A Wait-Free Queue with Wait-Free Memory Reclamation

Pedro Ramalhete
Cisco Systems
pramalhe@gmail.com

Andreia Correia
Concurrency Freaks
andreiacraveiroramalhete@gmail.com

Abstract

Queues are a widely deployed data structure. They are used extensively in many multi-threaded applications, or as a communication mechanism between threads or processes. Real-time multi-threaded applications, like the ones running on networking devices, will typically need low-latency concurrent queues. Lock-based solutions have high tail latency, and even lock-free queues can cause a fat tail in the latency distribution when the number of threads increases, making them unsuitable for low-latency scenarios, with only wait-free (bounded) queues being capable of providing low enough latency at the tail of the distribution due to their strong fairness. Several wait-free queues are known, with three of them being multi-producer-multi-consumer, but they are complex, to the point that software engineers and concurrency experts have difficulties implementing them correctly.

We propose a new linearizable multi-producer-multi-consumer queue we named Turn queue, with a wait-free progress bounded by the number of threads, and an integrated wait-free memory reclamation technique. Its main characteristics are: a simple algorithm that does no memory allocation apart from creating the node that is placed in the queue, a new and fast wait-free consensus algorithm, the capability to achieve wait-free progress without requiring no other atomic instruction besides compare-and-swap (CAS), comes with an embedded wait-free bounded memory reclamation technique, has low memory usage when compared with other wait-free queues, is easy to plugin with other algorithms on either the enqueuing or dequeueing side.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; E.1 [*Data Structures*]: Lists, stacks and queues

Keywords wait-free, non-blocking queue, low latency

1. Introduction

A queue is one of the simplest data structures. It contains only two methods, an `enqueue()` and `dequeue()`, which for a single-threaded implementation or a lock-based implementation can take less than 10 lines of code. One would expect that a concurrent non-blocking queue would be just as simple, but unfortunately that may not be so. There are algorithms for simple lock-free queues with the most well known being the one by Maged Michael and Michael Scott (MS queue) [20] which can be implemented in 30 lines of code, but all other non-blocking algorithms for a memory-unbounded Multi-Producer-Multi-Consumer (MPMC) queue are more complex.

There are many queues in the literature with good throughput and blocking or lock-free progress, like for example the lock-free queue by Morrison and Afek [21], or the blocking queue by Fatourou and Kallimanis [7] based on the combined technique. When it comes to fairness and low latency, the most desirable of the progress conditions is wait-freedom. Of the known queues with wait-free properties, there are a few honorable mentions when it comes to simplicity: a simple Single-Produced-Single-Consumer (SPSC) wait-free queue was devised by Herlihy and Wing [12] but it is memory bounded; a simple MPSC queue using `atomic_exchange()` by Dmitry Vyukov with wait-free enqueues and blocking dequeues, where a lagging enqueuer can block all dequeuers indefinitely [26]. However, as we will see in section 1.2, we are interested in MPMC and low latency and therefore, we will compare solely with MPMC wait-free queues.

Wait-free queue algorithms are rare, but they are even less common when the enqueue and dequeue operations can be executed by multiple threads (MPMC). To the best of our knowledge, there are three memory-unbounded MPMC queues with wait-free progress.

The first of the three is the wait-free queue of Alex Kogan and Erez Petrank, which is based in the consensus algorithm of Lamport's Bakery [18]. At its core, it uses the enqueing and dequeing mechanism of the MS queue [20]. In this queue, all threads performing an enqueue or dequeue may help other operations to complete. The Kogan-Petrank (KP) queue does not take into consideration memory reclamation, and a substantial amount of memory is allocated for the immutable objects that are stored in the state array. The KP queue is wait-free bounded and uses an `OpDesc` object as an enqueue request and dequeue request. Each time a change of state is required, to guarantee atomicity, a new `OpDesc` object must be created, which causes churn on the memory allocation system and automatic Garbage Collector (GC).

The KP queue was designed to be used on a system with an automatic Garbage Collector (GC) and its publicly available implementation is in Java. In the original design there was no *self-linking* of the dequeued nodes which is essential to help the GC cleanup the unused nodes. If the GC has to traverse a very long list of nodes to determine if they are still in the list or not, it can cause large GC pauses which will adversely impact throughput and latency.

This queue achieves its wait-free progress using only loads, stores and Compare-And-Swap (CAS) operations on atomic variables. Because it is designed for Java, it can run in any CPU architecture for which a Java Virtual Machine (JVM) implementation exists. This algorithm is simple and elegant when compared to the other two MPMC wait-free queues.

Another wait-free queue was designed by Panagiota Fatouru and Nikolaos Kallimanis (FK queue) [6], using a wait-free consensus mechanism developed by the same authors, which they named P-Sim. The publicly available implementation is in C99 and was designed for Total-Store-Order (TSO) CPU architectures, having been tested on x86 and Sparc, and it is unlikely that it would work on ARM or PowerPC without some additional fence/barrier instructions. This queue makes usage of both CAS and fetch-and-add (FAA) instructions to achieve wait-free progress. This queue achieves better throughput than Michael-Scott when in high thread

count, due to aggregating multiple enqueues with a single CAS.

More recently, Chaoran Yang and John Mellor-Crummey implemented a wait-free queue (YMC queue) based in the FAA instruction [27]. Unlike the KP and FK queues which are singly-list based, this queue is based on a list of arrays, where each node of the queue contains an array of 10 million entries.

The YMC queue uses a slow-path-fast-path technique [17] to improve its throughput when running uncontended, but can cause worse latency at the tail of the distribution. Such an approach comes with a tradeoff between throughput and latency: the more times a fast-path is attempted, the higher (worse) will the latency at the tail of the distribution be.

When following the fast path, any operation of enqueue will take an enqueue ticket and a dequeue will take a dequeue ticket, using an FAA to guarantee unique and monotonically increasing tickets. Both enqueue and dequeue compete to get a position in the node's array, and even if the queue is empty, the ticket taken by a dequeue can not be reused and that position on the array will be taken and will never contain an item. This queue has an excellent throughput if the usage pattern is X enqueues followed by X dequeues, and if the amount of enqueues does not surpasses the size of the initial array. Also, adding a single item to the queue requires allocating an array in the node, and this array requires at all time the use of that memory occupying 10 million pointers, and as soon as the array is full, all threads currently attempting an enqueue will try to allocate the next node containing another array of 10 million entries, which will impact latency.

It uses an FAA to achieve consensus in the order of enqueing and dequeueing. On the slow-path, when method `enq_slow()` is called (listing 3 of [27]), the tail is read before the FAA is done (lines 75 and 78). If a thread gets suspended between the lines 75 and 78 it will have to traverse all the nodes until it finds the node with the entry with a matching id provided by the FAA. During this time, another thread(s) may enqueue a large amount of new nodes, advancing the `tail` each time. When the first thread wakes up in `enq_slow()` it will have to traverse an *unbounded* number of nodes until it finds the one it is looking for. Because of this, the method `enq_slow()` is *wait-free unbounded*.

The consensus algorithm of this queue uses an FAA instruction to achieve wait-free progress, thus implying that it will be only wait-free when deployed in CPU architectures that support a native FAA instruction. The authors also made an attempt to design a wait-free memory reclamation, we will show in section 4 to be incorrect. Even though the design of this memory reclamation technique can be easily corrected, it is still inherently *blocking* for the threads doing a dequeue, as we explain in more detail in section 3.

Table 1 displays a summary comparison between the different MPMC wait-free queues known in the literature and our own algorithm we named `Turn` Queue.

We present the following contributions:

- An algorithm to enqueue in a singly-linked list, that is wait-free bounded.
- An algorithm to dequeue from a singly-linked list, that does no memory allocation and is wait-free bounded.
- An embedded memory reclamation into the previous enqueue and dequeue algorithms, using hazard pointers such that they are wait-free bounded.
- A variant of Hazard Pointers we named Conditional Hazard Pointers, where the object is deleted after a certain condition is matched.
- An implementation of the KP queue that uses Hazard Pointers and Conditional Hazard Pointers, with wait-free memory reclamation.
- Three different implementation errors in the FK queue.
- One flaw in the design of the memory reclamation scheme of the YMC queue, and one implementation flaw.

In this paper we present a memory-unbounded multi-producer-multi-consumer wait-free queue named TurnQueue, that minimizes memory allocation, and with wait-free memory reclamation. As we have seen by the presentation on prior art, this is a hard task, that few have attempted and even fewer have succeeded.

1.1 Progress Conditions

There are multiple progress guarantees for non-blocking methods in data structures. A method acting on a concurrent object is:

- *obstruction-free* if a thread can perform an arbitrary operation on the object in a finite number of steps when it executes in isolation;
- *lock-free* if it guarantees that at least one thread running one of the concurrent methods of the object finishes in a finite number of steps;
- *wait-free* if it guarantees that every call finishes its execution in a finite number of steps;

Wait-free progress can be further classified as:

- *wait-free unbounded* if it is wait-free and there is no known bound on the number of steps;
- *wait-free bounded* if there is a known bound and it depends on the number of threads/processes.
- *wait-free populations oblivious* if there is a known bound and it does not depend on the number of threads/processes; This implies the bound is a constant;

A concurrent object is said to be lock-free if all of its methods are at least lock-free. A concurrent object is said to be wait-free if all of its methods are wait-free, and therefore, for a queue to be wait-free, it must have a wait-free `enqueue()` and wait-free `dequeue()`.

1.2 Design Goals

When designing the queue shown in the next section, we followed these guidelines, with descending order of importance: Low-Latency; Simplicity; Low resource usage.

Our primary goal was to achieve **low latency**, or to be more precise, low latency at the tail of the latency distribution (*tail latency*). In this context, we refer to *latency* as the amount of time it takes for a call to a method `enqueue()` or `dequeue()` to complete. Lock-based queues are *blocking*, and even when starvation free, it can happen that a thread grabs the lock and goes to sleep, blocking other threads from enqueueing or dequeueing, thus causing a *fat tail* in the latency distribution. *Lock-free* queues are susceptible to starvation, when one or multiple threads are the only ones doing any enqueueing or dequeueing and starving all other threads, which means that their tail latency can theoretically run to infinity, also causing a *fat tail* in the latency distribution. *Unbounded wait-free* queues are better, but can be hard to characterize, because they're unbounded they can have issues with memory reclamation, and have a tendency to have long tails in the latency distribution when running in scenarios with thread oversubscription (more threads than cores to run them on). A *wait-free bounded* queue is capable of providing strong latency guarantees even at the tail of the latency distribution.

	Progress of `enqueue()`	Progress of `dequeue()`	Consensus Protocol	Needs Atomic Instruction	Implementation for CPU architecture	Memory Reclamation	Min Memory Usage
Kogan-Petrank (KP)	wf bounded	wf bounded	Lamport's bakery	CAS	JVM	GC	$O(N_{threads})$
Fatourou-Kallimanis (FK)	wf bounded	wf bounded	FK algorithm	FAA + CAS	TSO (x86/Sparc)	none	$O(N^2_{threads})$
Yang-Mellor-Crummey (YMC)	wf unbounded	wf unbounded	FAA + Dijkstra	FAA + CAS	TSO (x86)	epoch	$O(N_{threads})$
Turn	wf bounded	wf bounded	Turn algorithm	CAS	C++14	wfb HP	$O(N_{threads})$

Table 1. Comparison between different characteristics of linearizable MPMC wait-free queues known in the literature. All queues have wait free progress (first two columns), with the YMC being *wait-free unbounded* while the other queues are *wait-free bounded* by the number of threads.

Three MPMC wait-free queues are known in the literature [6, 16, 27], but they are all considerably complex when compared to the MPMC lock-free queue by Michael and Scott [20]. This complexity of the algorithms makes the queues difficult to implement, causes them to be more susceptible to bugs, and this complexity is so steep that it shuns away library implementers and even concurrency experts, thus making them less likely to be deployed in production applications. Most large organization have code-reviewing practices as part of their quality control process. Even if there is a software engineer in such an organization willing to spend the considerable amount of effort required to understand and implement any of these queues, she then has to convince her colleagues during the code-review phase, of the correctness of the algorithm and the correctness of her implementation. We aimed for **simplicity** in our designs because a simple algorithm results in simple code, and a simple code has less bugs, is easier to understand, easier to teach, and therefore, easier to explain to your colleagues and pass through a code review.

Our third criteria was **low resource usage**. The wait-free queues known so far are all very heavy on the GC or memory management system. They require the creation of several objects to represent an enqueue request, or a dequeue request, or intermediate states. This constant creation and deletion of objects causes churn on the memory allocation system, possibly memory fragmentation, and can become very tricky to implement on non-managed languages like C/C++, to the point that KP queue is provided only with a Java implementation. Ideally, a queue based on a singly-linked list should not need to create any object other than the node that is placed in the queue by the `enqueue(item)` method, and this node should have as little extra members as possible, with the minimum being a pointer to the `item` that is being enqueued and a pointer to the `next` node in the list.

Obviously, we care about the throughput of the queue. If the operations are too slow then there will be no practical application for the queue. Throughput has the advantage that it is easily *measurable* with microbenchmarks, therefore, most papers focus on it, and we will report it as well on section 4.4. However, we put a higher emphasis on the three factors stated previously, and we will provide measurable comparisons based on these factors, where possible.

2. Queue algorithm

We define a queue to be correct if:

- One call to `enqueue(item)` inserts `item` in the end of the list.
- One call to `dequeue()` returns either the first item in the list, or `nullptr` if there are no items in the list.

Following the common convention, we refer to the `tail` as a pointer to the last node of the list, where new nodes are enqueued, and to the `head` as a pointer to the first node of the list, where nodes are dequeued. When `head` and `tail` refer to the same node, the queue is considered empty, there is no item in the queue.

We will start by presenting the necessary building blocks of the Turn queue. An essential part of any singly-list based queue is the nodes that compose it. These nodes store the items that the queue contains at any given time. The `Node` is presented in Algorithm 1, and it contains the pointer to the item, the pointer to the next node that follows in the list of items present in the queue, and there are two extra variables, `enqTid` that indicates by which thread the node and corresponding item was enqueued by, and `deqTid` that indicates which thread the node was dequeued by.

Algorithm 1 Node

```
1  struct Node {
2    T* item;
3    const int enqTid; // Used only by enqueue()
4    std::atomic<int> deqTid = { IDX_NONE }; // Used only by dequeue()
5    std::atomic<Node*> next = { nullptr };
6    Node(T* item, int tid) : item{item}, enqTid{tid} { }
7  };
```

Two arrays are also required, `enqueuers` and `dequeuers`, with size `MAX_THREADS`, where a unique entry is attributed to each thread that performs an enqueue or dequeue, respectively. These two arrays contain only references to nodes of the queue. In `enqueuers`, each entry has been assigned to a specific thread, the index of which is placed in the node's `enqTid`. `enqueuers` is an array of atomic pointers to nodes to be inserted in the queue, and when the pointer is `nullptr` it means there is no node to insert and therefore, no request. The `dequeuers` is in reality two arrays of nodes named `deqself` and `deqhelp`, which will be explained later.

The Turn queue starts by containing a sentinel node that both the `head` and `tail` of the queue point to. This sentinel node has an `enqTid` of 0. We have chosen to set the sentinel node's `enqTid` to 0 but it could have been any number between 0 and `MAX_THREADS-1`. This indicates to all threads calling the enqueue operation that the next node to be inserted in the queue is the one present in position 1 of the `enqueuers` array. In case that position in the `enqueuers` array has a null pointer, then all threads will continue to the next entry to the right, until they find a entry with a non-null node to be inserted, or at most, `MAX_THREADS` entries to the *right*. The enqueue method is guaranteed to return at most after `MAX_THREADS` iterations because in case it does not find its own node to process, it means some other thread has enqueued it.

2.1 Wait-free bounded enqueue

In this section we describe an enqueue algorithm with a number of steps bounded by the number of threads. At the basis of the Turn queue is a singly-linked list, where enqueing consists of one CAS on the last node of the list and another CAS to advance the `tail` pointer, just like on the MS queue [20]. This algorithm requires each node to have an `enqTid` where a unique identifier of the thread is stored. This variable does not need to be atomic, and has a purpose similar to the one in KP [16], although it uses a different consensus mechanism. Our consensus algorithm is based on the consensus mechanism of the CRTurn starvation-free mutual exclusion lock by Correia and Ramalhete [5], inspired by Lamport's One Bit Solution, where each thread publishes its intent, in this case, an intent to enqueue a node, and the decision of who is the next thread

is based on who is the next request *to the right* of the current `turn`. The current turn is the `enqTid` of the node at the tail of the queue.

In our implementation, the intent to enqueue is signaled with a non-null store of the pointer to the node to be enqueued in an array named `enqueuers`. Each entry of `enqueuers` is uniquely assigned to a thread, whose index is used as the `enqTid`. The first non-null entry in the `enqueuers` array that is *to the right* of the current `turn` is the one that all threads will attempt to help next to enqueue in the singly-linked list, using a CAS, similarly to the MS lock-free enqueue algorithm. Each thread calling `enqueue()` will continue helping the next enqueue request to complete until its own request has been completed.

The mechanism that insures a wait-free bounded enqueue relies on the `enqTid` variable of the node referenced by `tail`. This indicates to all threads that are executing an enqueue, that the last request that was satisfied belonged to the thread that published its request in position `enqTid` of the `enqueuers` array. All threads realize that the thread's request on that position was the last to be executed so it will be the next request *to the right* to have its turn and get its request executed. When it reaches the end of the array with all requests processed to the right, it will start processing requests at the start of the array, functioning like a circular array, modulos `MAX_THREADS`. It can happen that not all threads actively calling `enqueue()` will be attempting to enqueue the same node because some threads may have seen an entry immediately to the right of the current turn as being non-null, while other threads saw it as null (no request) and are instead attempting to insert a request further to the right. Regardless of which node gets enqueued, the new `turn` will be based on the node that was successfully inserted at the tail of the list, and from then on, there is a guarantee that all requests will be seen by the current threads or new threads calling `enqueue()`, thus implying that *at most* there will be `MAX_THREADS`-1 other nodes being inserted in front of any given request to enqueue, which means that the enqueue will complete in a number of steps that is bounded by the number of threads.

The C++ source code for `enqueue()` is shown in Algorithm 2.

Algorithm 2 Turn enqueue()

```
1  if (item == nullptr) throw std::invalid_argument("item_can_not_be_nullptr");
2  const int myidx = getIndex();
3  Node* myNode = new Node(item,myidx);
4  enqueuers[myidx].store(myNode);
5  for (int i = 0; i < maxThreads; i++) {
6    if (enqueuers[myidx].load() == nullptr) {
7      hp.clear();
8      return;
9    }
10   Node* ltail = hp.protectPtr(kHpTail, tail.load());
11   if (ltail != tail.load()) continue;
12   if (enqueuers[ltail->enqTid].load() == ltail) {
13     Node* tmp = ltail;
14     enqueuers[ltail->enqTid].compare_exchange_strong(tmp, nullptr);
15   }
16   for (int j = 1; j < maxThreads+1; j++) {
17     Node* nodeToHelp = enqueuers[(j + ltail->enqTid) % maxThreads].load();
18     if (nodeToHelp == nullptr) continue;
19     Node* nullnode = nullptr;
20     ltail->next.compare_exchange_strong(nullnode, nodeToHelp);
21     break;
22   }
23   Node* lnext = ltail->next.load();
24   if (lnext != nullptr) tail.compare_exchange_strong(ltail, lnext);
25 }
26 enqueuers[myidx].store(nullptr, std::memory_order_release);
27 hp.clear();
```

2.2 `enqueue()` correctness

To help prove that there is one and only one new node being inserted in the linked list when an enqueue request is created, we define the following invariants for the `enqueue()` method:

Invariant 1. *A new node can only be inserted after the tail.*
This is done with a CAS from `nullptr` to `newNode` in line 20.

Invariant 2. *The tail can only advance after the insertion of a new node.*
The tail is advanced with `tail.CAS(ltail,lnext)` only if `lnext` is non-null, as shown in lines 23 and 24.

Invariant 3. *The tail is always pointing to either the last node, or to the before-last node.*
`tail` starts by pointing to the last node on the list. By Invariant 1, it is only possible to append a new node to `tail`, by doing CAS on tail.next (line 20), this will make `tail` point to the before-last node on the list. From this moment on, all CAS on `tail.next` will fail until `tail` advances to the last node on the list (line 24).

Invariant 4. *Every node inserted in the list will be at some point in time the tail of the list.*
By Invariant 3 and because all threads have to ensure that `tail` will point to the last node of the list before the insertion of a new node on the list.

Invariant 5. *After publishing a node in `enqueuers`, the tail will advance at most `MAX_THREADS`-1 before the node is inserted in the queue.*
Suppose that there are no other nodes in `enqueuers` between *turn* and Thread A's entry i, but before inserting the `newNode`, Thread B with entry $i+1$ succeeds in inserting its own node (because it hadn't seen Thread A's node). Afterward, all other threads come and fill all entries of `enqueuers`, thus having their nodes inserted in the list in front of Thread A's node (the Thread A itself will help the others do this if needed). This implies that after reading `tail`, at most, `MAX_THREADS`-1 nodes will be inserted before Thread A's node.

Invariant 6. *An entry in the `enqueuers` array is `nullptr` only if the node has been added to the queue.*
An entry in the `enqueuers` is set to `nullptr` through a CAS from `tail` to `nullptr` in line 14 and every node added to the queue was at some point in time `tail` (Invariant 4).
It is also possible for the owner thread to set its own entry to `nullptr` in line 26, this is only done after `MAX_THREADS` iterations, and, by Invariant 5, after `MAX_THREADS` iterations it is guaranteed that the node was added to the queue.

Invariant 7. *A node will not be inserted in the queue more than once.*
By Invariant 4, every node will be at some point in time the tail of the list. Also, before trying to insert a new node, every thread insures that the node pointed by `tail` is no longer in the `enqueuers` array (line 14). Making sure that the node is not in the `enqueuers` array before searching for the next node to insert, guarantees that it will not be inserted again.

From invariant 5 we see that the method `enqueue()` is *wait-free bounded* by the maximum number of threads.

By itself, this algorithm suffices to implement a wait-free MPSC queue. To implement a wait-free MPMC queue we need a wait-free `dequeue()` like the one shown in section 2.3.

2.3 Wait-free bounded dequeue

Similarly to the `enqueue()`, we devised a dequeuing algorithm that is inspired by the lock-free dequeueing algorithm created by Maged Michael and Michael Scott, by decomposing the work in a wait-free sequence of steps. At the basis of the wait-free consensus is the same CRTurn consensus algorithm that was used in `enqueue()`. The C++ source code for `dequeue` is shown in Algorithm 3, and its helper functions in Algorithm 4.

Algorithm 3 Turn dequeue()

```
1  T* dequeue() {
2    const int myidx = getIndex();
3    Node* prReq = deqself[myidx].load();
4    Node* myReq = deqhelp[myidx].load();
5    deqself[myidx].store(myReq);
6    for (int i=0; i < maxThreads; i++) {
7      if (deqhelp[myidx].load() != myReq) break;
8      Node* lhead = hp.protectPtr(kHpHead, head.load());
9      if (lhead != head.load()) continue;
10     if (lhead == tail.load()) {
11       deqself[myidx].store(prReq);
12       giveUp(myReq, myidx);
13       if (deqhelp[myidx].load() != myReq) {
14         deqself[myidx].store(myReq, std::memory_order_relaxed);
15         break;
16       }
17       hp.clear();
18       return nullptr;
19     }
20     Node* lnext = hp.protectPtr(kHpNext, lhead->next.load());
21     if (lhead != head.load()) continue;
22     if (searchNext(lhead, lnext) != IDX_NONE) casDeqAndHead(lhead, lnext, myidx);
23   }
24   Node* myNode = deqhelp[myidx].load();
25   Node* lhead = hp.protectPtr(kHpHead, head.load());
26   if (lhead == head.load() && myNode == lhead->next.load()) {
27     head.compare_exchange_strong(lhead, myNode);
28   }
29   hp.clear();
30   hp.retire(prReq);
31   return myNode->item;
32 }
```

Algorithm 4 Helper functions for Turn's dequeue()

```
34 int searchNext(Node* lhead, Node* lnext) {
35   const int turn = lhead->deqTid.load();
36   for (int idx=turn+1; idx < turn+maxThreads+1; idx++) {
37     const int idDeq = idx%maxThreads;
38     if (deqself[idDeq].load() != deqhelp[idDeq].load()) continue;
39     if (lnext->deqTid.load() == IDX_NONE) {
40       lnext->casDeqTid(IDX_NONE, idDeq);
41     }
42     break;
43   }
44   return lnext->deqTid.load();
45 }
46
47 void casDeqAndHead(Node* lhead, Node* lnext, const int myidx) {
48   const int ldeqTid = lnext->deqTid.load();
49   if (ldeqTid == myidx) {
50     deqhelp[ldeqTid].store(lnext, std::memory_order_release);
51   } else {
52     Node* ldeqhelp = hp.protectPtr(kHpDeq, deqhelp[ldeqTid].load());
53     if (ldeqhelp != lnext && lhead == head.load()) {
54       deqhelp[ldeqTid].compare_exchange_strong(ldeqhelp, lnext);
55     }
56   }
57   head.compare_exchange_strong(lhead, lnext);
58 }
59
60 void giveUp(Node* myReq, const int myidx) {
61   Node* lhead = head.load();
62   if (deqhelp[myidx].load() != myReq) return;
63   if (lhead == tail.load()) return;
64   hp.protectPtr(kHpHead, lhead);
65   if (lhead != head.load()) return;
66   Node* lnext = hp.protectPtr(kHpNext, lhead->next.load());
67   if (lhead != head.load()) return;
68   if (searchNext(lhead, lnext) == IDX_NONE) {
69     lnext->casDeqTid(IDX_NONE, myidx);
70   }
71   casDeqAndHead(lhead, lnext, myidx);
72 }
```

The dequeue operation can have two possible outcomes, it can return an item from the queue or a null pointer in case the queue is empty. In case the queue is not empty, the algorithm follows the same approach used in the enqueue, it uses the Turn consensus to determine which request is to be satisfied. When a thread i calls `dequeue()` it will *open* a request for a dequeue by setting `deqself[i]` to point to the same node as `deqhelp[i]`. Following the Turn consensus protocol, when a thread decides that it's thread's i turn, it will assign the next node's `deqTid` to i using a CAS, and if successful, from that moment on the node belongs to thread i. All threads that see `node.deqTid` different from `IDX_NONE` will be able to help by doing CAS in `deqhelp[i]`, this will guarantee that the result is visible to thread i and at the same time, it will make `deqhelp[i]` different from `deqself[i]` which closes the request. It is now safe to advance `head`.

By using nodes from the queue as a way to represent a logical dequeue request, we avoid the overhead of allocating a dequeue request object, and of subsequently tracking the request object's lifetime, which would require more hazard pointers. Having more hazard pointers would impact performance and add complexity to the algorithm.

It is possible to design an algorithm with the same functionality but using a single array of dequeue requests. For example, we can have a single array of pointers to nodes named `dequeuers`, and an extra atomic boolean member in each node, named `isRequest`, which is set to `false` on initialization. To open a dequeue request this member is set to `true` at the start of the dequeue method. When a thread wants to close thread's i request it will make `dequeuers[i]` point to the node assigned to its request using a CAS. This will at the same time, guarantee that the result is visible to thread i and close the request, because the node's member `isRequest` is by default set to false. However, with such an approach the `searchNext()` will have to protect each pointer with an hazard pointer for every entry in `dequeuers` that it scans, to be able to dereference `node->isRequest`, thus increasing the synchronization cost in the consensus mechanism. As shown in line 38 of `searchNext()` in Algorithm 4, our algorithm with two arrays avoids the need to dereference the nodes that are scanned, thus reducing the usage of hazard pointers.

2.3.1 Giveup procedure

When thread i sees an empty queue due to `head` pointing to the same node as `tail`, it will have to rollback the dequeue request. This procedure is executed by `giveUp()` in lines 60 to 71.

To rollback the dequeue request, thread i will set `deqself[i]` to its previous value, which was stored in a local variable `prReq`.

It can happen that during the rollback, another thread that saw the request of thread i opened, assigned it a node, and that node

was present on the list between the read of `head` in line 8 and the read of `head` in line 61. In that case, the result is published in `deqhelp[i]`, and thread i can return the item, thus aborting the giveup. Unfortunately, it is still possible that some thread saw the request opened but has not yet published in `deqhelp[i]`. In this case it is necessary to guarantee that the first node of the queue is assigned to some thread, and if there isn't any open request, the thread i will assign the node to it self in line 69. Due to happens-before guarantees, once the head advances, all threads seeing the new head will subsequently see the rollback. From this moment on, if it wasn't already assigned a node to thread i, then it is guaranteed that there will not be a node assigned to thread i for this request.

Through the giveup procedure, thread i has to ensure that the request was never satisfied by any thread, in order to return `nullptr`, and it will always return any item that was assigned to the current request. The `giveUp()` method can be seen as an attempt to give up, not a definite give up. This is a complex code path but fortunately, it will be rarely executed, only when the `dequeue()` doesn't see a node in the queue and then comes an `enqueue()` and adds a new node and the `dequeue()` now sees it between the two checks of line 10 and line 63.

By itself, the dequeue method suffices to implement a wait-free SPMC queue. To implement a wait-free MPMC queue we need a wait-free `enqueue()` like the one shown in section 2.1.

2.3.2 `dequeue()` correctness

To prove that `dequeue()` returns either the item from a unique node, or `nullptr` if the queue is empty, we define the following invariants for the `dequeue()` method:

Invariant 8. *A node that becomes unreachable because the head advanced, will always be accessible to the assigned thread.*
If a node is assigned to thread i (node.deqTid is i), then whatever thread j advances the `head`, must first confirm that `deqhelp[i]` has the pointer to the node and if not, place it there with a CAS before advancing the `head`, as shown in `casDeqHead()` (line 54). Thread i will be able to access the node and its `item` by reading the contents of `deqhelp[i]`, as shown in lines 24 and 31 of `dequeue()`.

Invariant 9. *Each node can be assigned to a unique request for dequeue.*
A node is assigned to a thread i using a CAS operation on node.deqTid from `IDX_NONE` to the thread's id i, line 40 or 68 of `dequeue()`. Once a node has been assigned to a thread, this assignment will not change for the duration of the lifetime of the node. After a node has been assigned to a thread asking a dequeue request, it can be published to the corresponding request only if the head and the request have not changed, as seen in lines of 53 and 54 of `dequeue()`) and it is published with a CAS operation. Also, the hazard pointer in line 52 guarantees that the precedent node in `deqhelp[i]` is not deleted and reused, preventing an ABA issue.

Invariant 10. *Each node can be retired and subsequently deleted by only one thread.*
The thread that has been assigned the node is the one responsible for retiring it and from Invariant 9 we know that each node is assigned to only one thread request.

Invariant 11. *A thread i returning `nullptr` will not be assigned a node.*
As explained in the giveup section, the `giveUp()` method will guarantee that it will only return a null pointer if and only if all other helping threads have not assigned a node to `deqhelp[i]`, and that the head has advanced after the request has been rolled-backed, guaranteeing that all other threads will see a closed request for thread i once they realize the head has advanced.

The consistency model we use for our queue is *linearizability* [13], which means that each operation on the queue must appear to occur instantaneously at a point in time within the interval during which the operation executes. We chose linearizability as the goal because, although queues with weaker consistency models exist and they may theoretically provide better throughput [24], they are harder to reason about, which goes against our goal of simplicity, and it is more difficult to prove their correctness. Items added by a thread calling `enqueue()` become visible to threads calling `dequeue()` (line 10 of Algorithm 3) when the `tail` advances (line 24 of Algorithm 2). An item that is removed from the queue is no longer visible once the head advances (line 57).

2.4 Hazard Pointers and ABA

One of the requirements for the correct usage of Hazard Pointers is that a node can only be retired after it is no longer accessible by other threads through shared variables. In a singly-list based queue, this typically happens in `dequeue()` when the head advances to the next node, thus making the previous node unreachable. In our algorithm, the node may still be accessible through `deqhelp` or `deqself`, and it is only after it can not be accessible trough those arrays that it can be retired by calling `hp.retire(node)`. When a node is dequeued, it is placed in `deqhelp`. On the next dequeue operation, it is placed in `deqself` to indicate an open request for dequeue, and if the dequeue is successful, it will be taken out of `deqhelp` (replaced by a new node). On the following successful dequeue, it will be permanently removed from `deqself` and it is then that it can be safely retired.

A fundamental step for the correct usage of Hazard Pointers, is to read the content of a shared variable (a pointer) and *publish* that pointer in an array or list of hazard pointers, and then *validate* the content of the shared variable has not changed. In all our hazard pointers in the dequeue procedure, we check that the `head` has not advanced after publishing the pointer. If the `head` has advanced, the pointer needs to be re-read and re-published, but it also means that at least one thread has had its dequeue request completed and it is now another thread's turn. We therefore jump to another iteration of the loop. This validation step is done immediately after publishing an hazard pointer, in lines: 9, 21, 26, 53, 65 and 67.

In line 52 of the `casDeqAndHead()` method, we publish a hazard pointer to protect the node pointed by `deqhelp[ldeqTid]` even though we don't dereference it. The reason for doing this is related to ABA. If there was no hazard pointer protecting the node in `deqhelp[ldeqTid]` and a thread would sleep before executing the CAS in line 54, it could happen that this request could be closed by another thread, then the node retired, and later the node reused and end up again in `deqhelp[ldeqTid]`. The sleeping thread then wakes up and executes the CAS in line 54 which incorrectly succeeds. To prevent this kind of ABA issue with a succession of events *retired-deleted-created-enqueued-dequeued*, we need one hazard pointer protecting `deqhelp[ldeqTid]`.

A non-obvious detail is the reason why the comparison of `deqself[idDeq]` and `deqhelp[idDeq]` in `searchNext()` is not affected by ABA issues, and therefore, we don't need to protect these two variables with hazard pointers. If the thread id i is different from `idDeq`, it means that at any point in the execution, the thread whose id is $idDeq$ may retire and subsequently delete the nodes in `deqself[idDeq]` and `deqhelp[idDeq]`. It may occur that in line 38 between the load of `deqself[idDeq]` and the load of `deqhelp[idDeq]`, the node that was on `deqself[idDeq]` gets *retired-deleted-created-enqueued-dequeued* and placed in `deqhelp[idDeq]` as a closing request, however, this will not cause

thread i to satisfy a request that was already closed. When thread i executes the load of `deqhelp[idDeq]`, it will see the same node that was in `deqself[idDeq]` and erroneously conclude that there is an open request for thread $idDeq$, and this is an example of an ABA occurrence. This ABA event is harmless because, in order for the node in `deqhelp[idDeq]` to be retired it means there must have been at least two successful dequeue operations in the queue, i.e. the head advanced at least twice, therefore implying that the node after the head that was read before calling `searchNext()` must already be assigned, and therefore, `searchNext()` will return the currently assigned thread id. In summary, it can happen that a closed request is seen as opened, but in such case, the condition in line 39 will always be false. On the other hand, an opened request will never be seen as closed, because it would imply that the node on `deqself[idDeq]` has been retired-deleted-created-enqueued-dequeued, which is impossible because a node in `deqself[idDeq]` can only be retired after the request is closed.

3. Memory reclamation

Progress has been made in the field of automatic Garbage Collectors (GC), but when it comes to latency, most GC still require stop-the-world pauses, or safepoints, or the algorithms have to be modified to use a GC, such as doing self-linking of unused nodes, which makes them ill suited for low latency applications. Moreover, using GC simplifies the design, but unfortunately, native languages such as C and C++ do not typically have a GC (with D being the exception), and even on languages that do have a GC, the implementation of the GC is blocking. There is currently no wait-free GC algorithm in the known literature [22].

An integral part of designing an efficient queue is to safely and efficiently reclaim the memory of the unused nodes. This job typically falls on the `dequeue()` method because any given node can only be reclaimed after it has been dequeued. When designing a wait-free queue this becomes even more difficult due to the need of providing such reclamation in an equally wait-free fashion. Manual concurrent memory reclamation techniques are composed of two distinct operations, the *protect* operation which protects one or a group of objects, and the *reclaim* operation which attempts the reclamation of the memory for one or multiple objects. Many such memory reclamation techniques are known in the literature [1, 2, 14, 19] but very few can provide efficient wait-free bounded progress for both *protect* and *reclaim*. For the particular case of a queue, the `enqueue()` method calls the *protect* operation and the `dequeue()` calls both *protect* and *reclaim*, which implies that to have both methods wait-free bounded we must employ a memory reclamation that is at least wait-free bounded, for both *protect* and *reclaim*. Table 2 shows a comparison of the progress conditions for different memory reclamation techniques, where the second line is a variant of Hazard Pointers we designed to be used in the KP queue.

One important detail about table 2 is that the epoch-based reclamation technique described by Harris [10], on which the technique used for the YMC queue is based off, is blocking when doing memory reclamation. If there is a thread that lags behind while holding a pointer to an older node/epoch/ticket, no further memory reclamation will be done. The progress condition of this technique is *blocking* because it is not resilient to faults although some articles argue otherwise [27].

The definition of lock-free progress condition assumes that at least one thread is making *progress*. In our understanding, if a function whose purpose is to reclaim memory has its work postponed indefinitely, then there is no progress being made, which contradicts the definition of lock-free. What is the purpose of having a method that returns in a finite number of steps if the work that it is supposed to perform is never attained? This is a waste of CPU resources and energy. Both lock-free and wait-free progress conditions imply progress even in the presence of faults [11], and therefore, these epoch-based reclamation techniques are neither. This issue is particularly noticeable with oversubscription, where it is common for a reader to be put to sleep while other threads are scheduled. During such a sleep period, no memory reclamation will occur, and the list of retired objects will grow quickly. With wait-free (bounded) memory reclamation this will not be the case and the number of unreclaimed objects will be limited to the maximum bound. We will now describe the two different approaches to reclaim memory that we have used.

	protect operation	reclaim operation
Hazard Pointers	lock-free/wf bounded	wf bounded
Conditional Hazard Pointers	lock-free/wf bounded	wf bounded
RCU-Epoch	wfpo	blocking
Epoch-based	wfpo	blocking*
StackTrack	lock-free	lock-free
Drop the anchor	lock-free	lock-free
Pass the buck	lock-free	lock-free

Table 2. Progress conditions of different memory reclamation techniques. In the case of a queue, the `enqueue()` method calls the protect operation, and the `dequeue()` method calls the protect and reclaim operations. Hazard pointers are typically implemented in a lock-free way for the readers, but depending on the algorithm, they may be used in a way that allows them to be wait-free. Methods that are wait-free bounded are represented as *wf bounded*, and wait-free population oblivious are represented as *wfpo*. * Although some literature refers to epoch-based reclamation as being *wait-free unbounded*, the proper designation is *blocking*, as explained in section 3.

3.1 Wait-Free Hazard Pointers

Our Turn queue algorithm is wait-free bounded, for both `enqueue()` and `dequeue()`, and therefore, to avoid losing progress guarantees, we need to use a memory reclamation method with at least the same guarantees of progression. Our choice was to use Hazard Pointers (HP). In HP, the protect operation can be implemented in a wait-free way if instead of creating a loop with load-store-load of the pointer that is being read, we do a single load-store-load sequence and if the second load is different from the first one, then we jump to the next iteration in the algorithm's loop, therefore advancing at least one step in the wait-free algorithm. Pseudo-code for both cases can be seen in Algorithm 5. Notice that this technique is not applicable to the traversal of a singly-linked list, but it works fine on the mutative operations of a queue.

Our HP implementation has three API methods: `protectPtr(int index, Node* node)` that will do a seq-cst store of `node` on the current thread's `index` hazard pointer; `clear()` will clear all HPs of the current thread by setting them to `nullptr`; `retire(Node* node)` will scan the hazard pointers of all threads and if at least one of them is still using a pointer to `node`, it will store the pointer in a thread-local list, which it will scan at the next call to `retire()` to check if the node can then be safely deleted. In our implementation of hazard pointers, used by all three queues (MS, KP, Turn), we chose an R parameter of zero (see figure 2 in [19]), with the purpose of reducing latency on `dequeue()` as much as possible.

In all algorithms in which the Hazard Pointers technique is deployed, for every shared pointer that is dereferenced, one hazard pointer is needed to protect that pointer/object. If the pointer is not dereferenced, then there may still occur ABA issues, in which case an hazard pointer is needed.

Algorithm 5 Lock-Free or wait-free bounded Hazard Pointers

```
73 void lockFreeMethod() {
74   Node* lhead = head.load();
75   do {
76     hp[mytid].store(lhead);
77   } while (lhead != head.load());
78   doSomethingWithHead(lhead);
79 } // May need infinite steps to complete
80
81 void waitFreeBoundedMethod() {
82   for (int istep=0; istep < maxSteps; istep++) {
83     Node* lhead = head.load();
84     hp[mytid].store(lhead);
85     if (lhead != head.load()) continue;
86     doSomethingWithHead(lhead);
87     break;
88   }
89 } // Completes in maxSteps
```

3.2 Conditional Hazard Pointers

The KP queue was originally designed for a system with a GC, namely Java. To implement this queue without a GC we had to do several modifications pertaining to memory reclamation, such as: tracking the lifetime of `OpDesc` instances with HP; determine which and how many hazard pointers are needed, and when can they be cleared; transform every `while()` loop that is not guaranteed to complete by the algorithm into a `for()` loop, including the HP related loops; apply the Conditional Hazard Pointers technique to track the node's lifetime.

One of these modifications involved designing a variation on HP we called Conditional Hazard Pointers (CHP). We needed such an approach because in the KP queue there is no easy way to know if the node will still be dereferenced or not, even if the `head` has advanced. Suppose `head` is currently pointing to `nodex` and Thread 1 calls a dequeue operation that assigns the `nodey` to Thread 2, then advances `head` to the next node `nodey`, after placing it in the `OpDesc` instance of Thread 2. Then comes Thread 3 calls a dequeue operation and advances the `head` from `nodey` to `nodez` and retires `nodey`. How will Thread 3 know whether or not it is safe for `nodey` to be deleted, seen as there is no hazard pointer to protect it, but there is still a way to access it, namely from the `state` array? The Turn algorithm avoids this problem by having the item to be returned on the same node that is being dequeued and therefore, retired.

In CHP, after a node is retired and placed in the list of retired nodes, there must first be a condition that is matched before the node is deleted. For the particular case of the KP queue, we modified the `item` in the node to be of type `std::atomic<>`. An `item` with a value of `nullptr` represents an object that is safe to be deleted. The thread that dequeues the node places it in the retired list, but it is up to the thread that is returns the node's item set the `Node.item` to `nullptr` so that it can be deleted. Systems with a GC do not have this issue because the node will not be deleted while there is a pointer to it, namely, the member `next` of the previous node is still referencing it.

4. Comparisons

We will now describe some of the issues we encountered in the FK and YMC queue that led us to not include these queues in our comparisons.

After running our stress tests on the FK queue we identified an issue that would occur whenever more than one thread would contend on the queue. With more than one thread acting on the queue, all threads, at the exception of the thread with id 0, are very likely to return 0 (null), regardless of what was the item enqueued. This was a simple bug to correct and the authors did so in less than a week. This fix has been pushed to the authors github repository [15] on commit 4724892.

Later, we found two other issues related to reading an inconsistent `EnqState` instance. To fix these issues, we must add a check that `tmp_sp` and `enq_sp` are still the same before the CASPTR() in `connectQueue()` (line 159 on [8]), and another similar check in `enqueue()` before returning if the operation has already been applied (line 196 on [8]) to guarantee that while reading the `applied` variable no other thread is writing to it at the same time, avoiding bit flicker [3].

A more serious issue is the fact that there is no embedded memory reclamation and it is not simple to add one. This means that successive enqueues will allocate new nodes that will never be deleted, until the application where the queue is running exhausts all available memory. Our benchmarks and stress tests run for long durations of time, tens or hundredths of seconds, to account for warmup and provide more stable results, which means that such a leak will eventually consume all the memory available on the system, making this queue unsuitable to be used in our benchmarks. This continuous allocation of new pages of memory causes high latency in the tail of the distribution for enqueues, making it uncomparable with the other queues. Furthermore, tracking and reclaiming memory for the nodes in the queue can consume a non negligible amount of resources, and by not reclaiming the memory allocated to the nodes, this implementation has an unfair performance advantage over any other queue that correctly performs memory reclamation. For these reasons, we decided not to include the FK queue in our microbenchmarks.

In YMC, threads that are traversing the queue, whether they're calling `enqueue()` or `dequeue()`, need to *publish* the ticket of the the first node they've seen, which is the `head` for `enqueue()`, or the `tail` for `dequeue()`. When a thread dequeues and then retires a node, it will then scan the tickets published by the other threads and determine what is the lowest ticket. If the ticket of the retired node is lower than the lowest of the published tickets, it is safe to delete the node because no thread is holding a reference to it nor is it able to reach it.

A detail that can easily escape, is that publishing the ticket of the `head` or `tail` nodes requires doing a dereference of the node, which can be catastrophic if another thread advances the head. That node is not yet protected and can be deleted before the ticket is read. This behavior is incorrect and will lead to a crash of the application. Unfortunately, the original design and the code on github for the YMC queue suffers this exact issue (as of commit 6a97077), and running it on our suite of stress tests under AddressSanitizer [23] has shown it as an *heap-use-after-free* bug. One scenario where this bug occurs, is when calling `dequeue()` which calls `deq_fast()` which calls `find_cell()` which dereferences the pointer to the head `sp` in line 36 of their paper [27], to read the `id` (what we call the ticket), and although there was a store of an hazard pointer of the head (line 213 of the paper), without re-validation the node may have been deleted. Doing the validation lopp for the hazard pointer shown in `lockFreeMethod()` of Algorithm 5 would fix this issue. Moreover, when we disable memory reclamation and let our stress tests run, for just a few seconds to prevent the server from running out of memory, we see failures in the stress tests due to *missing* items that were enqueued but never dequeued, which indicates there may be other issues in this implementation besides the design flaw in the memory reclamation. Due to it not having a wait-free memory reclamation, we decided not to include the YMC

queue in our microbenchmarks.

Regarding the KP queue, we have ported the code from Java to C++14 and modified several pieces of the algorithm to allow for the inclusion of HP and CHP. Although the KP queue is the simplest of the known wait-free queues, this was a difficult task, particularly to deploy the hazard pointers in a wait-free way. The source code to this C++14 implementation will be made available along with the Turn queue and the microbenchmarks.

4.1 Latency measurements

Table 3 shows the latency for the enqueue and dequeue operations measured for 30 threads, on an AMD Opteron 6272 server with a total of 32 cores, running Windows 10, using the following procedure: Each thread will pre-allocate two arrays with $2 \times 10^8/N_{threads}$ entries where the measurement of the delays of the individual calls to `enqueue()` and `dequeue()` are stored respectively. Each individual measurement is made with a call to `std::chrono::steady_clock::now()` before and after the call to `enqueue()` or `dequeue()` and the value is saved in nanoseconds units on the pre-allocated array. Each thread will enqueue $10^6/N_{threads}$ items in the queue, then wait for all the other threads to complete and then do $10^6/N_{threads}$ dequeues from the queue. This cycle of enqueuing/dequeuing executes two hundred times ($2 \times 10^8/10^6 = 200$), and guarantees that all the threads are either dequeueing or enqueueing, which provides more deterministic results. There are 10 extra initial bursts that are not accounted for in the distribution, that act as a *warmup*. At the end of the 200 bursts, the arrays of all threads are aggregated into a single array for the enqueues and a single array for the dequeues, and then sorted so that we can obtain the delay for a given quantile. This procedure is done for each of the queues MS, KP, and Turn, and executed for 7 $runs$. For a chosen quantile, we select the minimum and maximum of each run, and present it on table 3 in units of microseconds. The process that measured latency with this procedure was running on a 32 cores machine with *Realtime priority*, the maximum scheduling priority allowed on Windows 10.

	50%	90%	99%	99.9%	99.99%	99.999%
enqueue()						
MS	**51 - 54**	245 - 247	649 - 670	1303 - 1357	2166 - 2304	3193 - 3557
KP	222 - 230	272 - 282	317 - 330	354 - 370	**417 - 441**	**706 - 773**
Turn	142 - 149	**173 - 180**	**195 - 204**	**218 - 226**	558 - 580	1127 - 1155
dequeue()						
MS	**24 - 27**	**166 - 175**	629 - 690	3900 - 4642	8238 - 11870	13336 - 23637
KP	318 - 328	424 - 434	513 - 530	578 - 601	634 - 659	**750 - 792**
Turn	199 - 202	216 - 220	**232 - 236**	**248 - 258**	**481 - 495**	857 - 896

Table 3. Table showing the latency quantiles in microsecond units (min and max), with 30 threads, and 200 million measurements. Lower values are better, highlighted in bold for each group. As an example, we can interpret the value at the 99.999% for the MS queue meaning that in one of the 7 runs, from the 200 million calls to `enqueue()`, *there were* 2000 *calls that took more than 3557 microseconds to complete*.

As expected, the tail latency values for the MS queue are significantly higher (99.999% column) relatively to the KP and Turn queues. For `dequeue()`, there is a small advantage of the KP queue over the Turn queue on the extreme tail of the latency distribution, which we believe to be related to having less contention on the shared variables because KP spends a non-negligeable amount of time creating and deleting `OpDesc` objects and doing CAS on the `state` array, thus reducing contention on the `head` and `tail`, which acts as a kind of *backoff*. For the `enqueue()`, our Turn queue is better from the 99% to the extreme tail of the distribution 99.999%. Contrary to blocking queues, after *publishing* the intent to enqueue/dequeue, a wait-free method acting on a queue will make progress with a high probability even if it is not scheduled or is sleeping. This happens because the other threads will *help* it to enqueue/dequeue, and therefore, a valid (and perhaps interesting deliberate) strategy is to backoff and wait a while for another thread to help the enqueue/dequeue. Hence the intuitive advantage for having any kind of *backoff* in a wait-free queue, whether deliberate or not.

We repeated the same kind of procedure on the same 32 core server, for a range of different threads, and made a plot of each quantile as function of the number of threads, which we show in Figure 1. Each data point in the plot is the median of 7 runs. The maximum number of threads we used for measuring latency was 30 so that two cores were still available for the operating system to continue running correctly without impacting the microbenchmark.

Unlike prior studies of queue performance [20, 27] where each thread performs a random amount of "work" (from 50 to $100ns$) between operations to avoid artificially long run scenarios [20], we did not add a random delay. Such a random delay could have an impact on latency, but more importantly, we wanted to show that the tail latency on a lock-free queue is severely impacted as contention increases, while on wait-free queues it is not. Introducing such a delay would artificially reduce contention, although a *real-life application* would certainly do some kind of work between successive queue operations.

As seen in Figure 1, at the tail of the latency distribution the MS lock-free queue rises much faster than the KP and Turn wait-free queues. This effect is particularly evident for the `dequeue()`, likely because a dequeueing operation in the MS queue consists of a single CAS in the `head` variable, which creates a lot of contention in that variable, thus increasing the probability that one thread will starve, and therefore increasing tail latency.

We've added the Michael and Scott (MS) queue which is probably the simplest of the lock-free queues to serve as a baseline comparison on how simple can a non-blocking queue be. The MS queue has no thread-local variables, and the only shared variables are the `head` and the `tail`.

4.2 Memory usage

Networking devices and embedded devices typically have limited memory, which causes the software developers working on such platforms to use data structures that have a small memory footprint, hence the motivation to devise a wait-free queue with a low memory footprint. Table 4 shows a comparison of the memory usage for four different wait-free queues.

	KP	FK	YMC	Turn
sizeof(Node)	24	16	40	24
sizeof(EnqueueRequest)	80	32+	16	0
sizeof(DequeueRequest)	80	$32 \times N_{threads}$	16	0
sizeof(fixed per thread)	8	$80 \times N_{threads}$	72	24
Heap allocations per item	5+	1	3	1

Table 4. The values on the top four rows are for a 64 bit architecture, without padding or cache line alignment. Lower is better.

In KP, a node contains an `enqTid` and an `deqTid` besides the pointer to the item and the pointer to the next node, which results in 24 bytes. The KP queue has no enqueue or dequeue request, but there is an `OpDesc` object with immutable members whose state represent a similar concept. The size of an `OpDesc` object is at least 80 bytes. Multiple `OpDesc` objects can be created at each step of the enqueing and dequeuing, but at least two instances will be created for the `enqueue()` and another two for the `dequeue()`, to open and close each of the requests respectively. Each thread will be allocating one node and four `OpDesc` instances per item, plus more allocations in case of failed attempts to help other threads.

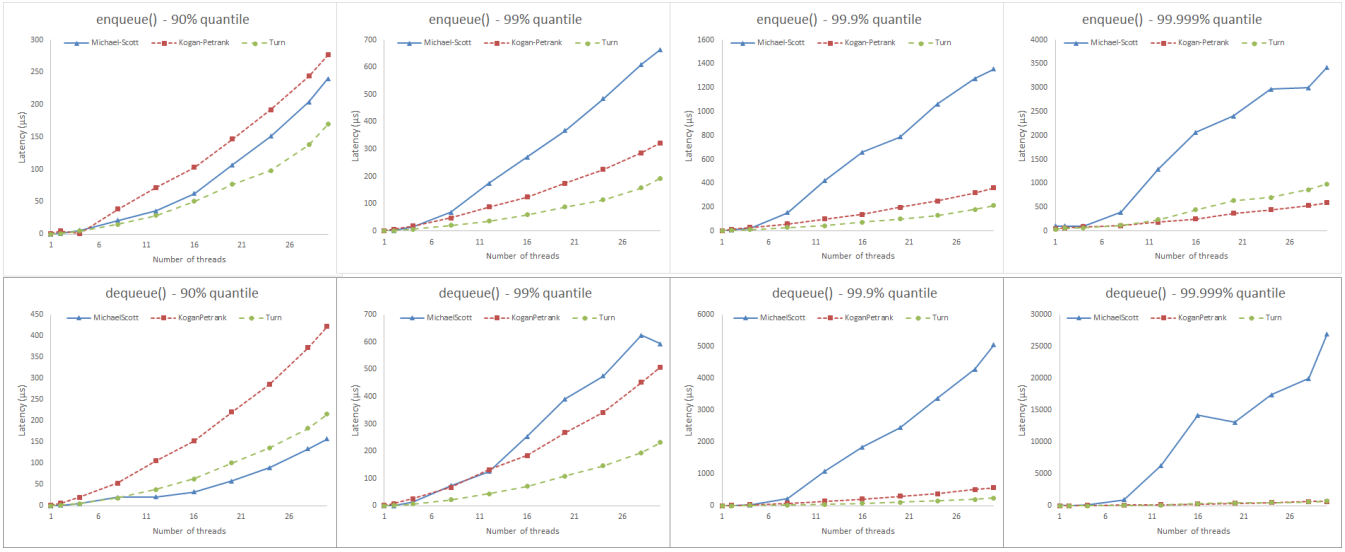

Figure 1. Measurements of the time it takes to complete a call to `enqueue()` or `dequeue()` in microseconds for different quantiles of the latency distribution, as a function of the number of competing threads. Lower is better.

The minimum number of heap allocations (calls to `new`) per item is therefore 5.

For FK, each node contains only the pointer to the item and a pointer to the next node, which results in 16 bytes. This queue requires the usage of one `EnqState` per enqueue and one `DeqState` per dequeue, but these are taken from a pre-allocated pool of such objects with size 16 times the number of threads $N_{threads}$, and therefore, the only real memory allocation is of the node that will be put in the list. Each `DeqState` contains an array with $N_{threads}$ entries where the pointer to the item to be returned by the dequeue operation is placed, which causes the minimum memory usage for this queue to be quadratic with the number of threads $O(N_{threads}^2)$.

In the YMC queue, each node contains a pointer to the next node, a `ticket`, and one `cell_t` which itself has a pointer to the item, a pointer to the enqueue request, and a pointer to the dequeue request, thus resulting in a size of 40 bytes. Several cells can be placed in a node, which amortizes the cost of the next pointer and the ticket (16 bytes), but each cell still takes 24 bytes, and it is only fair to compare with the other queues if the number of cells per node is 1. The actual enqueueing/dequeueing requires the creation of an enqueue/dequeue request, which added to the allocation of the node, means that three objects are allocated per item that is inserted and removed from the queue.

A node in the Turn queue has a pointer to the item, a pointer to the next node, an `enqTid` and an `deqTid`, which results in 24 bytes. The Turn queue does not require an object for the enqueue request or dequeue request, and therefore, the only memory allocation that needs to be done is for the node in the `enqueue()` method.

Any production application using queues will typically have multiple instances, sometimes thousands of queue instances. By design, a queue is not meant to store data for a long period of time, therefore, it is reasonable to expect that most of the queue instances will have items during small bursts of time when compared to the entire time the application runs. Any fixed or pre-allocated objects that need to be kept even when the queue is empty, are a cost in memory that needs to be multiplied by the number of queue instances in the application. This is represented in the fourth row of table 4 where we show on a per-thread basis how much memory is being used by an empty queue.

4.3 Portability

Instead of algorithms with pseudo-code, we showed C++14 code with embedded memory reclamation. We chose C++14 because it is a native language that is supported by most recent compilers and has a well defined memory model and atomics API, which makes the code portable across different architectures without any modification and without introducing CPU specific memory fences. By default, in the C++1x memory model, loads, stores, Compare-And-Swap (CAS) operations on atomic variables are sequentially consistent. Where relaxation is permitted we pass the appropriate `std::memory_order_release` or `std::memory_order_relaxed`.

As shown in table 1, our Turn queue and the KP queue are the only ones that can be deployed on multiple CPU architectures, provided that there is respectively, a recent C++ compiler or a JVM implementation. For correctness, the FK and YMC queues would require adding memory fences specific to the CPU that they are ported to, and they require that the target CPU provides an atomic wait-free FAA instruction for the queue to remain wait-free.

4.4 Throughput

For the throughput measurements we use two different microbenchmarks. The first microbenchmark is similar to [20] where each thread does one pair of operations, consisting of an enqueue followed by a dequeue. Each thread executes $10^8/N_{threads}$ pairs of such operations until it completes. This procedure is repeated 5 times and the median of the 5 runs is plotted on Figure 2. The throughput of the Turn queue ranges from 2x to 5x of the KP queue.

On the second microbenchmark we do a burst of enqueues, wait for all threads to complete, then a burst of dequeues, wait for all threads to complete, and then again a burst of enqueues, and so on, for 10 iterations. Each burst will enqueue (or dequeue) one million items in the queue, divided by the number of threads, which means that at most, there will be one million items present in the queue. We measure the time each burst takes to complete. Before executing the 10 bursts we do one burst for warmup without measuring. The results can be see in Figure 3.

The second microbenchmark procedure, with bursts, has two advantages over the first microbenchmark, with a single enqueue

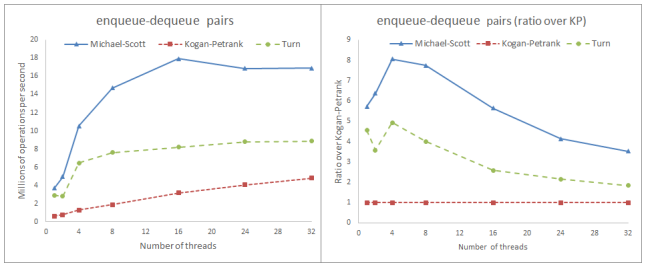

Figure 2. Number of operations per second for each of the three queues. Each data point is the median of 5 runs of 100 million of pairs of enqueue/dequeue. On the right we show the ratio of left plot, normalized by the values for the KP queue. Higher is better.

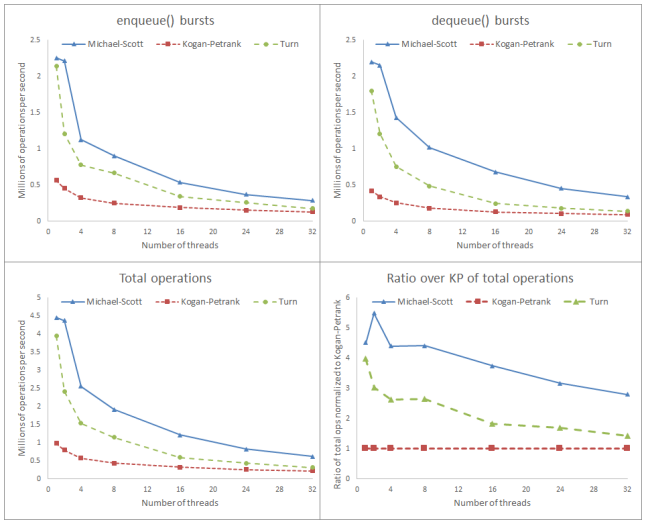

Figure 3. Results for burst microbenchmark. We can measure enqueues and dequeues separately. Higher is better.

and single dequeue: it has enough time granularity for a precise measurement of each of the operations `enqueue()` and `dequeue()`, which are shown in the top two plots of Figure 3; the other advantage is that it guarantees that all threads are either enqueueing or all dequeueing, while the single-enqueue-single-dequeue has a propensity to have about half the threads doing enqueues and the other half doing dequeues, which artifically reduces contention and provides less deterministic results.

Looking at the ratios for both microbenchmarks in Figure 2 and Figure 3, we can see that the Turn queue has always better throughput than the other wait-free queue (KP) with 1.4x to 4x. The results from our microbenchmark show that the Turn queue has a small performance penalty on the uncontended scenario (1 thread), and drops to about half the throughput of the MS queue as the number of threads increases. In scenarios where fairness is desired, with low tail latency, without paying an excessive cost in throughput, the Turn queue can be a good fit.

Source code is available at `https://github.com/pramalhe/ConcurrencyFreaks/tree/master/CPP/papers/crturnqueue`

5. Conclusion

Lock-free queues have an infinite tail in the latency distribution, and wait-free queues, particularly wait-free bounded, have a tail cut-off guarantee. This reason alone is enough motivation to chose a wait-free queue over a lock-free one when low latency in the tail is desired, or if a strong fairness is needed.

It is perhaps unsurprising that out of the three known MPMC wait-free queues (KP, FK, and YMC), there is only one that passes our extensive set of stress tests, the one by Alex Kogan and Erez Petrank, which we believe to be the simplest and more elegant of the three.

Until now, there was no freely available correct implementation of a memory-unbounded Multi-Producer-Multi-Consumer wait-free queue. Even the KP queue requires an automatic Garbage Collector (GC), and as the authors themselves mention on their paper, there is no algorithm for a wait-free GC in the known literature.

The enqueue and dequeue algorithms of the KP queue have been used as a building block of other wait-free data structures, such as: a wait-free stack [9], a generic wait-free construct [4], a wait-free list [25]. The Turn queue can be used as a replacement, to provide higher throughput for these applications, and to allow for their implementations in a non-GC runtime.

We have presented here the first linearizable memory-unbounded MPMC wait-free queue with an accompanying wait-free memory reclamation system, such that it can be used in C/C++ and other languages with or without a GC. We have shown the importance for tail latency of having a progress that is wait-free bounded by the number of threads. The Turn queue's main characteristics are: it uses a new fast wait-free consensus protocol with a bound on the number of threads; does no memory allocation apart from `enqueue()` creating the node that is placed in the queue; achieves wait-free progress without requiring no other atomic instruction besides compare-and-swap (CAS); the algorithm is simple; enqueuers only help other enqueuers, and dequeuers only help other dequeuers; the algorithm for enqueueing is independent from the algorithm for dequeuing which means it can used to make a SPMC or MPSC queue, or plugged in with other enqueuing/dequeueing algorithms that use singly-linked lists; we provide an implementation that uses the C++1x memory model and therefore, can be deployed in a multitude of different architectures, working on any target that has a C++14 compiler.

References

[1] D. Alistarh, P. Eugster, M. Herlihy, A. Matveev, and N. Shavit. Stacktrack: An automated transactional approach to concurrent memory reclamation. In *Proceedings of the Ninth European Conference on Computer Systems*, page 25. ACM, 2014.

[2] A. Braginsky, A. Kogan, and E. Petrank. Drop the anchor: lightweight memory management for non-blocking data structures. In *Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures*, pages 33–42. ACM, 2013.

[3] P. A. Buhr, D. Dice, and W. H. Hesselink. High-performance n-thread software solutions for mutual exclusion. *Concurrency and Computation: Practice and Experience*, 27(3):651–701, 2015.

[4] A. Correia and P. Ramalhete. Cowmutationq - a copy-on-write technique with wait-free progress. `https://github.com/pramalhe/ConcurrencyFreaks/blob/master/papers/cowmq-2015.pdf`, 2015.

[5] A. Correia and P. Ramalhete. Mutual exclusion - two linear wait software solutions. `https://github.com/pramalhe/ConcurrencyFreaks/blob/master/papers/cralgorithm-2015.pdf`, 2015.

[6] P. Fatourou and N. D. Kallimanis. A highly-efficient wait-free universal construction. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, pages 325–334. ACM, 2011.

[7] P. Fatourou and N. D. Kallimanis. Revisiting the combining synchronization technique. In *ACM SIGPLAN Notices*, volume 47, pages 257–266. ACM, 2012.

[8] P. Fatourou and N. D. Kallimanis. simqueue.c. `https://github.com/nkallima/sim-universal-construction/blob/master/sim-synch1.4/simqueue.c`, 2016.

[9] S. Goel, P. Aggarwal, and S. R. Sarangi. A wait-free stack. In *International Conference on Distributed Computing and Internet Technology*, pages 43–55. Springer, 2016.

[10] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC*, volume 1, pages 300–314. Springer, 2001.

[11] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(5):745–770, 1993.

[12] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Elsevier, 2012.

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

[14] M. Herlihy, V. Luchangco, P. Martin, and M. Moir. Nonblocking memory management support for dynamic-sized data structures. *ACM Transactions on Computer Systems (TOCS)*, 23(2):146–196, 2005.

[15] N. D. Kallimanis. sim-universal-construction. `https://github.com/nkallima/sim-universal-construction`, 2016.

[16] A. Kogan and E. Petrank. Wait-free queues with multiple enqueuers and dequeuers. In *ACM SIGPLAN Notices*, volume 46, pages 223–234. ACM, 2011.

[17] A. Kogan and E. Petrank. A methodology for creating fast wait-free data structures. In *ACM SIGPLAN Notices*, volume 47, pages 141–150. ACM, 2012.

[18] L. Lamport. A new solution of dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, 1974.

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

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

[21] A. Morrison and Y. Afek. Fast concurrent queues for x86 processors. In *ACM SIGPLAN Notices*, volume 48, pages 103–112. ACM, 2013.

[22] E. Petrank. Can parallel data structures rely on automatic memory managers? In *Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, pages 1–1. ACM, 2012.

[23] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Address-sanitizer: A fast address sanity checker. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 309–318, 2012.

[24] N. Shavit and G. Taubenfeld. The computability of relaxed data structures: queues and stacks as examples. In *International Colloquium on Structural Information and Communication Complexity*, pages 414–428. Springer, 2014.

[25] S. Timnat. Practical parallel data structures. `http://www.cs.technion.ac.il/users/wwwb/cgi-bin/tr-get.cgi/2015/PHD/PHD-2015-06.pdf`, 2016.

[26] D. Vyukov. mpsc node based queue. `http://www.1024cores.net/home/lock-free-algorithms/queues/non-intrusive-mpsc-node-based-queue`, 2016.

[27] C. Yang and J. Mellor-Crummey. A wait-free queue as fast as fetch-and-add. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, page 16. ACM, 2016.