BitNext - A Lock-Free Queue

Pedro Ramalhete
Cisco Systems
pramalhe@gmail.com

Andreia Correia
Concurrency Freaks
andreiacraveiroramalhete@gmail.com

ABSTRACT

The best known lock-free Multi-Producer-Multi-Consumer (MPMC) queue is the algorithm by Maged Michael and Michael Scott (MS), which is itself the basis of many other lock-free and wait-free queue algorithms. The MS queue has a simple design and is efficient under low contention, but under high contention can have low throughput and high tail latency.

We present BitNext queue, a singly-linked list lock-free linearizable MPMC queue, based on the idea of the lock-free list by Tim Harris, of having a marked bit in the `next` pointer. We compare BitNext versus MS queue using standard benchmarks of throughput and latency, and we show that under high contention our BitNext queue has a much lower tail latency, and a higher throughput of up to 3x.

Categories and Subject Descriptors

D.4.1.f [**Operating Systems**]: Synchronization

Keywords

queues; non-blocking; lock-free

1. INTRODUCTION

Concurrent linearizable queues are one of the simplest data structures used in multi-threaded applications. If multiple threads can call the methods `enqueue()` and `dequeue()`, the queue is said to be Multi-Producer-Multi-Consumer (MPMC).

When it comes to lock-free MPMC singly-linked list based queues, the best known is the one by Maged Michael and Michael Scott (MS queue) [3]. When running uncontended, the enqueue method is capable of inserting a new node using just two Compare-And-Swap (CAS) instructions, and dequeuing an item with a single CAS. Its simple design and low overhead make it extremely attractive to practitioners and researchers alike, so much so that it is the algorithm used in Java's ConcurrentLinkedQueue, and the basis for the fast lock-free queue LCRQ [4]. Unfortunately, the throughput and tail latency of MS queue degrade under high contention.

On the next section we present BitNext, an alternative to the MS queue that has better throughput for the enqueue operation, and better tail latency guarantees.

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

2. ALGORITHM

The main idea behind BitNext is inspired by the lock-free list of Tim Harris [1], and it consists of marking the `next` of each node with a bit to represent this node has been dequeued, and new nodes don't need to be inserted after the current tail. As shown in Algorithm 1 a node contains two members, `item` and `next`.

Dequeuers start by advancing `head` only if its `next` is marked. If the `next` is not marked, the dequeuer will attempt to mark the bit with a CAS. A node with a marked bit is said to be *logically* dequeued. If the CAS to set the marked bit on `node.next` fails, it means another dequeuer was successful, or an enqueuer inserted a new node immediately after `node`.

For enqueuers there are two scenarios, both starting with the enqueuer reading the current `tail` into the local variable `ltail`. When the `ltail` is the last node on the list, `ltail.next` is null, then, similarly to the MS queue, the enqueuer will attempt to insert its node by doing a CAS on `ltail.next` from null to its node. When the `ltail` is not the last node on the list, the enqueuer will advance the `tail` to the next node, `ltail.next`, and then attempt to insert its node in `ltail.next` with a CAS. This attempt will be done at most `maxThreads-1` times, after which, either the

Algorithm 1 Node

```
struct Node {
    T* item;
    std::atomic<Node*> next;
    Node(T* item) : item{item}, next{nullptr} { }
};
```

Algorithm 2 Enqueue algorithm with Hazard Pointers

```
1  void enqueue(T* item) {
2    if (item == nullptr) throw std::invalid_argument("null_item");
3    Node* myN = new Node(item);
4    while (true) {
5      Node* ltail = hp.protectPtr(kHpTail, tail.load());
6      if (ltail != tail.load()) continue;
7      Node* lnext = ltail->next.load();
8      if (getUnmark(lnext) != nullptr) {
9        casTail(ltail, getUnmarked(lnext));
10     } else {
11       for (int i=0; i < 2; i++) {
12         myN->next.store(nullptr, std::memory_order_relaxed);
13         Node* node = isMark(lnext) ? getMark(myN) : myN;
14         if (ltail->next.compare_exchange_strong(lnext, node)) {
15           casTail(ltail, myN);
16           return;
17         }
18         lnext = ltail->next.load();
19         if (getUnmark(lnext) != nullptr) {
20           casTail(ltail, getUnmark(lnext));
21           break;
22         }
23       }
24     }
25     for (int i = 0; i < maxThreads−1; i++) {
26       lnext = ltail->next.load();
27       if (isMark(lnext)) break;
28       myN->next.store(lnext, std::memory_order_relaxed);
29       if (ltail->casNext(lnext, myN)) return;
30     }
31   }
32 }
```

CAS is successful, or the `ltail` node has been logically dequeued, thus disallowing an insertion in `ltail.next`. If the enqueuer were to insert a node in `ltail.next` even if it had a marked bit, this would provide linearizable behavior as long as the next node had not been logically dequeued. Given that we can't provide such a guarantee, we forbid the enqueuer from inserting a node after a logically dequeued node, unless `node.next` is null, in which case there is no node after it.

The C++ source code for `enqueue()` and `dequeue()` are shown in Algorithms 2 and 3 respectively. There is a two-iteration `for` loop starting in line 11 of Algorithm 2. This is needed due to the possibility of inserting a node after `tail` when `tail` is the last node on the queue. As stated before, independently of the last node being marked or not, an enqueuer is always allowed to insert a node in those conditions. An ongoing insertion at the end of the queue may have to try two times to insert a node after tail, in case tail is deleted.

Although counter-intuitive, the behavior of the queue is FIFO (First-In-First-Out) and linearizable. By reading `tail` into `ltail` and having each enqueuer insert its node after `ltail`, we provide two guarantees: First, a node inserted in the list before `ltail` must have been inserted by an enqueuer that started earlier, which saw a previous node as its `ltail`. For this ealier enqueuer we can choose a linearization point which occurred before the current enqueuer started; Second, all enqueuers which saw the same `tail`, can insert their nodes in any order among themselves.

All enqueuers that read the same tail node are guaranteed to have had their operations overlap in time, and therefore, any order among them provides a linearizable history. All these enqueuers will try to insert immediately after `ltail` and some may see `ltail.next` as null, while others will see it as non-null. The first node that is inserted after `tail` (when `tail.next` was null) will be the next tail, and all enqueuers will guarantee that `tail` has advanced before inserting their own node between `ltail` and the new `tail`.

To guarantee FIFO order and linearizability in our singly-linked list backed queue, an enqueuer must advance the `tail` up to his node or to a later node. Advancing the `tail` can be done *before* the insertion or *after* the insertion. Advancing after the insertion is a more intuitive approach and it is how the MS queue works, however, the goal of our algorithm was to have multiple insertions with a single tail advance. If the advance of `tail` were to occur after insertion of the node, it would require traversing the nodes of the queue from `ltail` to the last node, which would be costly. In BitNext, when the tail is already the last node, we use an approach like MS, inserting the node and then advancing the `tail`, however, when the `tail` is not the last node, we advance `tail` before inserting the node.

For read-only traversals of the queue, BitNext does not provide a linearizable traversal because there may be on-going enqueuers inserting nodes between `head` and `tail`. Conversely, the MS queue is capable of providing linearizable read-only iterations if and only if, after reading `head`, the previously read `tail` has not changed and the traversal is done between those nodes.

One way to optimize dequeuers is to update the head in a *lazy* way. During a dequeue, instead of advancing the `head` whenever a node with a marked `next` is found (line 8 of Algorithm 3), we can skip the advance until a node with a non-marked bit is found and we successfully set the marked bit (lines 21 and 23 of Algorithm 4). This approach reduces contention on the `head` variable, which improves throughput and tail la-

Algorithm 3 Dequeue algorithm with Hazard Pointers

```
1  T* dequeue(void) {
2    while (true) {
3      Node* lhead = hp.protectPtr(kHpHead, head.load());
4      if (lhead != head.load()) continue;
5      Node* lnext = lhead->next.load();
6      if (isMark(lnext)) {
7        if (getUnmark(lnext) == nullptr) return nullptr;
8        if (casHead(lhead, getUnmark(lnext))) hp.retire(lhead);
9        continue;
10     }
11     if (lhead->casNext(lnext, getMark(lnext))) {
12       return lhead->item;
13     }
14   }
15 }
```

Algorithm 4 Lazy Head Dequeue with HP

```
T* dequeue(void) {
  while (true) {
    Node* lhead = hp.protectPtr(kHpHead, head.load());
    if (lhead != head.load()) continue;
    Node* lcurr = lhead;
    for (int i = 0; ;) {
      Node* lnext = lcurr->next.load();
      if (lnext == getMark(nullptr)) {
        if (lhead != lcurr && casHead(lhead, lcurr)) {
          retireSubList(lhead, lcurr);
        }
        return nullptr; // Queue is empty
      }
      if (isMark(lnext)) {
        hp.protectPtr(kHpNext+(i&0x1), getUnmark(lnext));
        if (lhead != head.load()) break;
        lcurr = getUnmark(lnext);
        i++;
        continue;
      }
      if (!lcurr->casNext(lnext, getMark(lnext))) continue;
      T* item = lcurr->item;
      if (lcurr != lhead && casHead(lhead, lcurr)) {
        retireSubList(lhead, lcurr);
      }
      return item;
    }
  }
}

void retireSubList(Node* start, Node* end) {
  for (Node* node = start; node != end; ) {
    Node* lnext = getUnmark(node->next.load());
    hp.retire(node);
    node = lnext;
  }
}
```

tency for dequeues. Algorithm 4 shows this variant's C++ code as BitNextLazyHead.

3. THROUGHPUT AND LATENCY

Using an AMD Opteron 6272 server with a total of 32 cores, we executed two different benchmarks following a procedure similar to [5]. The single-enqueue-single-dequeue benchmark is shown in figure 1, with the right-side plot showing the ratio normalized to the MS queue, with BitNext having a 3x higher throughput for enqueueing. Figure 2 shows the burst benchmark. To demonstrate the effects of high contention, our benchmark does not have a random sleep in between each operation. These effects are particularly visible on the latency plot of the 99.99% quantile in figure 3, where the BitNextLazyHead latency can be 25x better for both dequeues and enqueues.

4. CONCLUSION

In the known literature, most lock-free and wait-free queues backed by singly-linked list are based on the MS queue. BitNext requires marking a bit in the `next` pointer of the node, which is not possible in languages with a Garbage Collector, but just like for Harris list, there are efficient alternatives like RTTI [2] or AtomicMarkableReference. Compared to MS queue, BitNext requires an extra atomic operation for the dequeue due to the added constraint of marking a bit in the `next` pointer of the node. However, BitNext's natural *spreading* of contention on the queue improves throughput and reduces latency at the tail of the distribution. Although there may exist faster queues (like SimQueue or LCRQ) with similar low latency, BitNext has a smaller memory footprint, making it suitable for systems with low RAM and near real-time constraints, for example, small networking devices.

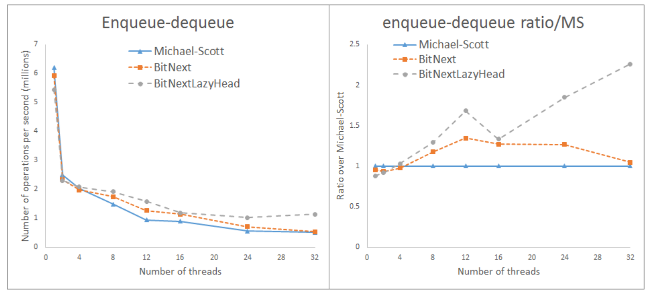

Figure 1: Single-enqueue-single-dequeue

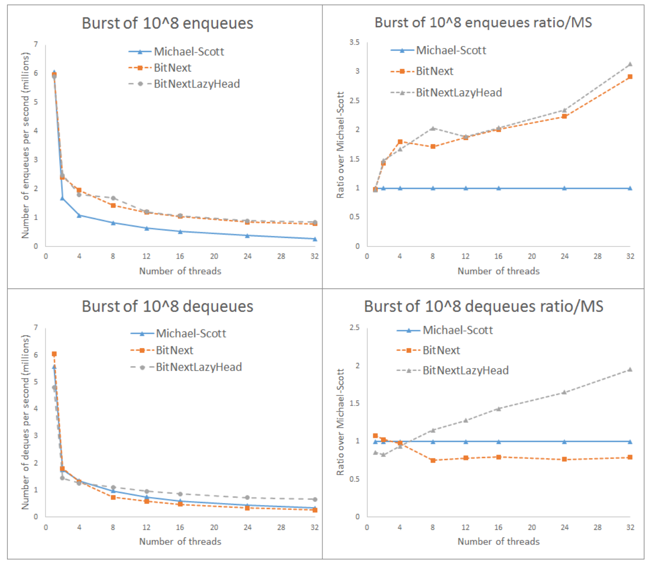

Figure 2: Burst. Higher is better.

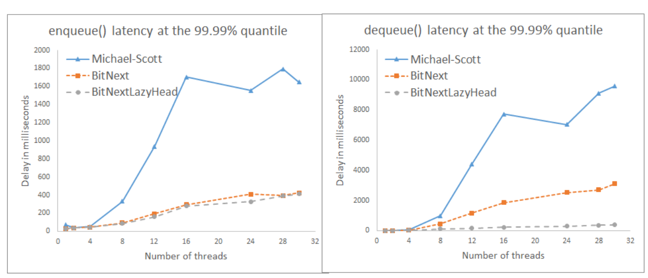

Figure 3: 99.99% latency. Lower is better.

5. REFERENCES

[1] Harris, T. L. A pragmatic implementation of non-blocking linked-lists. In *International Symposium on Distributed Computing* (2001), Springer, pp. 300–314.

[2] Heller, S., Herlihy, M., Luchangco, V., Moir, M., SCHERER III, W. N., and Shavit, N. A lazy concurrent list-based set algorithm. *Parallel Processing Letters 17*, 04 (2007), 411–424.

[3] Michael, M. M., and Scott, M. L. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing* (1996), ACM, pp. 267–275.

[4] Morrison, A., and Afek, Y. Fast concurrent queues for x86 processors. In *ACM SIGPLAN Notices* (2013), vol. 48, ACM, pp. 103–112.

[5] Ramalhete, P., and Correia, A. A Wait-Free Queue with Wait-Free Memory Reclamation. https://github.com/pramalhe/ConcurrencyFreaks/papers/crturnqueue-2016.pdf, 2016.