A Non-Blocking Concurrent Queue Algorithm

Bruno Didot
`bruno.didot@epfl.ch`

June 2012

Abstract

This report presents a new non-blocking concurrent FIFO queue backed by an unrolled linked list. Enqueue and dequeue operations can be run concurrently, without misbehaving.

Contents

1 Introduction

Many applications take advantage of multi-core computer architectures by executing multiple tasks concurrently, often accessing shared data concurrently. Correctness of these data structures is ensured by synchronizing concurrent accesses. There are generally two approaches to synchronization; mutual exclusion locks and lock-free algorithms. Locks are more traditional but do not come without any issue. On asynchronous, concurrently programmed, multi-core systems, a thread holding a lock might get delayed (e. g. by getting preempted by the scheduler or because of a page fault) or halted, making other threads wait until the lock is freed, thus preventing any progress to be made by those other threads. Similarly, if a slow thread gets the lock first, other faster threads will have to wait indefinitely for the lock to be released. More importantly, a failure in a thread holding a lock might lead other threads to wait indefinitely. In case of such events, lock-free algorithms are more robust.

A lock-free concurrent data structure guarentees that if several threads are trying to perform operations on a shared data strucutre, some operation will complete within a finite number of steps. As such, operations on lock-free data structures are immune to deadlocks, do not get delayed by slower threads nor by preemption on other threads.

Many implementations of lock-free concurrent FIFO queues have already been proposed (see Hwang and Briggs, Sites, Michael and Scott). All of them rely on a linked list to achieve their goal. An unrolled linked list is slightly different from a standard linked list as each node stores multiple elements. This increases cache performance and decreases the memory overhead associated with storing a reference to the next node in the list for each node of the list and and allocating a new node for each element in the list. Therefore, we hereby propose a FIFO queue based on an singly linked unrolled list.

2 Algorithm

The algorithm implements the queue as an unrolled singly linked list with *head* and *tail* pointers. *head* always points to a dummy node, which is the first node in the list. *tail* points to a node in the list. Tail is used to find the reference to the last node in the list without having to go through the entire list for each *enqueue* operation. Unlike a standard linked list, whose nodes store a single element, each node of an unrolled list stores multiple elements using an array.

The usage of a dummy node at the beginning of the list has first been proposed by Valois, and allows easy and consistent maintenance of *head* and *tail* pointers. Using this node, we can guarentee that *tail* will never fall behind *head*; it will never point to a node that is not accessible through the *head* pointer. It also guarentees that both *head* and *tail* pointer will never be set to *null*.

In this implementation, elements are only enqueued at the first null slot in the unrolled list (the $DELETED$ flag being non null). "First" null slot is intuitively defined as the array slot with the lowest index whose value is null, provided that all *elements* array slots in all nodes before the considered node are non null. A *dequeue* operation will only delete the first non $DELETED$ element reachable from *head.next*. The special $DELETED$ flag, which has first been proposed by Aleksandar Prokopec, has been introduced to distinguish two different states. Without the $DELETED$ flag, node elements can only be deleted by being set to null. Therefore, operations on a queue can not distinguish an array slot that is null because it has never been set to any other value, with an array slot that is null because it previously held an element that has been deleted. This would result in an inconsistency with the *enqueue* definition.

An hint has been introduced for both *enqueue* and *dequeue* operations. Since elements are only deleted from the beginning of an *element* array, the position of newly deleted elements in the array is always increasing. Therefore, $deleteHint$ has been added to prevent the loop that looks for the next element to be dequeued from checking all the slots in the array. $deleteHint$ is then increased by the thread that has just finished dequeuing an element. The variable $addHint$ behaves similarly.

Pseudocode 1 Data structures and initialization

```
head: Node
tail: Node
NODE_SIZE = integer constant
DELETED = flag constant

structure Node {
  next: Node
  elements: Array[NODE_SIZE]
  addHint: Int
  deleteHint: Int

  def Node(elem) =
    elements[0] = elem
    addHint = 1

}

def init() = {
  head = new Node()
  head.elements[0 until NODE_SIZE] = DELETED
  tail = head
}
```

3 Correctness

Definition 1 (Basics)**.** An unrolled node (from now on node) is a structure holding a reference *next* to another node and an array of references *elements* to its elements.

Definition 2 (Unrolled queue)**.** An **unrolled queue** is defined as the reference *head* to the root of the list. An unrolled queue **state** $\mathbb{S}$ is defined as the sequence of nodes reachable from *head*. Nodes that are not reachable anymore are considered as deleted. The set of reachable nodes from a certain *node* can be expressed as:

$$reachables(node) = \begin{cases} \{node\} \cup reachables(node.next) & \text{if } node \neq null \\ \emptyset & \text{otherwise} \end{cases}$$

Pseudocode 2 Enqueue operation

```
def enqueue(elem) = {
  if (elem == null) {
    throw ERROR
  }

  loop {
    val t = tail
    val n = t.next
    if (n == null) {
      var i = t.addHint
      while (i < NODE_SIZE and t.elements[i] != null) {
        i += 1
      }

      if (i < NODE_SIZE) {
        if (CAS(t.elements, i, null, elem)) {
          t.addHint = i + 1
          return
        }
      } else {
        val n_ = new Node(elem)
        if (CAS(t.next, null, n_)) {
          CAS(tail, t, n_)
          return
        }
      }
    } else {
      CAS(tail, t, n)
    }
  }
}
```

The values held by the array inside each consecutive node of a sequence of nodes can be concatenated into a string as:

$$elems(node) = \begin{cases} node.elements \ \cdot \ elems(node.next) & \text{if } node \neq null \\ \varepsilon & \text{otherwise} \end{cases}$$

We also introduce a relation to strip down leading $DELETED$ and trailing $null$ values of a string:

$$core(s : String) = E^e \text{ if } s = DELETED^d \cdot E^e \cdot null^n, \text{ where } d, e, n \geq 0$$

Definition 3. We define the following invariants for the unrolled queue.
INV 1 $head \neq null$
INV 2 $null \notin head.elements$

INV 3 $tail \in reachables(head)$
INV 4 $node \notin reachables(node.next)$
INV 5 $elems(head.next) = DELETED^d \cdot E^e \cdot null^n$ where $d, e, n \geq 0, E \in set\ of\ enqueued\ elements$

Definition 4 (Validity)**.** A state $\mathbb{S}$ is valid if and only if the invariants INV 1-5 are true in the state $\mathbb{S}$.

Definition 5 (Abstract queue)**.** An abstract queue $\mathbb{Q}$ is a string of elements $e_0 \cdot e_1 \cdot ... \cdot e_n$, where $n \geq 0$. An empty abstract queue is defined by the empty string ε. Abstract queue operations are $enqueue(\mathbb{Q}, e) = \mathbb{Q} \cdot e$ and $dequeue(\mathbb{Q}) = \mathbb{Q}_1 : \mathbb{Q}_1 = e_1 \cdot e_2 \cdot ... \cdot e_n$ if $\mathbb{Q} = e_0 \cdot e_1 \cdot ... \cdot e_n$. Operations *enqueue* and *dequeue* are destructive.

Definition 6 (Consistency)**.** An unrolled queue state $\mathbb{S}$ is consistent with an abstract queue $\mathbb{Q}$ if and only if $\mathbb{Q} = core(elems(head.next))$. A destructive unrolled queue operation *op* is consistent with the corresponding abstract queue operation op' if and only if applying *op* to a state $\mathbb{S}$ consistent with $\mathbb{Q}$ changes the queue into $\mathbb{S}'$ consistent with an abstract queue $\mathbb{Q}' = op(\mathbb{Q}\ [, k])$.

3.1 Safety

Safety means that the unrolled queue corresponds to some abstract queue and that all operations change the corresponding abstract queue consistently.

Theorem 1 (Safety)**.** At all times t, an unrolled queue is in a valid state $\mathbb{S}$, consistent with some abstract queue $\mathbb{Q}$. All unrolled queue operations are consistent with the semantics of the abstract queue $\mathbb{Q}$.

Trivially, if the state $\mathbb{S}$ is valid, then the unrolled queue is also consistent with some abstract queue $\mathbb{Q}$. We prove the theorem using induction. Initially, when the unrolled queue is empty, all the invariants hold: the unrolled queue is valid and consistent. The induction hypothesis is that the unrolled queue is valid and consistent at some time t. Implicitly using induction hypothesis, we show that they continue to hold.

Lemma 1. INV 1 — $head \neq null$

The only assignment instruction on *head* occurs in the *dequeue* operation, line 62. This instruction only changes its value to the next node, atomically. The next node could not be *null* because if there is only one node in the unrolled list, $head = tail$ and the *dequeue* operation does not complete. The node *head* used to point to is considered as deleted.

Lemma 2. Once an array slot is set to a valid non null values, it is never set to any other value than $DELETED$

Array slot assignments are happen only at l. 16, l. 57 and l. 63. The atomic assignment at l. 16 is only successful when the corresponding slot equals *null* (which obviously is not the case here). The assignment operation at l. 57 only sets values to $DELETED$. The assignment at l. 63 affects only sets values to $DELETED$.

Lemma 3. For each node, $elements[i] = DELETED, \forall\, 0 \leq i < deleteHint$

$deleteHint$ is only modified in the *dequeue* operation, at l. 57. The $deleteHint$ value is only increased to the value $i+1$ after checking that elements from index $deleteHint$ to index $i-1$ are set to $DELETED$. The CAS operation at l. 57 will only succeed if $elements[i]$ has been set to $DELETED$ by the current thread. Therefore, if array slots that have been set to $DELETED$ are never set to any other values than $DELETED$, the lemma holds. Which is just what lemma 2 guarentee us.

Lemma 4. INV 2 — $null \notin head.elements$

The initial node *head* does not contain *null* values. *head* is only advanced to its next node at l. 62. Using lemmas 2 and 3, and considering the loop at l. 48 - 50, we can say that all values, of this next node, prior to $NODE\_SIZE-1$ have been checked to be set to $DELETED$ when CASing the head. $elements[NODE\_SIZE-1] \neq null$ otherwise the if at l. 52 would have succeedded. The subsequent delete operation at l. 63 also sets the array slot to a non null value. Therefore, the lemma holds.

Lemma 5. INV 3 — $tail \in reachables(head)$

tail always points to a node, reachable from *head*, in the unrolled list, because: in the *dequeue* operation, *head* never advances without checking that *tail* should be first set to the next node in the unrolled list. So *tail* never references a deleted node. Also, *tail* is never set to *null* as l. 38-39 ensures that the next node is never set to *null*. In the *enqueue* operation, *tail* is CASed to a node reachable from the unrolled list (as the insertion of a new node at l. 22 has succeeded), or the the next non null node in the unrolled list (l. 28).

Lemma 6. INV 4 — $node \notin reachables(node.next)$

Only new nodes are atomically added to the unrolled list (l. 21-22), so an existing node can not be added twice.

Lemma 7. For each node, $elements[i] \neq null, \forall\, 0 \leq i < addHint$

Using lemma 2, and considering the loop at l. 11-13, we can say that all slots prior to index i are set to a non null value. When addHint is increased at l. 17, an element has successfully been enqueued at index i, therefore all slots prior index $i+1$ are set to non null values.

Lemma 8. Enqueue operation is consistent: it only adds an element at the end of the unrolled queue

Elements are only added to a node that has a *null next* reference (first condition of enqueue). The only such node is the last node of the unrolled queue (as the *next* reference of nodes is never set to *null*). Using lemma 7, considering the loop at l. 11-13 and using lemma2, we can conclude that when enqueuing an element at l. 16 and when adding a new node to the unrolled list at l. 22, all previous slots are known to be non null. Therefore, in the *elements* array, an element is only added at the first non null slot value. When no null slot is available (l. 20), a new node is created and is added atomically to the last node of the unrolled list. Using the induction hypothesis, and the now proven fact that an element is only added to the first non null reference, we showed that enqueue is consistent.

Lemma 9. Dequeue operation is consistent: it only removes an element from the beginning of the unrolled queue

Elements are only deleted from *head.next* (delete CAS only occurs on *nh*, which points to *head.next*). Using lemma 3, considering the loop l. 48-50 and using lemma 2, we can conclude that when dequeueing an element at l. 57 and when CASing the head at l. 62, all previous array slots are set to $DELETED$. If the dequeued element is at the last position of the array, *head* is atomically swapped to the next node in the list. This next node is considered as deleted, and as such, the element has been dequeued.

Corollary. INV 5 — (Essentially $elems(head.next) = DELETED^d \cdot E^e \cdot null^n$)

Using lemmas 8 and 9 and using the induction hypothesis, INV 5 immediately falls.

The invariants always holds and both enqueue and dequeue operations are consistent. Safety is guarenteed.

3.2 Linearizability

An operation is *linearizable* if any external observer can only observe the operation as if it took place instantaneously at some point between its invocation and completion.

In our case, our operations are linearizable because there is a unique point during each operation at which it is considered to "take effect". An enqueue takes effect when:

- the element is CASed into the *element* array of the last node **if** a null slot is available in the array
- a newly allocated node, holding a reference to the enqueued value, is linked to the last node in the unrolled list **if** no non null slots are available in the array.

A dequeue operation takes effect when:

- the element is replaced using a CAS by a $DELETED$ flag **if** the element was located before the last slot of the *elements* array
- *head* swings to the next node in the list **if** the element was located at the last slot of the *elements* array.

3.3 Lock-freedom

Lock-freedom means that if some number of threads execute operations concurrently, then after a finite number of steps some operation must complete.

An enqueue operation loops only if the condition in line 9, the compare and swap at line 16, xor the compare and swap at line 22 fails.

A dequeue operation loops only if the condition at line 38 holds (and that the list contains more than one node), xor the CAS at line 57, or the CAS at line 62, or both conditions in lines 56 and 61 fail.

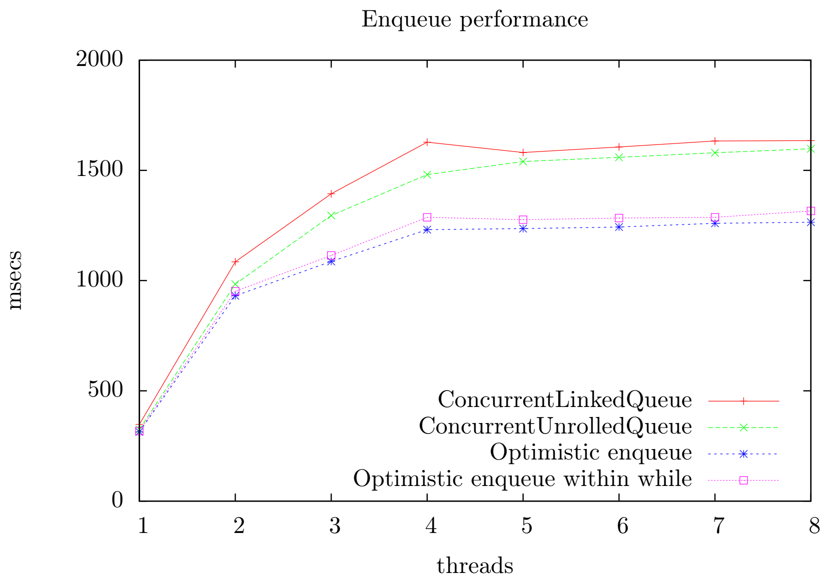
We show that the algorithm is lock-free by showing that a process loops beyond a finite number of times only if another process completes an operation on the queue.

4 Performance

As discussed in the introduction, one of the interest of lock-free data structures resides in performance. We introduce benchmarks of several slightly different implementations an unrolled queue queue, for both enqueue and dequeue operations. We also introduce benchmarks of the reference Java implementation. Benchmarks were ran on a Core i5 2500k machine, with 8GiB of memory. We use Oracle's implementation of the JVM, version 1.7.5, and use the Linux kernel 3.4.2. The JVM is given 4GiB of memory at the start. Tests were ran 20 times, and we take the average of the last 18 values to estimate the performance.

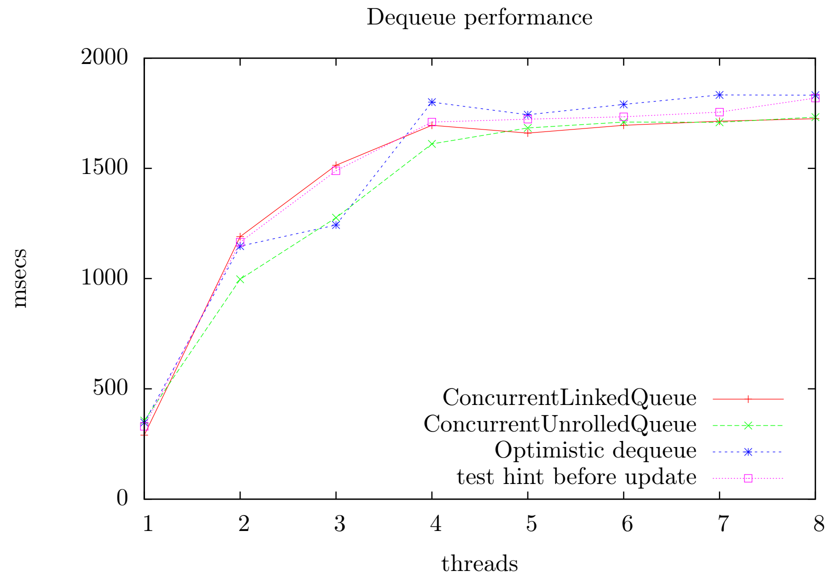
In all graphs, "ConcurrentLinkedQueue" refers to the Java implementation that can be found in package java.util.concurrent.ConcurrentLinkedQueue. "ConcurrentUnrolledQueue" refers to the implementation of described in the pseudocode. All tests work on 20'000'000 elements.

Two slightly modified implementations of the enqueue operation are also featured in the following "Enqueue performance" graph. "Optimistic enqueue" is a slightly modified version of the enqueue algorithm, where we try to add a value to the queue, assuming that tail points to the last node of the list, and assuming that $addHint$ points to the end of the list. If those assumptions are not correct, the algorithm procedes to the standard enqueue. "Optimistic enqueue within while" features a similar behaviour, but the optmistic behaviour is added inside the while loop. Therefore, for each time that an enqueue fails, we will try the optimistic behaviour before proceding to the standard behaviour.



From the graph, it follows that an enqueue in an unrolled queue is slightly faster than in the standard linked queue. This is to be expected, as nodes are being allocated less frequently, and the unrolled queue is making more use of the cache. The optimistic behaviour for the enqueue results in even better performance than the standard implementation of the unrolled queue.

Two slightly modified implementations of the dequeue operation are also featured in the following “Dequeue performance” graph. “Optimistic dequeue” is based on the same idea as “Optimistic enqueue”. “test hint before update” is an implementation of enqueue were the deleteHint is tested before being updated.



All dequeue operations perform similarly to the reference Java implementation. The performance of the ConcurrentUnrolledQueue's dequeue operation degrades less quickly than the performance of the reference ConcurrentLinkedQueue implementation, but reaches about the same performance in the end. The optimistic brings nothing in this case. This might be due to the fact that more variable reads are needed to perform a safe optimistic dequeue. It also seems to be much less stable than all other queue implementation (see performance when running with 3 and 4 threads).

All implementations can be found at github.com/axel22/concurrent-unrolled-queue/queue_implementations.

References

[1] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing.* McGraw-Hill, 1994.

[2] J. D. Valois. Lock-Free Data Structures. Ph. D. dissertation, Rensselear Polytechnic Institute, May 1995.

[3] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing.* McGraw-Hill, 1994.

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

[5] M. M. Michael and M. L. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms, 1996.

[6] Java 6 API and implementation, http://docs.oracle.com/javase/6/docs/api/.

Pseudocode 3 Dequeue operation

```
def dequeue() = {
  loop {
    val h = head
    val nh = h.next
    val t = tail

    if (h == t) {
      if (nh == null) {
        return EMPTY
      }

      CAS(tail, t, nh)
    } else {
      var i = nh.deleteHint
      var v = null

      while (i < NODE_SIZE and { v = nh.elements[i]; v == DELETED }) {
        i += 1
      }

      if (v == null) {
        return EMPTY
      }

      if (i < NODE_SIZE - 1) {
        if (CAS(nh.elements, i, v, DELETED)) {
          nh.deleteHint = i + 1
          return v
        }
      } else if (i == NODE_SIZE - 1) {
        if (CAS(head, h, nh)) {
          nh.elements[NODE_SIZE - 1] = DELETED
          return v
        }
      }
    }
  }
}
```