in high speed network equipment. The vast majority of concurrent queues published in the literature do not scale to high levels of concurrency. This is due to the fact that the threads try to synchronize on only two memory locations - the head and the tail of the queue. These bottlenecks impose a heavy penalty on the queue's parallelism. In this thesis I present a new approach that allows added parallelism in the design of concurrent shared queues.

In the 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 last-in-first-out (LIFO) order. Nevertheless, I will 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` method calls by allowing insertions into the different baskets to take place in parallel.

1.1 A Statement of the Problem

The primary goal of this thesis is to improve the time needed for threads to synchronize in highly concurrent environments. And in particular, to find a new technique to add more parallelism into FIFO-queues.

1.2 Terms and Definitions

Definition 1 (SMP). *SMP is a multiprocessor architecture where two or more processors are connected to a main memory.*

Definition 2 (sequential Queue). *A sequential FIFO queue as defined in [1] is a data structure that supports two methods:* `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 the* `enqueue` *and* `dequeue` *methods 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 method 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 3 (Linearizable History). *The history of a concurrent object is linearizable if[4]*

1. *All method calls have a linearization point at some instant between their invocation and their response*
2. *All functions appear to occur instantly at their linearization point, behaving as specified by the sequential definition*

Definition 4 (Lock-Freedom). *From any point in the execution, in a finite number of steps some thread will complete a method call.*

Definition 5 (Dynamic Memory). *Algorithms with dynamic memory allocation scheme, as opposes to static allocation, supports the allocation of needed memory in runtime.*

Definition 6 (Compare-and-Swap (CAS)). *The Compare-and-Swap hardware operation, also known as the CAS operation, atomically compares a memory location to a given value, and swaps it with a new given value if they are equal.*

Definition 7 (Winner of a CAS operation). *When a group of threads try to change a particular value in memory by applying a compare-and-swap operation, only one can succeed, hence it is called the winner.*

Definition 8 (Overlapping method calls). *A group of method calls overlap if there is a common instant between all the invocations and responses of the method calls in the group.*

Definition 9 (The A-B-A problem)**.** *The ABA problem might arise when a thread rereads a memory location, and if its value is unchanged subsequent to the first read it establishes that the overall state of the data-structure is unchanged. However if between the two read operations other threads did alter the data-structure but a final alteration reset the memory value to its original value, the ABA problem occurs, because the thread falsly concludes that the data structure was not changed.*

1.3 Literature review

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 [21, 22] and in a dynamic one [2, 4, 5, 7, 6, 9, 13, 14, 16, 17, 18, 19, 20, 22]. The classical concurrent queue is a linearizable structure that supports the `enqueue` and `dequeue` methods with the usual FIFO semantics.

Gottlieb, Lubachevsky, and Rudolph [2] present a statically-allocated FIFO queue based on the *replace-add* synchronization primitive. However the queue is blocking, as concurrent enqueue and dequeue method calls synchronize on the array's cells.

Stone [18] presents a dynamically-allocated FIFO queue based on the *double-compare-and-swap* operation. The queue consists of a linked list of nodes. In order to enqueue a new item, the thread sets the tail pointer to the new item, and then links the old tail item to the new item. However the algorithm is not non-blocking: a faulty enqueuer that halts after setting the tail pointer to the new item may block other dequeuers.

Prakash, Lee, and Johnson [14] present a FIFO queue that is both lock-free and linearizable. To overcome the blocking problem when slow threads leave the data structure in an intermediate state, a helping technique is used. A fast thread can help a slow thread to complete its operation.

Valois [23] presents a linearizable and lock-free dynamically-allocated queue based on a linked list of nodes. In order to solve the synchronization problems associated with empty and single-item queues, a dummy node is kept at the head of the linked list. However, the tail pointer may lag behind the queue's head, thus freeing items might be unsafe, and a special memory reference-counting mechanism must be employed to prevent it.

The best known linearizable FIFO queue implementation is the lock-free queue of Michael and Scott [10] which is included in the Java^{TM} Concurrency Package [8]. 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` method calls to execute in parallel with `dequeue` method calls.

Tsigas and Zhang [21] present a non-blocking and linearizable statically-allocated queue. By allowing the tail and head pointer to lag at most m behind the actual head and tail of the queue, only one of m method calls has to apply a CAS operation to adjust the head or tail. In this way the amortized number of CAS operations for a dequeue or an enqueue is only $1 + 1/m$.

A later article by Ladan-Mozes and Shavit [6] 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) oper-

ations 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`.

In an attempt to add more parallelism, Moir et. al [12] showed how one could use elimination [15] as a back-off scheme to allow pairs of concurrent `enqueue` and `dequeue` method calls to exchange values without accessing the shared queue itself. Unfortunately, in order to keep the correct FIFO queue semantics, the `enqueue` method cannot be eliminated unless all previous inserted nodes have been dequeued. Thus, the *elimination backoff queue* is practical only for very short queues or high overloads.

2. THE BASKETS QUEUE

2.1 The Abstract Baskets Queue

For a FIFO queue, an execution history is linearizable if one can pick a point within each `enqueue` or `dequeue` method's execution interval so that the sequential history defined by these points maintains the FIFO order.

The definition of linearizability allows overlapping method calls to be reordered arbitrarily. This observation leads to the key idea behind our algorithm: a group of overlapping `enqueue` method calls can be enqueued onto the 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` method calls can be "fixed" to meet the `dequeue` method calls order.

A concise abstraction of the new queue is a FIFO-ordered list of baskets where each basket contains one or more nodes (see Fig. 2.1). The baskets fulfill the following basic rules:

1. Each basket has a time interval in which all its nodes' `enqueue` method calls overlap.
2. The baskets are ordered by the order of their respective time intervals.
3. For each basket, its nodes' `dequeue` method calls occur after its time interval.
4. The `dequeue` method calls 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` method calls that share an interval according to the order of their respective `dequeues`, yields a linearizable FIFO-queue.

How do we detect which `enqueue` method calls overlap, and can therefore fall into the same basket? The answer is that in algorithms such as the MS-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 thread that succeeded) 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.

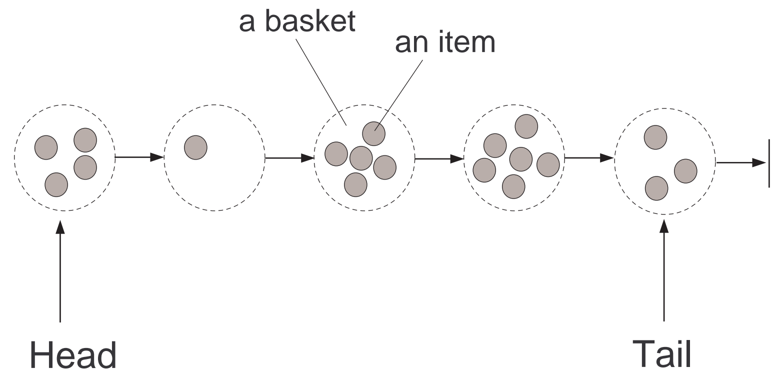

Fig. 2.1: The abstract Baskets Queue

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 winning thread (see Fig. 2.2). 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.

2.2 The Algorithm

The implementation of the Baskets Queue I present here is based on Michael and Scott's MS-queue. The algorithm maintains a linked list of nodes logically divided into baskets (see Fig. 2.3). Although, in the implementation the baskets have a stack-like behavior, any concurrent pool object that supports the `add` and the `remove` methods, can serve as a basket. The advantage of such objects is that they can deliver more scalability than the stack-like baskets.

Since CAS operations are employed in the algorithm, ABA issues arise [10, 20]. In Section 2.4, the `enqueue` and `dequeue` methods are described ignoring ABA issues. The tagging mechanism that was added to overcome the ABA problem is explained in Section 2.5. The code in this section includes this tagging mechanism.

2.3 Data Structures

Just as in the MS-queue, the queue is implemented as a linked list of nodes with `head` and `tail` pointers (see figure 2.4). The `tail` points either to a node in the last basket, or in the second to last basket. In contrast to the MS-queue, pointer marking [14] is used 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.

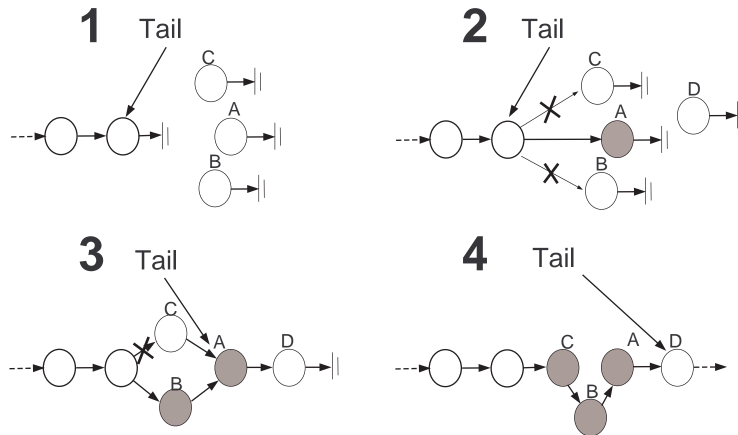

Fig. 2.2: (1) Each thread checks that the tail-node's next field is null, and tries to atomically change it to point to its new node's address. (2) Thread A succeeds to enqueue the node. Threads B and C fail on the same CAS operation, hence both of them will retry to insert into the basket. (3) Thread B was the first to succeed to enqueue, at the same time thread D calls the enqueue method, and finishes successfully to enqueue onto the tail. (4) thread C finishes successfully.

2.4 The Baskets Queue methods

The FIFO queue supports two methods: `enqueue` (figure 2.5) and `dequeue` (figure 2.6). 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 thread that succeeded to CAS (and possibly with more failed threads). 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 found (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 be inserted adjacent 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 [21] 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 re-checks it for consistency (D05). If the `head` and `tail` of the list point to

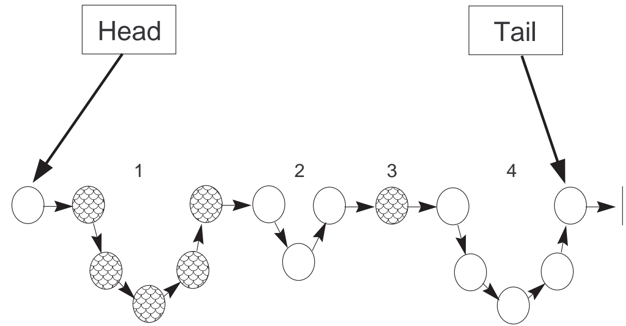

Fig. 2.3: Linked list of nodes logically divided into four different baskets

```
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. 2.4: Types, structures and initialization

the same node (D06), then either the list is empty (D07) or the `tail` lags. In the latter case, the last node is found (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. 2.4 for an illustration.

The `free_chain` procedure (figure 2.8)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).

2.5 Solving the ABA problem

In order to solve the ABA problem I use the standard tagging-mechanism approach [10, 11]. The low bits of the address pointer are used as "tag" bits. The tag bits are manipulated atomically with the pointer in each CAS operation.

```
void enqueue(queue_t* q, data_type val)

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+1>
E09:          if CAS(&tail.ptr->next, next, <nd, 0, tail.tag>):
E10:             CAS(&Q->tail, tail, <nd, 0, tail.tag+1>
E11:             return True
E12:          next = tail.ptr->next
E13:          while((next.tag==tail.tag) and (not next.deleted)):
E14:             backoff_scheme()
E15:             nd->next = next
E16:             if CAS(&tail.ptr->next, next, <nd, 0, tail.tag>):
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. 2.5: The `enqueue` method

Thus, the next CAS operations on the pointer will fail even though the address is the same, as the tag value is different.

When the queue is initialized the tags of the `head` and `tail` are initialized to zero. The tag of the `next` pointer of the dummy node is initialized to 1. As can be seen in the code, I apply the following tag manipulation rules:

1. When applying a CAS operation on the `head` and on the `tail`, the tag is incremented by 1 (lines E10, E22 and D11).
2. When a new node is linked to the `tail` of the queue, its `next` pointer tag is set to be one greater then the tail's tag (line E08).
3. When a new node is inserted into a basket, its `next` pointer tag is set to be the be same as the node it is linked to (line E15).

```
const MAX_HOPS = 3 # constant

data_type dequeue(queue_t* Q)

D01: repeat
D02:    head = Q->head
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>):
D26:                if (hops >= MAX_HOPS):
D27:                   free_chain(Q, head, next)
D28:                return value
D29:             backoff-scheme()
```

Fig. 2.6: The `dequeue` method

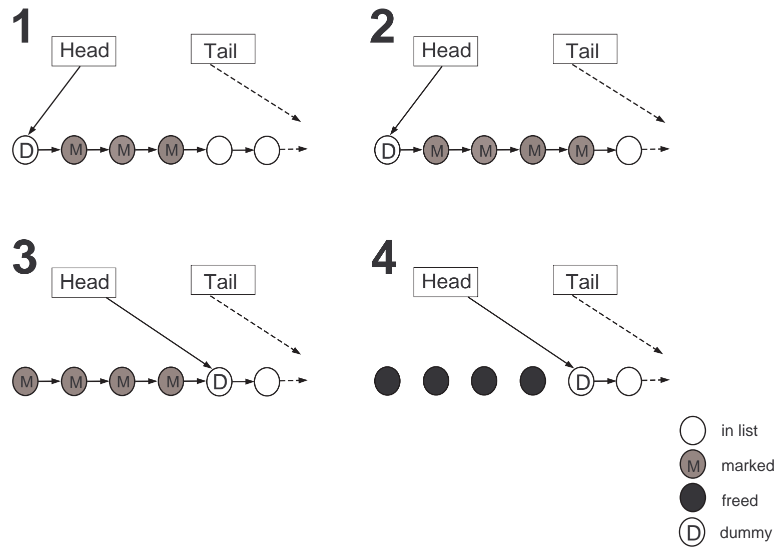

Fig. 2.7: (1) three nodes are logically deleted. (2) the first non-deleted node is deleted (3) the head is advanced (4) the chain of deleted nodes can be reclaimed

```
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. 2.8: The `free_chain` procedure

3. CORRECTNESS

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

Initially I will prove that the Baskets Queue algorithm has the correct set semantics regardless of the order in which the nodes are dequeued. Then I will show that the algorithm is linearizable to the abstract sequential queue. Finally, I will prove that the algorithm is lock-free.

3.1 Correct Set Semantics

Definition 10. *A queue data structure has correct set semantics [3] if its methods meet the following requirements:*

- *If a* `dequeue` *method returns an item i, then i was previously inserted by an* `enqueue` *method.*
- *If an* `enqueue` *method inserted the item i into the queue, then there is at most one* `dequeue` *method that returns it.*

Similarly to the MS-queue correctness proof, I will show that the algorithm maintains the following properties by induction on the states of execution:

1. The queue is a connected list of nodes.
2. The list consists of a sequence of marked nodes followed by a sequence of unmarked nodes.
3. Nodes are marked (logically-deleted) from the start of the unmarked nodes' sequence.
4. New nodes are inserted anywhere after the marked nodes' sequence.
5. Nodes are deleted (freed) only after they are marked (logically deleted) and removed from the list.
6. The `tail` always points to a node in the list.
7. The `head` always points to a node in the marked nodes' sequence.

These properties are fulfilled by the initial queue.

Property 1. *The queue is a connected list of nodes.*

Proof. The `next` pointers of the nodes in the list are changed only by `enqueue` method calls. When enqueuing into a basket, the `next` pointer is atomically set to point to the new node (E16), which already points to the next node (E15).

When the new node is added to the end of the list (E09), its `next` pointer is set to NULL (E08).

From property 5, a node is deleted (freed) only after it is removed from the list by advancing the `head` forward down the list. □

Property 2. *The list consists of a sequence of marked nodes followed by a sequence of unmarked nodes.*

Proof. By property 1, the list of nodes is connected. By property 3, nodes are marked only at the beginning of the unmarked nodes' sequence. Finally, by property 4, new nodes are inserted only to the unmarked nodes's sequence. □

Property 3. *Nodes are marked (logically-deleted) from the start of the unmarked nodes' sequence.*

Proof. In order to find an unmarked node, the thread searches for the first unmarked node (D15-D18). Since the `head` always points to a node in the marked nodes' sequence (property 7), the first node to be found is the first node in the unmarked sequence. □

Property 4. *New nodes are inserted anywhere after the marked nodes' sequence.*

Proof. Since the `tail` always points to a node in the list (property 6), the last node in the linked list can be found (E20-E21). The list is always connected (property 1), thus a node with a NULL pointer can only be the last node in the list. An `enqueue` method that adds a node to the end of the list, atomically verifies that the new node is inserted after the last node (E07,E09).

An `enqueue` method that enqueues a node into a basket, ensures the pointer is not marked (E13), thus it can only be inserted adjacent to another unmarked node. □

Property 5. *Nodes are deleted (freed) only after they are marked (logically deleted) and removed from the list.*

Proof. Nodes are deleted (F04) only after they are removed (F01) by atomically changing the `head`. The `head` always points to a marked node (property 7). By property 2, all the removed nodes between the old and the new `head` are marked nodes. □

Property 6. *The* `tail` *always points to a node in the list.*

Proof. On the one hand, the `head` of the queue is changed only by the calls to the `free_chain` procedure in lines D22 and D27. The *while* statement condition in line D15 ensures that `iter` will never point to a node beyond the `tail`. The call to `free_chain` in line D22 points the `head` to the `tail`. The `free_chain` procedure in line D27 is called only if `iter` points to an item strictly before the `tail`, therefore the queues's `head` can never point beyond the `tail`, and items pointed by the `tail` are never freed.

On the other hand, the `tail` only advances in the direction of the `next` pointers. □

Property 7. *The* `head` *always points to a node in the marked nodes' sequence.*

Proof. The while-loop ensures that the local variable "`iter`" points either to the queue's `head` (D13) or to another marked node (D15). Thus by calling the method `free_chain` (D20), the `head` will be pointed to a another marked node. In line D25, a call to the method `free_chain` (to point `head` to "`next`") is made only if "`next`" was successfully marked (D23). □

Theorem 1. *The queue has correct set semantics.*

Proof. By property 1, the queue is a connected list of nodes. Therefore if a `dequeue` method call returns an item i, then i was previously inserted by an `enqueue`. The `dequeue` method returns a value only if it successfully marked the pointer to its item(line D25). Therefore, an item can be dequeued only once. □

3.2 Linearizability of the algorithm

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

Definition 11. *The linearization point of a successful* `dequeue` *method (one that returned a value) is the successful pointer marking at line D23.*

Definition 12. *The linearization point of an unsuccessful* `dequeue` *method (one that returned "empty") is when reading the* `dummy` *node's next null pointer at line D04.*

Definition 13. *The linearization points of the* `enqueue` *method calls 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` *method calls of the same basket overlap in time.*

Proof. An `enqueue` method 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` method checks that the `next` pointer of the tail-node is null. Thus, all the failed `enqueue` method calls 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 thread that succeeded to CAS overlaps 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` method on the `tail` of the queue. The `enqueue` method calls 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.

Moreever, the order of nodes in the connected list is not changed by `dequeues`. □

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

Proof. In order for a `dequeue` method to complete, the node to be dequeued 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. This follows from property 3 and lemma 2. □

Lemma 5. *The linearization point of a* `dequeue` *method call that returned "empty" comes exactly after an equal number of successful* `enqueue` *and successful* `dequeue` *method calls.*

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 successful `enqueue` and successful `dequeue` linearization points. □

Theorem 2. *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` methods of a basket preceed its `dequeues`. Lemma 1 guarantees that the construction of definition 13 is possible. Thus the order of the `enqueue` method calls of a basket is identical to the order of its `dequeue` method calls, and the queue is linearizable.

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

3.3 Lock-freedom

This section contains the proof that the concurrent implementation is lock-free. In other words, if a thread does not complete a method call in a finite number of steps, it is because other other threads infinitely succeed in completing method calls.

Lemma 6. *If a thread failed on one of the CAS operations in lines E09, E16 or D25, then another thread has succeeded in one of those CAS operations.*

Proof. The `next` pointers of the list's nodes are changed only by the CAS operations in the lines E09, E16 and D25. From the semantics of the CAS operation, one thread must succeed. □

Lemma 7. *If one of the CAS operation in lines E09, E16 or D25 is executed infinitely often, an infinite number of* `enqueue` *method calls will also have been successfully completed.*

Proof. From lemma 6, one of these CAS operation is successfully executed infinitely often. If threads successfully execute the CAS operations is lines E09 and E16 infinitely often, then there is an infinite number of completed `enqueues` as well. By theorem 1, an item can be dequeued only once. Thus, if the CAS operation in line D25 is successfully executed infinity often, then there are infinite number of completed `enqueues`. □

Lemma 8. *If there is only a finite number of successful CAS operations at lines E09 and E16, then the queue's* `tail` *and* `head` *may be updated only a finite number of times.*

Proof. From the lemma assumption, only a finite number of nodes is added to the list. From properties 1, 6 and 7, the last node is reachable within a finite number of pointer traversals from the `head` and from the `tail`.

In each update of the `head` and `tail` (lines D11, D22, D27 and lines E10, E22), the `head` and the `tail` are changed to a node that is closer to the last node.

Thus after a finite number of updates the `head` and the `tail` will point to the last node.

□

Lemma 9. *If an* `enqueue` *method call does not terminate its execution in a finite number of steps, then other method calls complete infinitely often.*

Proof. Assume on the contrary, that an `enqueue` method call does not terminate, and only a finite number of concurrent `enqueue` and `dequeue` method calls succeed.

An `enqueue` method call may infinitely loop in the while-loop in lines E20-E21, E13-E18 or in the main loop (lines E03-E22).

By properties 1 and 6, the while-loop in lines (E20-E21) will eventually terminate by reaching the last node (or sooner, if the `tail` was changed).

If the `enqueue` method call infinitely loops in the while loop E13-E18 then the CAS operation in line E16 is being executed an infinite number of times. Thus, by lemma 7 there is an infinite number of successful `enqueues` - a contradiction.

Therefore, the `enqueue` method infinitely loops in the main loop. From lemma 7, the `enqueue` method reaches the CAS statements at line E09 only a finite number of times (otherwise there would be an infinite number of `enqueues`). Thus one of the conditions in lines E06 and E07 fails infinitely often.

- If the condition in line E06 fails infinitely often, then other threads update the `tail` infinitely often.
- If the condition in line E07 fails infinitely often, then the `enqueue` method will try to update the `tail` infinitely often, and whenever it fails to update the `tail`, it is because the `tail` was successfully updated by another thread.

Thus if one of the conditions fails infinitely often, then the `tail` is updated infinitely often. In contradiction to lemma 8. □

Lemma 10. *If a* `dequeue` *method does not terminate in a finite number of steps, then other method calls succeed infinitely often.*

Proof. Assume the contrary, that a `dequeue` method does not terminate, and only a finite number of concurrent `enqueue` and `dequeue` method calls succeed. A `dequeue` method may infinitely loop in the while-loop in lines D09-D10, D15-D18, in the `free_chain` method's loop or in the main loop (lines E03-E22).

By properties 1 and 6, the while-loop in lines (D09-D10) will eventually terminate by reaching the last node (or sooner, if the `tail` was changed).

By properties 1, 6 and 7, the `tail` is always reachable from the `head`, thus the while-loop in lines (D15-D18) will eventually terminate by reaching the node pointed to by `tail` (or sooner, if the `head` was changed).

By property 1, the while-loop of the `free_chain` method terminates as only the thread that changed the `head` frees the chain of nodes.

Therefore, the `dequeue` method infinitely loops in the main loop. The execution does not reach the statement in line D08 (otherwise the loop would terminate). Thus one of the conditions in lines D05, D06 or D07 fails infinitely often.

- If the condition in line D05 fails infinitely often, then other threads update the `head` infinitely often.
- If the condition in line D06 fails infinitely often, then the process will execute one of the *if* statement branches in lines D19, D21 or D23 infinitely often.
  - If the branch of line D19 is executed infinitely often then the `head` is updated infinitely often.
  - If the branch of line D21 is executed infinitely often then the thread will try to update the `head` infinitely often, and whenever it fails to update the `head`, it is because the `head` was successfully updated by another thread.
  - If the branch of line D23 is executed infinitely often, then the thread will execute the CAS operation in line D25 infinitely often. In contradiction to lemma 7.
- If the condition in line D07 fails infinitely often, then the thread will will try to update the `tail` infinitely often, and whenever it fails to update the `tail`, it is because the `tail` was successfully updated by another thread.

Thus if any one of the conditions fails infinitely often, then the `tail` or the `head` are infinitely being updated. This is in contradiction to lemma 8, and as well the `dequeue` method call can not infinitely loop in the main loop. □

Theorem 3. *The FIFO-queue is lock-free.*

Proof. By lemma 9 if an `enqueue` method call does not terminate in a finite number of steps then other method calls succeed infinitely often. By lemma 10 if a `dequeue` method call does not terminate in a finite number of steps then other method calls succeed infinitely often. □

4. EVALUATION

I compared the new lock-free queue algorithm to the lock-free MS-queue of Michael and Scott [10], and to the Optimistic-Queue by Ladan-Mozes and Shavit [6]. The algorithms were compiled in the C programming language with Sun's "CC" compiler 5.8 with the flags "-XO3 -xarch=v8plusa". The different benchmarks were executed on a 16 processor Sun Fire™ 6800 running the Solaris™ 9 operating system.

4.1 The Benchmarked Algorithms

The Baskets Queue algorithm is compared to the lock-free queue of Michael and Scott [10], and to the Optimistic Queue of Ladan-Mozes and Shavit [6]. To expose the possible effects of the 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 [21], the queue's head may be updated only once for several `dequeues`.

4.2 The benchmarks

The benchmarks used are those used by Ladan and Shavit [6] and Michael and Scott [10].

- 50% Enqueues: each thread chooses uniformly at random whether to perform an `enqueue` or a `dequeue`, creating a random pattern of 50% `enqueue` and 50% `dequeue` method calls.
- Enqueue-Dequeue Pairs: each thread alternately performs an tt enqueue or a `dequeue` method.
- Grouped method calls: each thread picks a random number between 1 and 16, and performs this number of `enqueues` or `dequeues`. The thread performs `enqueues` and `dequeues` alternately as in the Enqueue-Dequeue Pairs benchmark.

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

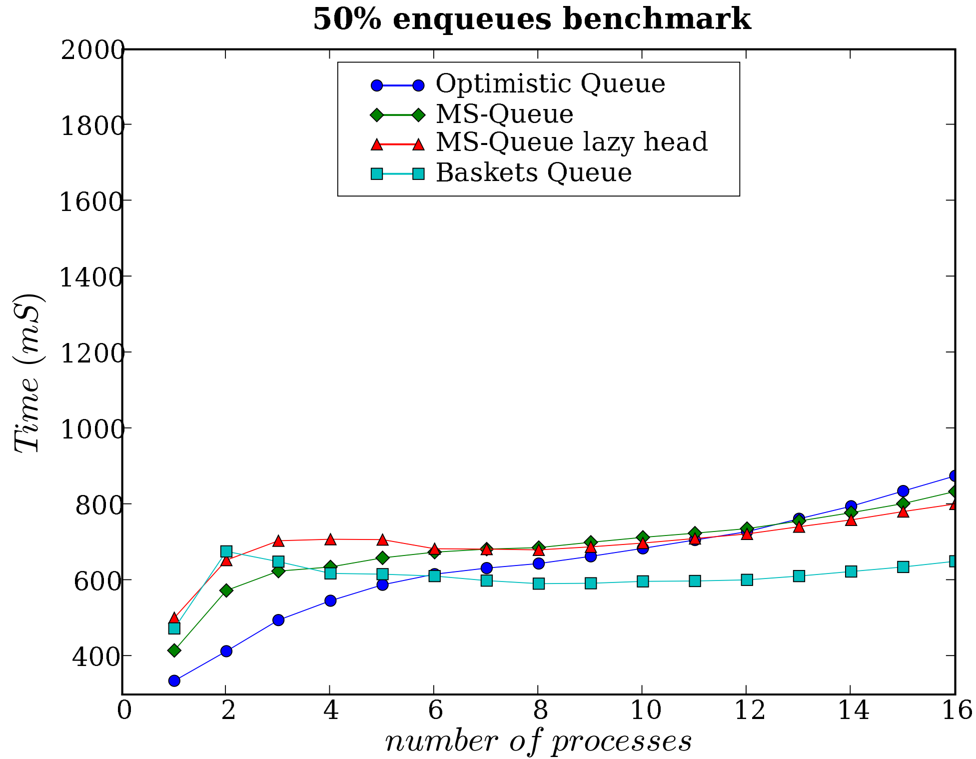

Fig. 4.1: The 50 % enqueues benchmark

4.3 The experiments

The benchmarks measure the total time required to perform one million method calls as a function of the number of processes. For each benchmark and algorithm I chose the exponential backoff delays that optimize the maximal latency (the maximal time required to complete an method).

To counteract transient startup effects, the processes' start time is synchronized (i.e: no thread started before all others finished their initialization phase).

4.4 Empirical results

Figures 4.1, 4.2 and 4.3 show the results of the four 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, 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 only upon contention.

When I optimized the exponential backoff delays of the algorithms for each

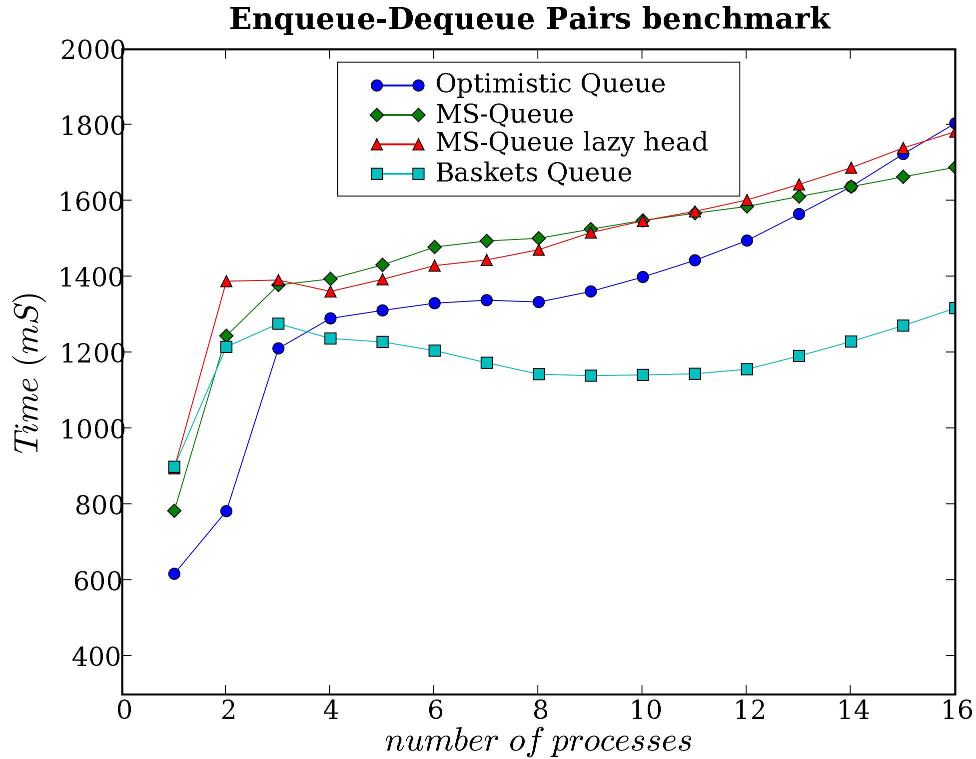

Fig. 4.2: The Enqueue-Dequeue pairs benchmark

benchmark, I noticed 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 the algorithm to exponential backoff delays, the same benchmark test was conducted without using exponential backoff delays. As seen in figures 4.4, 4.5 and 4.6, in this setting the Baskets Queue significantly outperforms the other algorithms. This robustness can be explained by the fact that the basket-mechanism assumes the role of the backoff-mechanism by distributing concurrent `enqueue` method calls to different baskets.

To gauge the effectiveness of the basket-mechanism on the 16 processor machine, snapshots of the list of baskets were taken. Figure 4.7 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` method will succeed to enqueue after at 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.

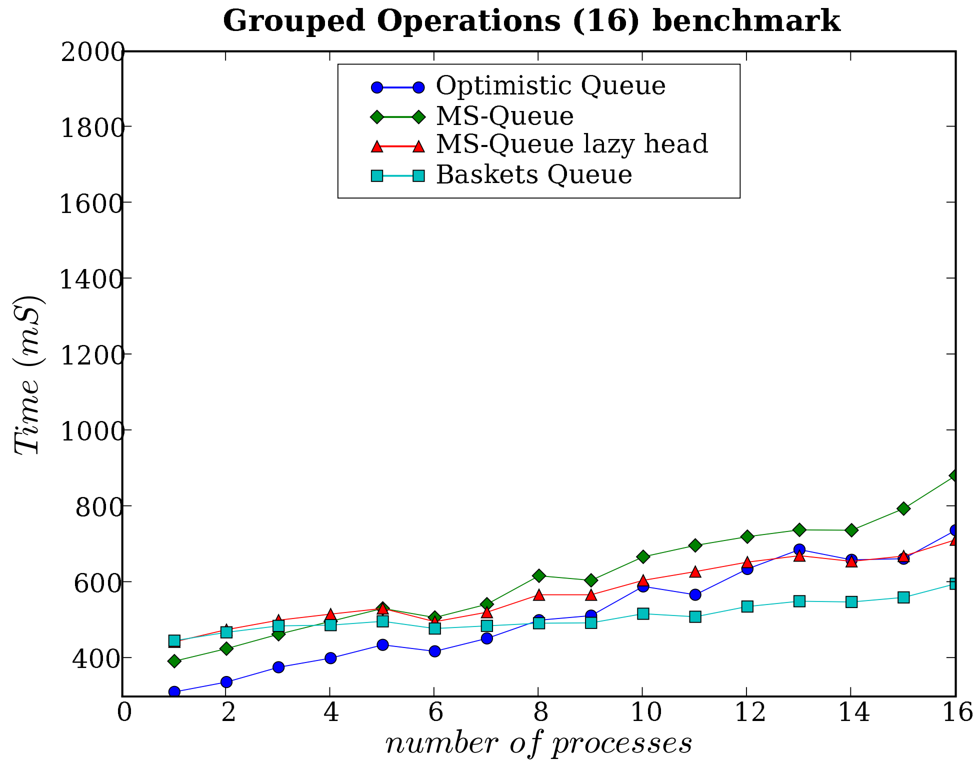

Fig. 4.3: The grouped method benchmark

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. I believe that for machines where exponential backoff techniques are crucial for performance, this robustness makes the Baskets Queue algorithm a natural solution as an out-of-the-box queue, to be used without the requirement of fine tuning.

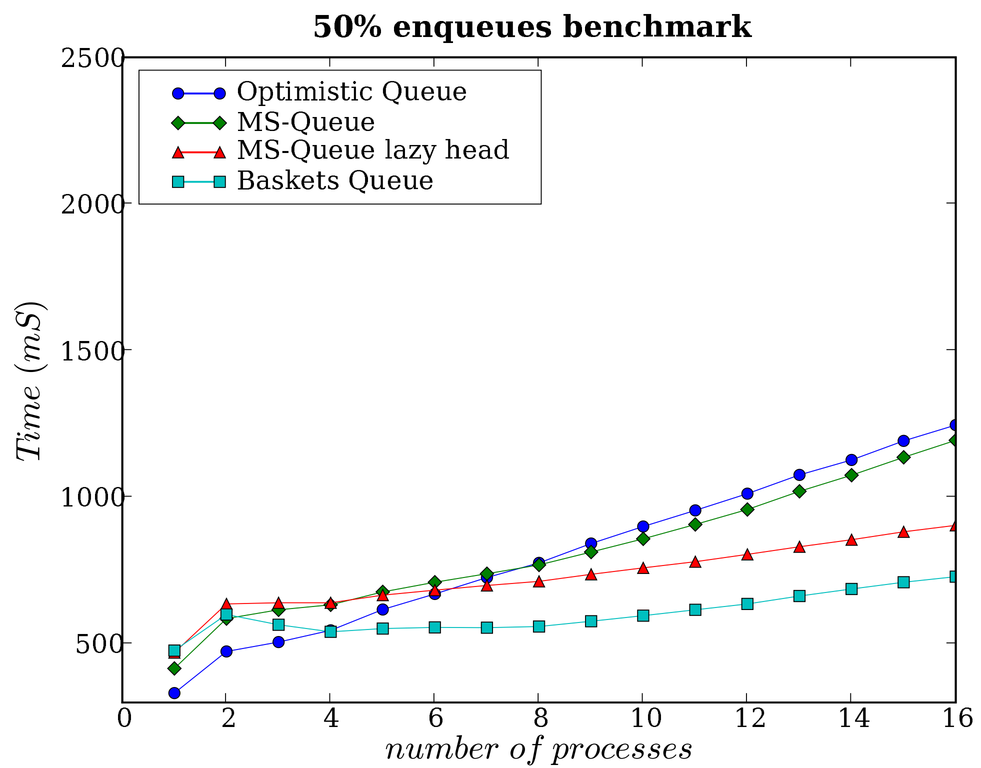

Fig. 4.4: The 50% enqueues benchmark without backoff

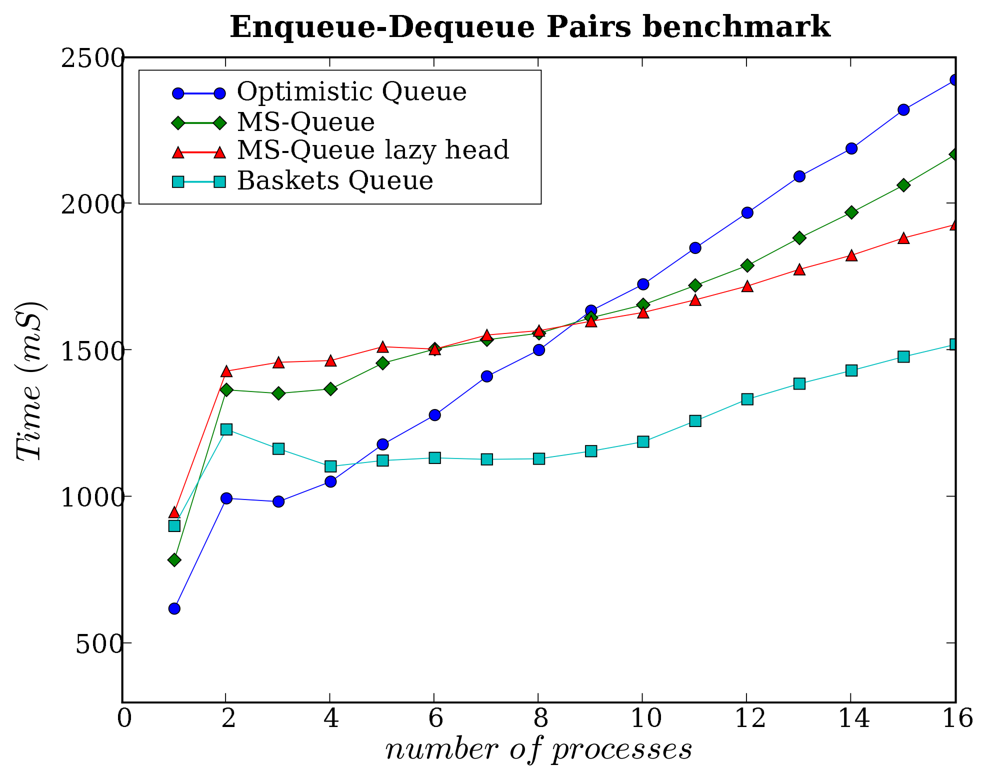

Fig. 4.5: The Enqueue-Dequeue pairs benchmark without backoff

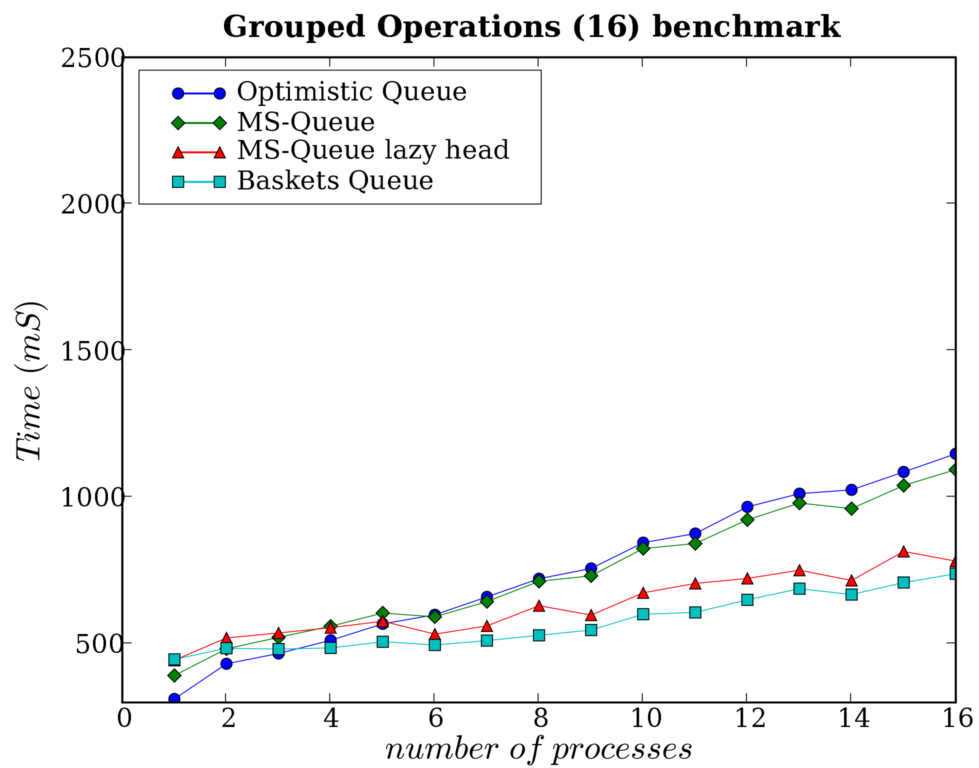

Fig. 4.6: The Grouped method calls benchmark without backoff

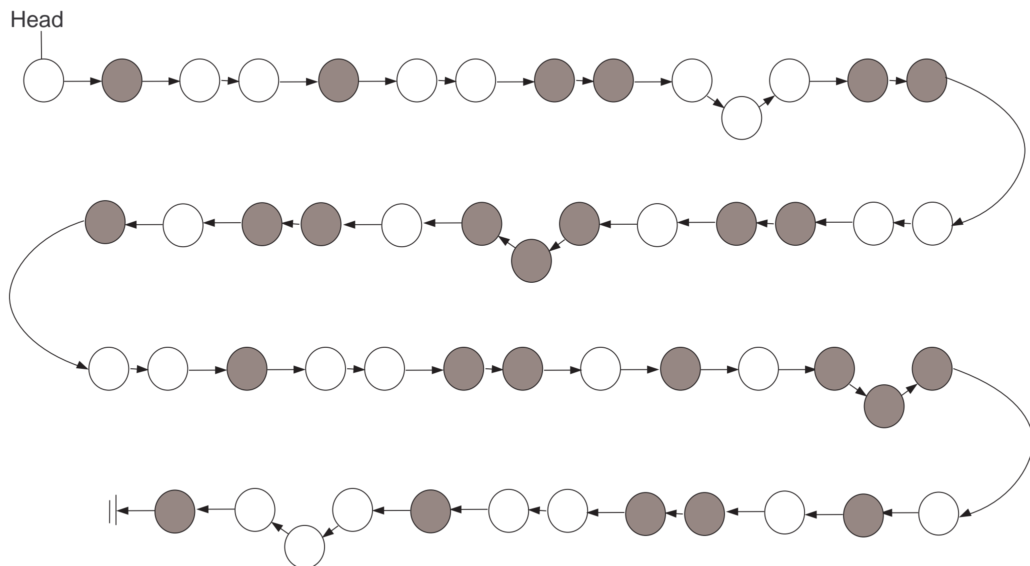

Fig. 4.7: A typical snapshot of the queue (16 processes)

5. CONCLUSIONS

5.1 Results

In this thesis I explored new techniques to improve the scalability of threads synchronization. Before this work there was no practical concurrent FIFO queue for which simultaneous enqueue method calls could be performed in parallel. The newly designed queue outperforms the best known concurrent queues in the literature in various benchmarks, even at low levels of concurrency.

Moreover, by integrating the basket-mechanism as the back-off mechanism, the time usually spent on backing-off is utilized by failed method calls to insert their items into the baskets, allowing enqueues to complete sooner. This leads to a queue algorithm that unlike all former concurrent queue algorithms requires virtually no tuning of the backoff mechanisms to reduce contention, making it an attractive out-of-the-box queue.

5.2 Future Work

Although the nodes of the same basket need not be ordered, they are inserted and removed in a stack-like manner, one by one. This unnecessary synchronization inside each basket limits the scalability of the algorithm. It is a subject for further research to determine if it feasible to exploit a weaker order semantics on insertion and removes to make the queue more scalable.

BIBLIOGRAPHY

[1] Cormen, T., Leiserson, C., Rivest, R., and Stein, C. *Introduction to Algorithms*, second edition ed. MIT Press, Cambridge, MA, 2001.

[2] Gottlieb, A., Lubachevsky, B. D., and Rudolph, L. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Trans. Program. Lang. Syst. 5*, 2 (1983), 164–189.

[3] Hendler, D., Yerushalmi, L., and Shavit, N. A scalable lock-free stack algorithm. Tech. Rep. TR-2004-128, Sun Microsystems Laboratories, 2004.

[4] Herlihy, M., and Wing, J. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems 12*, 3 (July 1990), 463–492.

[5] Hwang, K., and Briggs, F. A. *Computer Architecture and Parallel Processing.* McGraw-Hill, Inc., 1990.

[6] Ladan-Mozes, E., and Shavit, N. An optimistic approach to lock-free fifo queues. In *Proceedings of Distributed computing* (2004), Springer, pp. 117–131.

[7] Lamport, L. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems 5*, 2 (1983), 190–222.

[8] Lea, D. The java concurrency package (JSR-166). `http://gee.cs.oswego.edu/dl/concurrency-interest/index.html`.

[9] Mellor-Crummey, J. M. Concurrent queues: Practical fetch-and-ϕ algorithms. Tech. Rep. Technical Report 229, University of Rochester, November 1987.

[10] Michael, M., and Scott, M. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared - memory multiprocessors. *Journal of Parallel and Distributed Computing 51*, 1 (1998), 1–26.

[11] Moir, M. Practical implementations of non-blocking synchronization primitives. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing* (1997), pp. 219–228.

[12] Moir, M., Nussbaum, D., Shalev, O., and Shavit, N. Using elimination to implement scalable and lock-free fifo queues. In *SPAA '05: Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures* (New York, NY, USA, 2005), ACM Press, pp. 253–262.

[13] Prakash, S., Lee, Y.-H., and Johnson, T. Non-blocking algorithms for concurrent data structures. Tech. Rep. 91–002, Department of Information Sciences, University of Florida, 1991.

[14] Prakash, S., Lee, Y.-H., and Johnson, T. A non-blocking algorithm for shared queues using compare-and-swap. *IEEE Transactions on Computers 43*, 5 (1994), 548–559.

[15] Shavit, N., and Touitou, D. Elimination trees and the construction of pools and stacks. In *ACM Symposium on Parallel Algorithms and Architectures* (1995), pp. 54–63.

[16] Sites, R. *Operating Systems and Computer Architecture, In H. Stone, editor, Introduction to Computer Architecture, 2nd edition, Chapter 12.* Science Research Associates, 1980.

[17] Stone, H. S. *High-performance computer architecture.* Addison-Wesley Longman Publishing Co., Inc., 1987.

[18] Stone, J. A simple and correct shared-queue algorithm using compare-and-swap. In *Proceedings of the 1990 conference on Supercomputing* (1990), IEEE Computer Society Press, pp. 495–504.

[19] Stone, J. M. A nonblocking compare-and-swap algorithm for a shared circular queue. In *Parallel and Distributed Computing in Engineering Systems* (1992), Elsevier Science B.V., pp. 147–152.

[20] Treiber, R. K. Systems programming: Coping with parallelism. Tech. Rep. RJ 5118, IBM Almaden Research Center, April 1986.

[21] Tsigas, P., and 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* (2001), ACM Press, pp. 134–143.

[22] Valois, J. Implementing lock-free queues. In *Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems* (1994), pp. 64–69.

[23] Valois, J. D. Lock-free linked lists using compare-and-swap. In *Symposium on Principles of Distributed Computing* (1995), pp. 214–222.