DoubleLink - A Low-Overhead Lock-Free Queue

Pedro Ramalhete
Cisco Systems
pramalhe@gmail.com

Andreia Correia
Concurrency Freaks
andreiacraveiroramalhete@gmail.com

ABSTRACT

In 1996, Maged Michael and Michael Scott published a simple and elegant lock-free queue based on a singly-linked list (MS queue). Up to this day, this is the most widely deployed linearizable memory-unbounded Multi-Producer-Multi-Consumer lock-free queue. It is used in the industry, being part of Java's JDK, and in academia, as the basis of other lock-free and wait-free queues.

For non-managed languages (C/C++), it is easy to add memory reclamation to the MS queue using an Hazard Pointers technique (HP). Without contention, the `enqueue()` requires two Compare-And-Swap (CAS) operations to complete plus one sequentially-consistent store for the HP, and the `dequeue()` requires a single CAS plus two sequentially consistent stores for the HP.

We present DoubleLink, a new linearizable lock-free queue based on a double-linked list, with lower overhead than the MS queue, and properties similar to MS except the memory usage due to the extra pointer. Its `enqueue()` requires one CAS plus one sequentially-consistent store for the HP, and the `dequeue()` requires ones CAS plus one sequentially consistent store for the HP. Due to its low overhead, DoubleLink equals or surpasses the MS queue in all our benchmarks, going up to 1.3x the throughput, making this an interesting alternative to the MS queue.

Categories and Subject Descriptors

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

Keywords

queues, lock-free

1. INTRODUCTION

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

The best known singly-linked list lock-free queue is the algorithm created by Maged Michael and Michael Scott [3] (MS queue). There are many algorithms for linearizable queues that provide lock-free and wait-free progress for either `enqueue()` or `dequeue()` or both, but none as are simple as the MS queue. Its simplicity and elegance make it extremely attractive to both practitioners and researchers alike. The MS queue algorithm is used in Java's ConcurrentLinkedQueue, and is the basis for other lock-free queues like LCRQ [5], and even wait-free queues [1, 6]. Moreover, it has low memory usage, requiring one node with two pointers per item in the queue, and it is easy to add memory reclamation to it with Hazard Pointers [2].

When deploying MS with Hazard Pointers (HP), an uncontended call to `enqueue()` requires two Compare-And-Swap (CAS) operations to complete plus one sequentially consistent (seq-cst) store for the HP, with one CAS adding the new node to the last node, and the other CAS advancing the `tail`. The seq-cst store is needed for the HP to protect the last node from being deleted while its `next` member is being dereferenced. An uncontended call to `dequeue()` requires a single CAS to complete, and two seq-cst stores for the HP, with the CAS used to advance the `head`, therefore dequeuing the current head node from the queue. One seq-cst store protects the current `head` node, and the other seq-cst store protects the next node, from which the `item` will be dereferenced.

On the next section we present DoubleLink, an alternative to the MS queue with lower synchronization overhead and therefore higher throughput.

2. ALGORITHM

In our algorithm each node has: one pointer to the item, one pointer to the previous node, and one atomic pointer to the next node. Algorithm 1 shows the C++ code for `enqueue()` with Hazard Pointers [2], using the API currently being proposed to the C++ standard library [4].

The main idea behind DoubleLink is the usage of a doubly-linked list, following an approach similar to the Triber stack [7]. An `enqueue()` starts by reading the

Algorithm 1 Enqueue algorithm with Hazard Pointers

```
1  void enqueue(T* item) {
2    if (item == nullptr) throw std::invalid_argument("null_item");
3    Node* newNode = new Node(item);
4    while (true) {
5      Node* ltail = hp.get_protected(tail);
6      Node* lprev = ltail->prev; // lprev is protected by hp
7      newNode->prev = ltail;
8      if (lprev->next.load() == nullptr) {
9        lprev->next.store(ltail, std::memory_order_relaxed);
10     }
11     if (casTail(ltail, newNode)) {
12       ltail->next.store(newNode, std::memory_order_release);
13       hp.clear();
14       return;
15     }
16   }
17 }
```

current `tail` and create a new node whose `prev` is pointing to the current node (line 7 of Algorithm 1) and then make sure that the previous node is pointing to the current tail in case the enqueuer that inserted it got delayed (lines 8 and 9). Then, it tries to replace the current `tail` with its node executing a CAS (line 11) and if successful, set the previous tail's `next` to point to the current tail (line 12).

A `dequeue()` uses an algorithm similar to the one on MS queue, where it attempts to advance the current `head` to the next node with a CAS. However, as we will see in the next section, due to having a `prev` in each node and a special variant of Hazard Pointers, this algorithm requires a single seq-cst store to protect the two contiguous nodes, i.e. the `head` and the next node.

Algorithm 2 Dequeue algorithm with Hazard Pointers

```
1  T* dequeue() {
2    while (true) {
3      Node* lhead = hp.get_protected(head);
4      Node* lnext = lhead->next.load(); // lnext is protected by hp
5      if (lnext == nullptr) {
6        hp.clear();
7        return nullptr; // Queue is empty
8      }
9      if (casHead(lhead, lnext)) {
10       T* item = lnext->item;
11       hp.clear();
12       hp.retire(lhead, tail.load());
13       return item;
14     }
15   }
16 }
```

There are two innovations in DoubleLink. The first innovation is the novel algorithm that allows to enqueue an item in the doubly-linked list using a single CAS and one relaxed atomic store. Multiple threads may *race* on this store, but as long as atomicity is guaranteed, no fence/barrier is needed to perform this operation, which means it has low synchronization overhead.

The second innovation is the usage of a variant of Hazard Pointers that enables the reduction of the number of sequentially consistent stores done to protect the nodes that will be accessed. This will be shown in the next section.

3. MEMORY RECLAMATION

When implementing the DoubleLink queue in a non-managed language like C or C++, some kind of memory reclamation technique is required. We chose Hazard Pointers (HP) because it is capable of maintaining lock-free progress. Instead of using it as shown originally by Maged Michael [2], we made a variant that allows DoubleLink to protect three nodes with a single sequentially consistent (seq-cst) store, namely, the current node, the previous node, and the next node. A seq-cst store is typically a heavyweight synchronization operation, for example, on x86 it implies one `MFENCE` instruction, and therefore, any reduction in the amount of seq-cst stores will reflect itself in a throughput gain.

In HP, after a node is placed in the retired list, to determine if the node is safe to be deleted or not, the other thread's hazardous pointers are scanned for a match with the current node to be deleted. If there is match, then another thread may still access the node, and it can not be deleted. In our variant of HP, we scan the other thread's hazard pointers for a match of the node we want to delete and we scan also for the `next` and `prev` of the node we want to delete. If another thread is using the previous node and attempting a `dequeue()` operation, it may dereference the next node (line 10 of Algorithm 2), which is the node we're trying to delete, and therefore, we can not delete it yet. If another thread is using the next node and attempting an `enqueue()` operation, it may dereference the previous node (lines 7, 8, and 9 of Algorithm 1), which is the node we're trying to delete, and therefore, we can not delete it yet.

Each thread is looking for usages of the pointers in the `next` and `prev` of the nodes in its own retired list, and does not dereference the `next` or `prev` of the nodes in the list/array of hazardous pointers. Such behavior would be incorrect and lead to a crash.

This simple trick of scanning for occurrences of `node.prev` to protect `lnext` in `dequeue()`, and scanning for `node.next` to protect the `lprev` pointer in `enqueue()`, reduces one seq-cst store in each method. Unfortunately there is no gain in applying this HP variant to the MS queue because its `enqueue()` already requires one seq-cst store (only the tail node needs to be protected), and for the `dequeue()` the next node can not be protected because there is no way of knowing what is the previous node.

4. THROUGHPUT

We executed two different benchmarks using a procedure similar to [6]. The single-enqueue-single-dequeue benchmark is shown in figure 1, with the right-side plot showing the ratio normalized to the MS queue.

The burst benchmark is shown in figure 2. We did

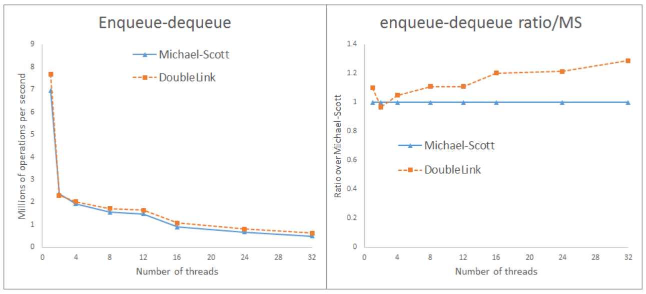

Figure 1: 10^8 pairs of enqueue-dequeue. Higher is better.

not add a random sleep between each operation, so as to maximize the effects of the high contention.

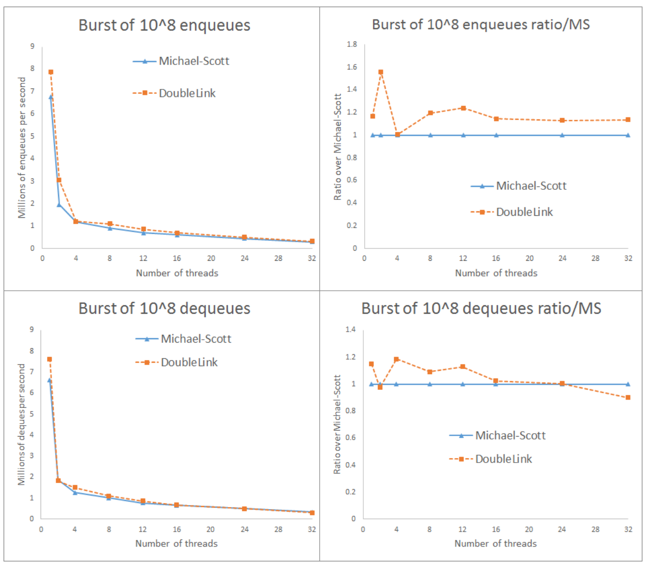

Figure 2: Two bursts of 10^8 enqueues / dequeues. Higher is better.

5. CONCLUSION

Multiple lock-free and wait-free singly-linked list based queues are known in the literature, however, many of them are based on the algorithm devised by Maged Michael and Michael Scott for their MS queue. We have presented here the DoubleLink queue, an alternative to the MS queue algorithm, based on a doubly-linked list. On managed languages where the memory reclamation is done by a GC, the advantage of the DoubleLink algorithm is only on the enqueue side, due to its reduction from two CAS to one CAS in the uncontended case. However, on non-managed languages, like C and C++, when using Hazard Pointers for memory reclamation, the DoubleLink queue requires one less sequentially consistent store for the dequeue, which means that both the `enqueue()` and `dequeue()` have less overhead than on the MS queue. When memory usage is not a stringent constraint, the DoubleLink queue can be used as a replacement to the MS algorithm, so as to provide equal or better throughput, going up to 1.3x.

6. REFERENCES

[1] KOGAN, A., AND PETRANK, E. Wait-free queues with multiple enqueuers and dequeuers. In *ACM SIGPLAN Notices* (2011), vol. 46, ACM, pp. 223–234.

[2] MICHAEL, M. M. Hazard pointers: Safe memory reclamation for lock-free objects. *Parallel and Distributed Systems, IEEE Transactions on 15*, 6 (2004), 491–504.

[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] MICHAEL, M. M., WONG, M., MCKENNEY, P., AND O'DWYER, A. Hazard Pointers - Safe Resource Reclamation for Optimistic Concurrency. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0233r2.pdf, 2016.

[5] MORRISON, A., AND AFEK, Y. Fast concurrent queues for x86 processors. In *ACM SIGPLAN Notices* (2013), vol. 48, ACM, pp. 103–112.

[6] RAMALHETE, P., AND CORREIA, A. A Wait-Free Queue with Wait-Free Memory Reclamation. https://github.com/pramalhe/ConcurrencyFreaks/papers/crturnqueue-2016.pdf, 2016.

[7] TREIBER, R. K. *Systems programming: Coping with parallelism*. International Business Machines Incorporated, Thomas J. Watson Research Center, 1986.