A Wait-free Queue as Fast as Fetch-and-Add

Chaoran Yang John Mellor-Crummey
Department of Computer Science, Rice University
{chaoran, johnmc}@rice.edu

Abstract

Concurrent data structures that have fast and predictable performance are of critical importance for harnessing the power of multi-core processors, which are now ubiquitous. Although wait-free objects, whose operations complete in a bounded number of steps, were devised more than two decades ago, wait-free objects that can deliver scalable high performance are still rare.

In this paper, we present the first wait-free FIFO queue based on `fetch-and-add` (FAA). While `compare-and-swap` (CAS) based non-blocking algorithms may perform poorly due to work wasted by CAS failures, algorithms that coordinate using FAA, which is guaranteed to succeed, can in principle perform better under high contention. Along with FAA, our queue uses a custom epoch-based scheme to reclaim memory; on x86 architectures, it requires no extra memory fences on our algorithm's typical execution path. An empirical study of our new FAA-based wait-free FIFO queue under high contention on four different architectures with many hardware threads shows that it outperforms prior queue designs that lack a wait-free progress guarantee. Surprisingly, at the highest level of contention, the throughput of our queue is often as high as that of a microbenchmark that only performs FAA. As a result, our fast wait-free queue implementation is useful in practice on most multi-core systems today. We believe that our design can serve as an example of how to construct other fast wait-free objects.

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

Keywords non-blocking queue, wait-free, fast-path-slow-path

1. Introduction

Over the last decade, multi-core processors have become ubiquitous; they are employed in a spectrum of computer systems that range from low-end microcontrollers to supercomputers. Furthermore, computer systems in which two or more multi-core processors share memory are common. Finally, modern processors often employ multiple hardware threads per core to hide latency. As a result, concurrent data structures that can deliver high performance while being manipulated by multiple threads are essential for programming such systems efficiently. Besides high performance, another important characteristic of a concurrent data structure is its *progress guarantee*. Concurrent data structures can be classified as either *blocking* or *non-blocking*. Blocking data structures include at least one operation where a thread may need to wait for an operation by another thread to complete. Blocking operations can introduce a variety of subtle problems, including deadlock, livelock, and priority inversion; for that reason, non-blocking data structures are preferred.

There are three levels of *progress guarantees* for non-blocking data structures. 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 *some* thread performing an arbitrary operation on the object will complete in a *finite* number of steps, or
- *wait-free* if *every* thread can perform an arbitrary operation on the object in a *finite* number of steps.

Wait-freedom is the strongest progress guarantee; it rules out the possibility of starvation for all threads. Wait-free data structures are particularly desirable for mission critical applications that have real-time constraints, such as those used by cyber-physical systems.

Although universal constructions for wait-free objects have existed for more than two decades [11], practical wait-free algorithms are hard to design and considered inefficient with good reason. For example, the fastest wait-free concurrent queue to date, designed by Fatourouto and Kallimanis [7], is orders of magnitude slower than the best performing lock-free queue, LCRQ, by Morrison and Afek [19]. General methods to transform lock-free objects into wait-free objects, such as the *fast-path-slow-path* methodology by Kogan and Petrank [14], are only suitable for lock-free data structures that are updated using a series of `compare-and-swap` (CAS) operations. CAS-based data structures, as Morrison and Afek have pointed out [19], may perform poorly under high contention due to *work wasted* by CAS failures. It remains an open question how to transform lock-free data structures that use *read-modify-write* primitives other than CAS into wait-free forms.

This paper presents the first design of a concurrent FIFO queue based on `fetch-and-add` (FAA) that is *linearizable*, *wait-free*, and *fast*. An empirical study of our queue under high contention on four different architectures with many hardware threads shows that it outperforms prior queue designs that lack wait-free semantics. Surprisingly, at the highest level of contention, throughput of our queue is often as high as that of a microbenchmark that only performs FAA. This disproves the myth that ensuring wait-freedom and linearizability together is too expensive to be efficient. Our design employs the fast-path-slow-path methodology [14] to transform a simple yet efficient FAA-based obstruction-free queue into a wait-free queue. Our queue employs a custom epoch-based scheme to reclaim memory. On x86 architectures, our memory reclamation scheme has lower overhead than other schemes since it adds no memory fence along common execution paths in queue operations. We believe that our design can serve as an example of how to construct other fast wait-free objects.

PPoPP '16 March 12–16, 2016, Barcelona, Spain
Copyright © 2016 ACM 978-1-4503-4092-2/16/03. . . $15.00
DOI: http://dx.doi.org/10.1145/2851141.2851168

The rest of this paper is organized as follows. Section 2 describes previous work on concurrent queues and wait-free objects. Section 3 presents our new wait-free queue and its memory reclamation scheme. Section 4 proves its correctness and wait-freedom. Section 5 presents a statistically rigorous study of our queue's performance on various platforms. Section 6 presents our conclusions.

2. Related Work

Concurrent FIFO queues have been a topic of active research for decades. Michael and Scott's lock-free queue [17], known as MS-Queue, is considered a classic non-blocking queue. MS-Queue is a singly-linked list that maintains pointers to the head and tail of the list. Its enqueue and dequeue methods manipulate the head and tail pointers using CAS operations in a retry loop. Under heavy contention, most CAS operations will fail, causing a dramatic drop in performance. Morrison and Afek dub this *the CAS retry problem* [19]. For this reason, the performance of MS-Queue does not scale well beyond a modest number of threads.

The first practical implementation of a wait-free queue was proposed by Kogan and Petrank [13]. Their queue is based on MS-Queue. To achieve wait-freedom, it employs a priority-based helping scheme in which faster threads help slower threads complete their pending operations. In most cases, this queue does not perform as well as the MS-Queue due to the overhead of its helping mechanism.

Many universal constructions have been proposed to create wait-free objects. These constructions can transform any sequential object into a lock-free or wait-free object. Herlihy proposed the first such universal construction in 1991 [11]. Since then, several faster universal constructions for wait-free objects have been devised [1, 2, 4, 18]; however, universal constructions are hardly considered practical in general. The first practical wait-free universal construction, called P-Sim, was proposed by Fatourou and Kallimanis in 2011 [7]. P-Sim uses FAA in addition to CAS to achieve wait-freedom. The wait-free queue constructed using P-Sim outperformed all prior designs for wait-free queues and MS-Queue.

In 2012, Fatourou and Kallimanis proposed a blocking queue, called CC-Queue [8], which delivers better performance than their prior design for a wait-free queue. CC-Queue uses *combining*, where all threads with a pending operation form a queue and the thread at the head of the queue performs operations for all other threads in the queue. Although this combining strategy has lower synchronization overhead, CC-Queue is not non-blocking, and it sacrifices parallelism which limits its performance.

In 2013, Morrison and Afek proposed LCRQ [19], a lock-free queue. This queue is a linked-list of segments, where each segment is a circular ring queue (CRQ). LCRQ uses FAA to acquire an index on a CRQ and then uses a double-width `compare-and-swap` (CAS2) to enqueue to and dequeue from the cell at the obtained index. The list of CRQs in LCRQ is managed like the linked-list in the MS-Queue. Because the head and tail hot spots in LCRQ are updated using FAA, LCRQ avoids the CAS retry problem and outperforms CC-Queue and all other non-blocking queues to date. However, LCRQ is a lock-free queue and its application is limited by its use of CAS2, which is not universally available.

In 2012, Kogan and Petrank proposed a *fast-path-slow-path* methodology for creating practical wait-free objects [14]. The fast-path, usually a lock-free algorithm, provides good performance; the slow-path ensures wait-freedom. Each operation attempts the fast-path several times; if all attempts fail, it switches to the slow-path. Their wait-free queue design using this methodology employs an MS-Queue as its fast-path; as a result, its performance is only as good as MS-Queue. In 2014, Timnat and Petrank showed that, using fast-path-slow-path, a lock-free object expressed in a *normalized form* can be automatically transformed into a practical wait-free object that is as fast as the lock-free one [21]. Their normalized form requires that the lock-free object be updated using a series of CAS operations. Hence, this automatic transformation cannot be applied to LCRQ to construct a wait-free version. It is unclear how the fast-path-slow-path methodology can be applied to non-blocking algorithms that perform updates using primitives such as FAA, which always succeed.

3. A Fast Wait-Free Queue

This section presents our design for a wait-free FIFO queue. Section 3.1 introduces concepts that provide a foundation for our design. Section 3.2 presents the high-level design of our wait-free queue. Sections 3.3–3.5 present our queue data structure, its operations, and their properties in detail. Section 3.6 describes a novel memory reclamation scheme used by our queue.

3.1 Preliminaries

We design our queue for an *asynchronous shared memory system* [12]. On such a system, a program is executed by p deterministic threads, where p may exceed the number of physical processors. Threads communicate via atomic operations on a predefined set of shared memory locations. A scheduler decides which threads to run and the mapping between threads and processors. We assume the scheduler may suspend execution of any thread at any time for arbitrarily long.

Memory model When writing code for concurrent data structures, one must mark shared variables as *volatile* to ensure that variable reads and writes access memory rather than just a thread's registers. To simplify the presentation of pseudo-code for our queue, we assume a *sequentially consistent*[1] memory model. However, today's programming languages and architectures provide weaker ordering guarantees and allow accesses to be reordered for performance. To prevent any undesired reorderings by a compiler or hardware, atomic operations or memory fences are necessary to guarantee the order of accesses to different shared memory locations.

We omit volatile declarations and memory fences in this paper and present them in the source code provided as supplemental material for the paper.[2]

Atomic primitives We model memory as an array of 64-bit values. We use the notation $m[a]$ for the value stored in address a of the memory. Algorithms in this paper use the following 64-bit atomic primitives:

- `read`, denoted as $x := m[a]$, which returns $m[a]$;
- `write`, denoted as $m[a] := x$, which stores x into $m[a]$;
- `FAA(a, v)`, which returns $m[a]$ and stores $m[a] + v$ at a;
- `CAS(a, t, v)`, denoted as $(a : t \mapsto v)$, which stores v into $m[a]$ and returns `true` if $m[a] = t$; otherwise, it returns `false`.

64-bit architectures from AMD and Intel support FAA and CAS natively. IBM Power architectures lack native support for FAA and CAS, but one can emulate them using `load-linked/store-conditional` (LL/SC). Although emulating FAA and CAS with LL/SC retry loops sacrifices the wait freedom of our queue on Power7, our algorithm still performs well on Power7 in practice.

[1] Sequential consistency [15] means that the result of any execution is the same as if all memory accesses by threads were executed in some sequential order, and the accesses of each individual thread appear in this sequence in the order specified by the program.

[2] In addition to the supplemental material in the ACM Digital Library, source code for our queue is also available at `https://github.com/chaoran/fast-wait-free-queue`.

```
1  enqueue(x: var) {
2    do t := FAA(&T, 1);
3    while (!CAS(&Q[t], ⊥, x));
4  }

5  dequeue(): var {
6    do h := FAA(&H, 1);
7    while (CAS(&Q[h], ⊥, ⊤) and T > h);
8    return (Q[h] = ⊤ ? EMPTY : Q[h]);
9  }
```

Listing 1. An obstruction-free queue using an infinite array.

Linearizability Stated informally, a FIFO queue is an object whose state Q is a sequence of zero or more values. It supports an $enqueue(x)$ operation that appends x to Q and a $dequeue()$ operation that removes the first value in Q and returns it, or returns EMPTY if Q is an empty sequence. The correctness condition we use for our concurrent queue design is *linearizability* [12], which means that each operation on our queue must appear to "take effect" instantaneously at a point in time within the interval during which the operation executes. A *linearizable* FIFO queue is a queue for which any concurrent operation history has a linearization that conforms to a correct sequential operation history of a FIFO queue.

3.2 High-level Design of the Queue

Our queue can be viewed as a wait-free realization of the non-blocking queue shown in Listing 1. This is similar to the base algorithm used for LCRQ [19]. Listing 1 represents a queue using an *infinite* array, Q, with unbounded *head* and *tail* indices, H and T, which mark the boundaries of the interval in Q that may contain values. We reserve two special values, $\bot$ (bottom) and $\top$ (top), which may not be enqueued. Initially, each cell $Q[i]$ contains a $\bot$. A dequeue may update a cell's value to $\top$ to mark it *unusable*, which prevents enqueues from putting a value into that cell.

An enqueue(x) obtains a cell index t by performing a FAA on T. Then it attempt to perform $(Q[t] : \bot \mapsto x)$. If the CAS succeeds, the enqueue operation completes. Otherwise, it tries again.

Similarly, a dequeue obtains a cell index h by performing a FAA on H. The dequeue repeats the FAA until either (1) $(Q[h] : \bot \mapsto \top)$ fails, or (2) it sees $T \leq h$. If (1), it means that an enqueued value is available in $Q[h]$ and the dequeue can return it; if (2), it means the queue is empty, the dequeue can return EMPTY.

This simple queue is *non-blocking* because neither a suspended enqueue or dequeue will block other operations. More precisely, the queue is obstruction-free because it is susceptible to livelock. For example, given an enqueuer E, a dequeuer D, and a queue with state $T = H$, if E and D interleave repeatedly in the following order, neither will make progress: (1) E performs a FAA on T, (2) D performs a FAA on H, (3) D performs a CAS, (4) E performs a CAS, and (5) D reads T.

We adopt Kogan and Petrank's *fast-path-slow-path* methodology to construct our wait-free queue. Our wait-free queue uses operations like those in Listing 1 as the fast-path. We designed compatible slow-path implementations of these operations to ensure wait-freedom. An operation on the queue first tries its fast-path implementation until it succeeds or the number of failures exceeds a threshold. If necessary, it falls back to its slow-path, which guarantees completion within a finite number of attempts. We now describe the high-level design of our wait-free queue's implementation.

Emulate an infinite array We emulate an infinite array using a singly-linked list of equal-sized array segments. Each thread maintains two local pointers, $head$ and $tail$, to segments in the list. A thread uses its $head$ pointer for dequeues and its $tail$ pointer for enqueues. To find a cell $Q[i]$ in the queue, a thread traverses the list starting with the segment at $head$ or $tail$ until it reaches the segment containing $Q[i]$. The traversing thread appends new segments if it reaches the end of the list before it finds $Q[i]$. Segments that are no longer in use are removed from the front of the list. To support memory reclamation, each thread maintains a hazard pointer. A thread publishes its $head$ or $tail$ pointer in the hazard pointer at the start of a dequeue or an enqueue and clears it after an operation. Periodically, a thread will scan every thread's $head$, $tail$, and hazard pointer to find the earliest segment that is in use by any thread and reclaim all preceding segments. Section 3.6 describes the details of our memory reclamation scheme.

Ensure wait-freedom Using Herlihy's approach [11], one makes an operation wait-free by turning its *contenders*—concurrent operations that prevent the operation from succeeding—into helpers, who help the operation to complete. For example, an enqueue on the fast-path may never succeed because contending dequeues mark every cell they visit as unusable. To make enqueues wait-free, a slow-path enqueue enlists the help of contending dequeuers by publishing an enqueue request. Initially, the local states of threads are linked in a ring; each dequeuer keeps a pointer to the state of an enqueue peer. Whenever a dequeue marks a cell unusable, it helps its enqueue peer if the peer has a pending enqueue request. When the peer's request completes, the dequeuer updates its peer pointer to the next enqueuer in the ring. Hence every time an enqueue fails, its contending dequeuer will help some enqueue request to complete. Eventually each pending enqueue request will get help from contending dequeuers. When all contending dequeuers become the enqueue's helpers, the enqueue will definitely succeed.

Similarly, a fast-path dequeue may never succeed because enqueued values are taken by contending dequeues. To make dequeues wait-free, each dequeuer maintains a pointer to a dequeue peer. Whenever a dequeue successfully takes a value, it helps its dequeue peer if the peer has a pending dequeue request. A slow-path dequeue and its helpers traverse the queue to find a value to dequeue. A dequeue on the slow-path will eventually succeed because all contending dequeues will help it to dequeue.

3.3 Building Blocks

Global state Listing 2 shows the data types used to implement our wait-free queue. The queue itself is represented by a triple (Q, H, T). Q is a singly-linked list of segments. Each segment in the list consists of (1) a unique id, (2) a $next$ pointer, and (3) an N-vector of $cells$. Initially, Q points to a segment with id zero. We denote the segment with id i as $segment[i]$ and the j-th cell in a segment as $cell[j]$. T and H are 64-bit head and tail indices used to index cells in the queue. Both H and T are zero initially. A cell at index i, known as $Q[i]$, corresponds to $cell[i \mod N]$ in $segment[i/N]$.

Thread-local state The local state of each thread is stored in a *handle*. Each handle contains $head$ and $tail$ pointers to segments in the queue; both point to $segment[0]$ initially. We maintain head and tail pointers to queue segments as part of each thread's local state to avoid contention that would inevitably arise if we maintained them as part of the queue's shared state. During initialization, all thread handles are linked in a ring using their $next$ pointers; this enables any thread to access the state of other threads.

A thread's enqueue state (Handle.enq) consists of an enqueue request req, a $peer$, and an id. The req structure is used to request help from other threads. The $peer$ pointer is used to identify another thread's request that might need help. The id is used to record the index of a pending request that the thread is helping. A thread's dequeue state also contains req and $peer$ fields. Physically, both enqueue and dequeue request are two 64-bit words. Logically, an

```
10  struct EnqReq
11    void *val;
12    struct { int pending : 1; int id : 63; } state;

13  struct DeqReq
14    int id : 64;
15    struct { int pending : 1; int idx : 63; } state;

16  struct Cell
17    void *val; EnqReq *enq; DeqReq *deq;

18  struct Segment
19    int id : 64; Segment *next; Cell cells[N];

20  struct Queue
21    Segment *Q; int T : 64; int H : 64;

22  struct Handle
23    Segment *tail, *head;
24    Handle *next;
25    struct { EnqReq req; Handle * peer; } enq;
26    struct { DeqReq req; Handle * peer; } deq;

27  Segment *new_segment(int id)
28    Segment *s := new Segment;
29    s->id := id;
30    s->next := null;
31    for (i := 0; i < N; i++) s->cells[i] := (⊥, ⊥e, ⊥d);
32    return s;

33  Cell *find_cell(Segment **sp, int cell_id)
34    s := *sp; // invariant: sp points to a valid segment
35    // traverse list to target segment with id cell_id/N
36    for (i := s->id; i < cell_id/N; i++) {
37      next := s->next;
38      if (next = null) {
39        // the list needs another segment.
40        // allocate one and try to extend the list.
41        tmp := new_segment(i + 1);
42        if (!CAS(&s->next, null, tmp)) {
43          free(tmp); // another thread extended the list
44        }
45        // invariant: a successor segment exists
46        next := s->next;
47      }
48      s := next;
49    }
50    // invariant: s is the target segment (cell_id/N)
51    *sp := s; // return target segment
52    return &s->cells[cell_id mod N]; // return target cell

53  void advance_end_for_linearizability(int *E, int cid)
54    // ensure head or tail index (*E) is at or beyond cid
55    do e := *E; while (e < cid and !CAS(E, e, cid));
```

Listing 2. Structures and auxiliary methods.

enqueue request is a triple $(val, pending, id)$, where val is a 64-bit value to enqueue, $pending$ is a bit flag, and id is a 63-bit integer. We also refer to ($pending$, id) as an enqueue request's $state$. An enqueue request is initially $(\bot, 0, 0)$, where $\bot$ is a reserved value that is distinct from any value that might be enqueued. Over time, a thread's enqueue state will be rewritten to represent different requests; the id field distinguishes different requests from the same thread. Similarly, a dequeue request is a triple $(id, pending, idx)$ where id is a 64-bit integer, $pending$ is a bit flag, and idx is a 63-bit integer indicating the index of a cell that is the target of a dequeue. A dequeue request's id and $pending$ serve the same purpose as those in an enqueue request. We call ($pending$, idx) a dequeue request's $state$. A dequeue request is initially $(0, 0, 0)$.

Cells Each cell in the queue is a triple (val, enq, deq). val is a value. enq and deq are pointers to enqueue and dequeue requests, respectively. Each component of a cell may also be a reserved value. For val, we reserve $\bot$ and $\top$. For enq, we reserve $\bot_e$ and $\top_e$. For deq, we reserve $\bot_d$ and $\top_d$. Every cell is $(\bot, \bot_e, \bot_d)$ initially.

The `find_cell` function locates a cell given sp, a pointer to a segment, and $cell\_id$, the index of the cell. `find_cell` traverses the queue starting at segment sp until it finds the segment with id $cell\_id/N$, i.e., the segment containing cell $cell\_id$. If `find_cell` reaches the end of the segment list before it finds the cell, it allocates a new segment, initializes it, and then tries to append it to the list using a CAS on the last segment's $next$ pointer. If the CAS fails, some other thread has already extended the segment list, so the thread frees the segment it failed to append. The thread continues the traversal with the successor segment now available. Eventually, the thread will find $segment[cell\_id/N]$ and return $cell[cell\_id \mod N]$. Note that `find_cell` has a side-effect: it updates the provided segment pointer to point to the segment that contains the returned cell (line 51).

3.4 Wait-free Enqueue

Listing 3 shows the wait-free code for `enqueue`. An enqueue first attempts to enqueue a value via the fast-path (`enq_fast`) until it succeeds or its "patience" runs out. If necessary, it will finish the enqueue using the slow-path (`enq_slow`), which is guaranteed to succeed. `enq_fast` is like `enqueue` in Listing 1; however, if it fails, it will return the cell index it obtained with FAA in an output parameter `id`. This index becomes the id of an enqueue request used by the slow-path.

When necessary, an enqueue begins its slow-path by by publishing its enqueue request in its thread handle's `enq.req` field. The state of a pending enqueue request is $(v, 1, id)$, where v is the value to enqueue, id is a cell index that serves as the request's identifier, and $pending = 1$. After publishing its enqueue request to solicit help, a slow-path enqueue continues trying to succeed on its own by obtaining indices of additional cells in the queue using FAA and trying to deposit its value into each candidate cell. This process continues until the enqueue deposits its value into a cell or a contending dequeuer helps it to complete. Each dequeuer tries to help enqueue requests by calling `help_enq` on each cell it visits. Given a cell $Q[i]$ and a pointer r to an enqueue request, an enqueuer or its helpers perform the following steps to enqueue:

1. reserve $Q[i]$ using $(Q[i].enq : \bot_e \to r)$,
2. claim r for $Q[i]$ using $(r\text{->}state : (1, id) \to (0, i))$,
3. write v in $Q[i']$ using $Q[i'].val := v$, where $i' = r\text{->}state.id$.

A cell's enqueue result state A cell's state is initially $(\bot, \bot_e, \bot_d)$. When an enqueuer (`enq_fast` or `enq_slow`) or an enqueue helper (`help_enq`) finishes with a cell, it will transition the cell into one of the following states:

- $(v, \bot_e, \bot_d)$: an enqueue succeeded using the fast-path,
- $(v, r, \bot_d)$: an enqueue succeeded using the slow-path, or
- $(\top, r, \bot_d)$ or $(\top, \top_e, \bot_d)$: an enqueue failed and a helper may return `EMPTY` at the cell.

We call these four states a cell's *enqueue result states*. The enqueue routines and `help_enq` maintain the following important invariant:

Invariant 1. *A cell in an enqueue result state cannot be changed by future enqueuers or enqueue helpers.*

Invariant 1 ensures that when an enqueue routine (`enq_fast` or `enq_slow`) and multiple `help_enq` are invoked on the same cell, they will produce one and only one unique enqueue result state for that cell.

Avoid enqueueing into an unusable cell An unusable cell is a cell that has been abandoned by dequeuers; a value written in an unusable cell will not be dequeued. Therefore, an enqueuer must not enqueue into a cell that has been marked unusable. To do this,

```
56 void enqueue(Queue *q, Handle *h, void *v)
57   for (p := PATIENCE; p >= 0; p--)
58     if (enq_fast(q, h, v, &cell_id)) return;
59   enq_slow(q, h, v, cell_id); // use id from last attempt

60 bool try_to_claim_req(int *s, int id, int cell_id)
61   return CAS(s, (1, id), (0, cell_id));

62 void enq_commit(Queue *q, Cell *c, void *v, int cid)
63   advance_end_for_linearizability(&q->T, cid+1);
64   c->val := v; // record value in claimed cell

65 bool enq_fast(Queue *q, Handle *h, void *v, int *cid)
66   // obtain cell index and locate candidate cell
67   i := FAA(&q->T, 1); c := find_cell(&h->tail, i);
68   if (CAS(&c.val, ⊥, v)) return true; // enq complete
69   *cid := i; return false; // fail, returning cell id

70 void enq_slow(Queue *q, Handle *h, void *v, int cell_id)
71   // publish enqueue request
72   r := &h->enq.req; r->val := v; r->state := (1, cell_id);
73   // use a local tail pointer to traverse because
74   // line 87 may need to find an earlier cell.
75   tmp_tail := h->tail;
76   do {
77     // obtain new cell index and locate candidate cell
78     i := FAA(&q->T, 1); c := find_cell(&tmp_tail, i);
79     // Dijkstra's protocol
80     if (CAS(&c->enq, ⊥e, r) and c.val = ⊥) {
81       try_to_claim_req(&r->state, id, i); // for cell i
82       // invariant: request claimed (even if CAS failed)
83       break;
84     }
85   } while (r->state.pending);
86   // invariant: req claimed for a cell; find that cell
87   id := r->state.id; c := find_cell(&h->tail, id);
88   enq_commit(q, c, v, id);
89   // invariant: req committed before enq_slow returns

90 void *help_enq(Queue *q, Handle *h, Cell *c, int i)
91   if (!CAS(&c->val, ⊥, ⊤) and c->val != ⊤) return c->val;
92   // c->val is ⊤, so help slow-path enqueues
93   if (c->enq = ⊥e) { // no enqueue request in c yet
94     do { // two iterations at most
95       p := h->enq.peer; r := &p->enq.req; s := r->state;
96       // break if I haven't helped this peer complete
97       if (h->enq.id = 0 or h->enq.id = s.id) break;
98       // peer request completed; move to next peer
99       h->enq.id := 0; h->enq.peer := p->next;
100    } while (1);
101    // if peer enqueue is pending and can use this cell,
102    // try to reserve cell by noting request in cell
103    if (s.pending and s.id ≤ i and !CAS(&c->enq, ⊥e, r))
104      // failed to reserve cell for req, remember req id
105      h->enq.id := s.id;
106    else
107      // peer doesn't need help, I can't help, or I helped
108      h->enq.peer := p->next; // help next peer next time
109    // if can't find a pending request, write ⊤e to
110    // prevent other enq helpers from using cell c
111    if (c->enq = ⊥e) CAS(&c->enq, ⊥e, ⊤e);
112  }
113  // invariant: cell's enq is either a request or ⊤e
114  if (c->enq = ⊤e) // no enqueue will fill this cell
115    // EMPTY if not enough enqueues linearized before i
116    return (q->T ≤ i ? EMPTY : ⊤);
117  // invariant: cell's enq is a request
118  r := c->enq; s := r->state; v := r->val;
119  if (s.id > i)
120    // request is unsuitable for this cell
121    // EMPTY if not enough enqueues linearized before i
122    if (c->val = ⊤ and q->T ≤ i) return EMPTY;
123  else if (try_to_claim_req(&r->state, s.id, i) or
124           // someone claimed this request; not committed
125           (s = (0,i) and c->val = ⊤))
126    enq_commit(q, c, v, i);
127  return c->val; // c->val is ⊤ or a value
```

Listing 3. Pseudo code for wait-free enqueue. help_enq is called by a dequeue on each cell from which it tries to obtain a value.

we employ Dijkstra's protocol [5] to synchronize an enqueuer and its helpers. Specifically, after an enqueuer reserves a cell $Q[i]$ for its request r using $(Q[i].enq : \bot_e \mapsto r)$, it reads $Q[i].val$ (line 80); if $Q[i].val$ has been set to $\top$, it aborts enqueueing in $Q[i]$ and restarts the process using a different cell. After a dequeuer marks a cell $Q[i]$ unusable by an enqueuer[3] using $(Q[i].val : \bot \mapsto \top)$, it reads $Q[i].enq$ (line 93); if some request has reserved $Q[i]$, i.e. $Q[i].enq \neq \bot_e$, it helps the request to complete. Dijkstra's protocol ensures that if a dequeuer marks a cell unusable by an enqueuer *after* an enqueuer reserves the cell by recording an enqueue request, the dequeuer will see the pending enqueue request. If the dequeuer sees a pending request, the dequeuer will help complete it.

Ensure wait-freedom To ensure slow-path enqueue is wait-free, enqueue's helpers (help_enq) maintains two key invariants:

Invariant 2. *An enqueue helper continues helping a peer until (1) the peer has published a new request or (2) the peer has no pending request for help.*

An enqueue helper only advances to a new peer on lines 99 and 108. On line 99, an enqueue helper advances to the next peer if the peer has published a new request since it helped last (h->enq.id != s.id). Line 99 only executes once because it establishes a breaking condition for the loop (h->enq.id := 0). On line 108, an enqueue helper advances to the next peer only if the peer doesn't need help (s.pending = 0), the request can't deposit into the current cell because that would violate linearizability (s.id > i) (explained later), or the helper reserved this cell for the request with $(c\text{->}enq : \bot_e \mapsto r)$

Invariant 3. *An enqueue helper does not help a thread again until it has helped all other threads.*

Invariant 2 ensures that an enqueue helper will advance to the next peer when the peer publishes a new request. A helper returns to a peer only after offering help to all other peer threads in the ring.

Ensure linearizability To ensure linearizability, we require that:

Invariant 4. *An enqueue can only deposit a value into a cell $Q[i]$, where $i < T$ when the enqueue completes.*

Invariant 4 is always satisfied when an enqueuer claims a cell for a value because it always obtains indices of cells using FAA on T. An enqueuer claims a cell for a value by either (1) depositing the value into the cell (line 68), or (2) claiming a request for a cell (line 81). If a helper claims an enqueue request, advance_end_for_linearizability is called (lines 88 and 126) to ensure that T is larger than the index of the cell where the value will be deposited.

Invariant 5. *A cell $Q[i]$ can only be reserved for enqueue request r where $r.id \leq i$.*

All cell indices a slow-path enqueue obtains on line 78 with FAA and tries to reserve on line 80 will be larger than the id of the request it published, which is the result of an earlier FAA on T. A helper can only reserve cell $Q[i]$ for an enqueue request r if $r.id \leq i$ (line 103).

Invariant 6. *A enqueue helper may return* EMPTY *to the dequeue that called it on cell $Q[i]$ only if no pending enqueue can deposit a value into $Q[i]$ and the helper sees $T \leq i$.*

A call to help_enq will return EMPTY iff $T \leq i$ and the cell is in enqueue result state (1) $(\top, r, \bot_d)$, where $r\text{->}id > i$ (line 122), or (2) $(\top, \top_e, \bot_d)$ (line 116).

[3] Although a cell may be unusable by an enqueuer, a helping dequeue may put a value there because at least one dequeue (itself) will try to consume it.

Write the proper value in a cell Since an enqueue request is two 64-bit words that aren't read or written atomically, when an enqueue helper reads the two words, it must identify if both belong to the same logical request and not write a cell value if they don't. A slow-path enqueue will never overwrite its thread's enqueue request before it finishes; thus, mismatch is only a potential problem with a helper. A helper reads an enqueue request's two words (line 118) in the reverse order they were written (line 72). Since the helper reads $r.state$ into s first, then it reads $r.val$ into v, value v belongs to request $s.id$ or a later request. In line 126, a helper commits v to cell i only if (1) it claims the request for cell i (line 123), or (2) someone else has claimed the request for cell i and no value has yet been written into cell i (line 125). If either of those conditions is true, the enqueuer has not finished yet, so the value v previously read in line 118 must be for request $s.id$.

3.5 Wait-free Dequeue

Listing 4 shows routines for our wait-free dequeue. A dequeue is in search of a *candidate cell* that has an unclaimed value or permits returning EMPTY. A dequeue first uses the fast-path (deq_fast) to dequeue until it succeeds or its "patience" runs out. If necessary, it finishes the dequeue using the slow-path (deq_slow). A dequeue and its helpers transforms a cell's state from a enqueue result state to one of the *final states*:

- $(v, \bot_e, \top_d)$ or $(v, r_e, \top_d)$: it returns v using the fast-path,
- $(v, \bot_e, r_d)$ or (v, r_e, r_d): it returns v using the slow-path, or
- $(\top, r, \bot_d)$ or $(\top, \top_e, \bot_d)$: it returns EMPTY or $\top$.

The difference between an enqueue result state and a final state is that, if the cell's enqueue result state has a value, a cell's deq field is transitioned from $\bot_d$ to $\top_d$ or a dequeue request.

A fast-path dequeue is similar to the dequeue in Listing 1, except that it calls help_enq on cells to try to secure a value, which helps enqueuers if necessary (line 143). If help_enq returns a value, the dequeuer should *claim* the value using $(Q[i].deq : \bot_d \mapsto \top_d)$. If help_enq returns $\top$ or the dequeue failed to claim the returned value, deq_fast outputs the cell's index, which will be used as the id of the dequeue request on the slow-path, and returns $\top$ to indicate failure. If a dequeuer successfully claims a value, it must help its dequeue peer (line 136) before it returns the value.

A slow-path dequeuer begins by publishing a dequeue request to announce its intent. Then both the dequeuer and its helpers call help_deq to complete the pending request. Since the dequeuer calls help_deq to complete its own request, we refer to both the dequeuer and the helping dequeues as *helpers*. help_deq ensures the request is complete before it returns.

A dequeue request $r = (id, pending, idx)$ is initially $(cid, 1, cid)$, where cid is the index of the cell that the dequeue examined last on the fast-path, and $prior$ (a prior candidate cell) is initially cid. To help a dequeue request r, a helper performs the following steps:

1. *look for* a cell $Q[cand]$ that has an unclaimed value or permits returning EMPTY (a candidate);
2. if the helper identified $Q[cand]$ as a candidate, try to *announce* $cand$ to other helpers using $(r\text{->}state : (1, prior) \mapsto (1, cand))$;
3. try to *claim* $Q[s.idx]$ if it has a value using $(Q[s.idx].deq : \bot_d \mapsto r)$, where $s = r\text{->}state$ (an announced candidate);
4. if $Q[s.idx]$ doesn't have a value or someone claimed $Q[s.idx]$ for request r, try to *close* r using $(r\text{->}state : (1, s.idx) \mapsto (0, s.idx))$ and return;
5. $prior := s.idx$. if $cand > s.idx$ begin with step 2; else return to step 1.

Finally, the dequeuer should take the request's result from $Q[s.idx]$.

```
128 void *dequeue(Queue *q, Handle *h)
129   for (p := PATIENCE; p >= 0; p--) {
130     v := deq_fast(q, h, &cell_id);
131     if (v != ⊤) break;
132   }
133   if (v = ⊤) v = deq_slow(q, h, cell_id);
134   // invariant: v is a value or EMPTY
135   if (v != EMPTY) { // got a value, so help peer
136     help_deq(q, h, h->deq.peer);
137     h->deq.peer := h->deq.peer->next; // move to next peer
138   }
139   return v;

140 void *deq_fast(Queue *q, Handle *h, int *id)
141   // obtain cell index and locate candidate cell
142   i := FAA(&q->H, 1); c := find_cell(&h->head, i);
143   v := help_enq(q, h, c, i);
144   if (v = EMPTY) return EMPTY;
145   // the cell has a value and I claimed it
146   if (v != ⊤ and CAS(&c->deq, ⊥_d, ⊤_d) return v;
147   // otherwise fail, returning cell id
148   *id := i; return ⊤;

149 void *deq_slow(Queue *q, Handle *h, int cid)
150   // publish dequeue request
151   r := &h->deq.req; r->id := cid; r->state := (1, cid);
152   help_deq(q, h, h);
153   // find the destination cell & read its value
154   i := r->state.idx; c := find_cell(&h->head, i);
155   v := c->val;
156   advance_end_for_linearizability(&q->H, i+1);
157   return (v = ⊤ ? EMPTY : v);

158 void help_deq(Queue *q, Handle *h, Handle *helpee)
159   // inspect a dequeue request
160   r := helpee->deq.req; s := r->state; id := r->id;
161   // if this request doesn't need help, return
162   if (!s.pending or s.idx < id) return;
163   // ha: a local segment pointer for announced cells
164   ha := helpee->head;
165   s := r->state; // must read after reading helpee->head
166   prior := id; i := id; cand := 0;
167   while (true) {
168     // find a candidate cell, if I don't have one
169     // loop breaks when either find a candidate
170     // or a candidate is announced
171     // hc: a local segment pointer for candidate cells
172     for (hc := ha; !cand and s.idx = prior;) {
173       c := find_cell(&hc, ++i);
174       v := help_enq(q, hc, c, i);
175       // it is a candidate if it help_enq return EMPTY
176       // or a value that is not claimed by dequeues
177       if (v = EMPTY or (v != ⊤ and c->deq = ⊥_d)) cand := i;
178       // inspect request state again
179       else s := r->state;
180     }
181     if (cand) {
182       // found a candidate cell, try to announce it
183       CAS(&r->state, (1,prior), (1,cand));
184       s := r->state;
185     }
186     // invariant: some candidate announced in s.idx
187     // quit if request is complete
188     if (!s.pending or r->id != id) return;
189     // find the announced candidate
190     c := find_cell(&ha, s.idx);
191     // if candidate permits returning EMPTY (c->val = ⊤)
192     // or this helper claimed the value for r with CAS
193     // or another helper claimed the value for r
194     if (c->val = ⊤ or CAS(&c->deq,⊥_d,r) or c->deq = r) {
195       // request is complete, try to clear pending bit
196       CAS(&r->state, s, (0, s.idx));
197       // invariant: r is complete; r->state.pending=0
198       return;
199     }
200     // prepare for next iteration
201     prior := s.idx;
202     // if announced candidate is newer than visited cell
203     // abandon "cand" (if any); bump i
204     if (s.idx ≥ i) { cand := 0; i := s.idx; }
205   }
```

Listing 4. Pseudo code for wait-free dequeue.

Find a candidate cell The first step of helping a dequeue request is to find a candidate cell, which either has a value that is not claimed by any dequeues or some help_enq called on it returned EMPTY. A helper may find a candidate in two ways: it may traverse the queue and perform help_enq on each cell it visits (line 174) until it finds one, or it may read $r\text{->}state.idx$ (line 179) to see if some other helper has announced a candidate cell. It is possible that a helper finds a candidate $Q[i]$ by traversing the queue and noticing that someone has announced another candiate $Q[j](j \neq i)$ when it tries to announce $Q[i]$ and failed (line 183). In this case, if $j > i$, the helper abandons $Q[i]$ and uses $Q[j]$ as its candidate; otherwise, the helper uses $Q[j]$ and keeps $Q[i]$ as a backup in case helpers for this request fail to complete the request using $Q[j]$.[4] In short, we ensure the following invariant in selecting candidates:

Invariant 7. *The announced candidate cell index of a dequeue request r is initially $r.id$ and increases monotonically.*

Ensure linearizability Similar to wait-free enqueue, wait-free dequeue maintains the following invariants for linearizability.

Invariant 8. *A dequeue can only take a value from a cell $Q[i]$ where $i < H$ where the dequeue completes.*

Since a fast-path dequeue obtains cell indices using FAA, this is obviously true. A slow-path dequeue ensures this invariant by ensuring that $H > i$ on line 156 before it returns.

Invariant 9. *A dequeue may return* EMPTY *after some helper saw a cell $Q[i]$ with a $\top$ when $T \leq i$ and announced $Q[i]$.*

The dequeue may return EMPTY if some help_enq on the last announced cell returned EMPTY. When a helper finds an announced candidate $Q[i]$ that does not contain a value, i.e., $Q[i].val = \top$ (line 194), the helper can safely assume the announcer's help_enq performed on $Q[i]$ returned EMPTY which means the announcer saw $T \leq i$. Therefore, it can close the request and the dequeuer can return EMPTY.

Invariant 10. *For a dequeue request r, every cell $Q[i]$: $r.id \leq i \leq r.idx$ has been visited by at least one helper.*

Every helper traverses the queue sequentially starting at $r.id$. help_deq may advance i by more than one only when an announced candidate has an index $s.idx > i$. In this case, the helper that announced $s.idx$ must have visited $s.idx$.

Invariant 11. *If an announced candidate cell satisfies a dequeue request, no new candidate may be announced in the future.*

If the conditions on line 194 evaluate to true for one helper, it will clear the pending bit and return on line 198. All other helpers will return either on line 188 or on line 198 without changing the announced candidate cell.

Ensure wait-freedom For wait-freedom, a dequeue ensures invariants similar to those for wait-free enqueues:

Invariant 12. *A dequeue helper keeps helping a peer until its pending request completes or it has no pending request for help.*

If a help_deq returns at line 188 or line 198, it means some helper has completed the request being helped. A helper notices that (1) the request is no longer pending or a new request has been published (line 188), or (2) conditions that indicate that the request is complete (line 194).

Invariant 13. *A dequeue helper does not help multiple requests from the same peer until it has helped all other peers.*

A dequeuer cannot publish a new request without causing all helpers of its prior request to return. Publishing a new request changes the request's state, which will be seen by helpers at line 179 and 184, and will cause them to return at line 188 because r->id != id. Anyone helping a peer (line 136) will advance to the next peer (line 137) after helping one request. It will not offer help again to the same peer until after it offers help to all other threads in the ring.

Don't advance segment pointers too early When a helper finds a cell $Q[i]$ using find_cell, find_cell will update the segment pointer used to $segment[i/N]$. An announced candidate cell $Q[j]$ may be in a segment before $segment[i/N]$. To avoid irrevocably advancing beyond a segment containing announced candidates, help_deq uses two local segment pointers to traverse the queue: hc to find candidate cells and ha to find announced cells. Both hc and ha start at the $helpee$'s $head$ pointer. A helper does not change the $helpee$'s head pointer; a dequeuer may advance its $head$ pointer when it finds the resulting cell of the request (line 154).

3.6 Memory Reclamation

The only garbage that needs reclamation in our wait-free queue is segments of the queue that are no longer in use. A segment $segment[i]$ is *retired* when both T and H have moved past $i \times N$, and every enqueued value in $segment[i]$ has been dequeued. We designed an custom scheme to reclaim retired segments; it is essentially an *epoch based reclamation* originally proposed by Harris [10] to manage memory of a non-blocking linked list. Listing 5 shows pseudo code for our memory reclamation scheme and how it is applied to our queue. We augment the queue structure with a 64-bit integer I, which records the id of the oldest segment in the queue. We also augment each thread's handle with a *hazard pointer*, $hzdp$, which a thread uses to announce the segment it is currently using. When a thread begins an operation on the queue, it writes the $head$ or $tail$ pointer it is using to find cells in the queue into its $hzdp$; it clears its $hzdp$ when the operation completes. When a thread helps a dequeuer on the slow-path, it also writes the dequeuer's $head$ pointer into its $hzdp$ (line 220) to access cells starting at the id of the dequeue request. Note that help_deq does not read dequeuer's $head$ pointer again after it sets $hzdp$. Instead, help_deq reads the dequeue request's state again at line 165. If the segment at $helpee\text{->}head$ was reclaimed before $hzdp$ is set, the dequeue request read at line 160 must have completed. Before the dequeue completed, it would have updated the state of its pending request. In that case, the comparison of $s.idx = prior$ in line 172 will fail and help_deq will return on line 188 without accessing a (reclaimed) segment at $helpee\text{->}head$.

Cleanup retired segments When a dequeuer completes a dequeue, it invokes the cleanup routine to attempt to reclaim retired segments. To amortize the cost of memory reclamation, we allow the number of retired segments to accumulate up to a pre-defined threshold before initiating cleaning (line 225). Mutual exclusion between cleaning threads avoids the need for synchronization between concurrent cleaners. When a thread starts cleaning, it uses CAS to update I to -1, indicating that cleaning is in progress. When other threads see $I = -1$, they immediately return.

A cleaning thread initially attempts to reclaim every segment $segment[i]$, $i \in [I, head\text{->}id)$, where $head$ is the thread's head pointer. The cleaner uses s and e to denote the segments at the start and end of the queue's segments to reclaim; initially, $s =$

[4] It may be the case when one helper called help_enq(Q[j]), it could determine that the queue was EMPTY. However, after $q\text{->}T$ has been moved by subsequent enqueues, another helper later inspecting cell $Q[j]$ might return $\top$, unable to conclude that the queue was EMPTY. In this case, the first helper may later announce j as a candidate after the second helper has moved beyond j.

```
206 struct Queue { ...,  int I : 64; };
207 struct Handle { ..., Segment * hzdp; };

208 void enqueue(Queue *q, Handle *h, void *v)
209   h->hzdp := h->tail;
210   ...
211   h->hzdp := null;

212 void * dequeue(Queue *q, Handle *h)
213   h->hzdp := h->head;
214   ...
215   h->hzdp := null;
216   cleanup(q, h);
217   return v;

218 void help_deq(Queue *q, Handle *h, Handle *peer)
219   ...
220   h->hzdp := ha; // between line 164 and 165
221   ...

222 void cleanup(Queue *q, Handle *h)
223   i := q->I; e := h->head;
224   if (i = -1) return;
225   if (e->id - i < MAX_GARBAGE) return;
226   if (!CAS(&q->I, i, -1)) return;
227   s := q->Q; hds := []; j := 0;
228   for (p := h->next; p != h and e->id > i; p := p->next) {
229     verify(&e, p->hzdp);
230     update(&p->head, &e, p);
231     update(&p->tail, &e, p);
232     hds[j++] := p;
233   }
234   // reverse traversal
235   while (e->id > i and j > 0) verify(&e, hds[--j]->hzdp);
236   if (e->id <= i) { q->Q := s; return; }
237   q->Q := e; q->I := e->id;
238   free_list(s, e);

239 void update(Segment **from, Segment **to, Handle *h)
240   n := *from
241   if (n->id < (*to)->id) {
242     if (!CAS(from, n, *to)) {
243       n := *from;
244       if (n->id < (*to)->id) *to := n;
245     }
246     verify(to, h->hzdp);
247   }

248 void verify(Segment **seg, Segment *hzdp)
249   if (hzdp and hzdp->id < seg->id) *seg := hzdp;
```

Listing 5. Pseudo code for memory reclamation

$segment[I]$ and $e = head$. Since a segment between s and e may be still in use, a cleaner needs to inspect every other thread's state to avoid reclaiming such segments. If it finds such a segment $segment[j] : s\text{->}id \leq j < e\text{->}id$, it updates e to j. The cleaning thread returns if $e\text{->}id \leq s\text{->}id$ since there is nothing to reclaim.

Update head and tail pointers When a cleaning thread scans each other thread's state, it also updates a thread's head and tail pointers if the cleaning thread wants to reclaim them and they are not in use. This is to avoid having a thread blocking the garbage collection if it does not perform an operation on the queue for a long period. Here we also use Dijkstra's protocol [5] to synchronize between the cleaning and the visited thread. A cleaning thread attempting to update a thread's head or tail pointer p from $segment[i]$ to $segment[j]$, $j > i$, can reclaim $segment[i]$ if (1) the thread's $hzdp$ is `null` or $hzdp\text{->}id \geq j$ (line 229), (2) ($p : segment[i] \mapsto segment[j]$) succeeds or the updated pointer p' which caused the CAS to fail still has $p = segment[j']$, $j' > i$ (line 242), and (3) on line 246, the thread's $hzdp$ is still `null` or $hzdp\text{->}id \geq j$ after the CAS .

Visit threads in reverse order Normally, a thread's hazard pointer is only updated to newer segments, either by the thread or a cleaning thread. In particular, given a hazard pointer that points to $segment[i]$, a new non-null value of the hazard pointer is normally $segment[j]$, $j > i$. However, when a thread is helping a dequeue peer, the thread sets its hazard pointer to the peer's head pointer, which may be a segment that is older than the thread's head pointer. To avoid this "backward jump" causing problem for memory reclamation, we add a backward traversal step to a cleaning thread after it has visited each thread (line 235). The forward traversal updates each thread's $head$ and $tail$ pointers to segments that are newer than e. Therefore, no thread may "jump backward" to a segment $segment[k]$, where $k < e\text{->}id$ after a forward traversal. The backward traversal visits every thread in reverse order; it will capture any hazard pointer's "backward jump" that happened *during* the forward traversal.

Overhead When implementing this memory reclamation scheme on an system that provides a memory model that is weaker than sequential consistency, a memory fence is needed after a thread sets its hazard pointer. On x86 architectures, since atomic primitives flush store buffers, no extra memory fence is needed. Because our queue performs a FAA after setting the hazard pointer using a head or tail pointer, we do not need an extra fence in `enqueue` and `dequeue`. We only add a memory fence after setting the hazard pointer in `help_deq`. Therefore on x86 systems, our memory reclamation scheme adds almost no overhead to the fast-path execution, which is unprecedented among memory reclamation schemes for lock-free data structures. On Power architectures, which are weakly ordered, our reclamation protocol requires a full memory fence (`sync`) after setting a hazard pointer and a lightweight fence (`lwsync`) before a hazard pointer is cleared; this is the same as other epoch-based memory reclamation strategies.

Thread failure Our memory reclamation scheme is prone to thread failure; if a thread fails or suspended infinitely during an operation, it may cause unbounded memory leakage. The failure of an individual thread is impossible when using Posix threads since the whole process will fail if a Posix thread fails. If threads are individual processes that communicates via shared memory, thread failure is a possibility. We consider solving this problem as part of our future work. Recently, Brown [3] developed a distributed variant of epoch based reclamation, called DEBRA, that uses *signaling* to deal with thread failures. If thread failure is possible, one can apply DEBRA to reclaim memory in our wait-free queue.

4. Linearizability and Wait-freedom

In this section, we reason about the linearizability of our queue and its wait freedom.

4.1 Linearizability

We reason about linearizability using a logical queue L that contains only cells into which values were deposited, i.e., excluding cells marked unusable by a dequeuer. We denote the head and tail indices of L as H_L and T_L. We denote the cell at index k in L as $L[k], k \in \{1, 2, ...\}$. A monotonic function f maps a cell index k in L to the index of the corresponding cell in Q, i.e., $L[k] = Q[f(k)]$. We denote an enqueue that enqueues into $L[k]$ as E_k and a dequeue that dequeues from $L[k]$ as D_k. We use $\bar{D}$ to denote a dequeue that returns `EMPTY`.

Let $W = \{e_j : j = 1, 2, ...\}$ be a possibly infinite execution history of our wait-free queue. We assume that if an operation begins in W it also completes in W. Procedure P, shown in Figure 1, assigns a linearization point to each operation in W.

Essentially, procedure P linearizes each enqueue E_k at the FAA or CAS that moved T past $f(k)$. The FAA or CAS may be

```
250 H_L := 0; T_L := 0
251 for j := {1, 2, ...}
252   if e_j is a FAA or a CAS that updates T to t then
253     while true
254       k := T_L + 1
255       if f(k) > t then break
256       linearize E_k at e_j
257       T_L := k
258   if e_j is a read of T performed on behalf of some D̄ where
259        T ≤ i and Q[i] is the cell D̄ is visiting then
260      linearize D̄ at e_j
261   while H_L < T_L
262     k := H_L + 1
263     if f(k) > H then break
264     linearize D_k at e_j
265     H_L := k
```

Figure 1. Linearization procedure P. Operations linearized at the same e_j are ordered based on the order of P's steps.

performed by the enqueuer or its helper but it must happen during the execution interval of E_k. If E_k enqueues into $L[k]$ via the fast-path, it obtains the index $f(k)$ using FAA, which is its linearization point. If E_k enqueues into $L[k]$ via the slow-path, it publishes an enqueue request r with an id $r.id$ which is obtained using FAA on T. Thus, we know $T = r.id$ before the FAA on T returns $r.id$. Because Invariant 6 ensures $r.id < i$, where i represents $f(k)$, we have $T = r.id < f(k)$ before the FAA that returns $r.id$. In addition, Invariant 4 ensures $i < T$, where $i = f(k)$, when E_k's enqueue request completes. Since T increments monotonically, $T < f(k)$ before the FAA on T returns $r.id$, and $T > f(k)$ after the enqueue request completes, the FAA or CAS that moves T past $f(k)$ must happen during the execution interval of E_k.

Procedure P linearizes each dequeue D_k at the earliest point that (1) its matching enqueue E_k is linearized and (2) a FAA or CAS has moved H past $f(k)$. Such a linearization point is always within the execution interval of D_k. Because D_k takes a value from $L[k]$ using `help_enq` on $Q[f(k)]$, which ensures $f(k) < T$ before it exits.[5] This means E_k is linearized before D_k's `help_enq` on $Q[f(k)]$ returns. If D_k dequeues from $L[k]$ via the fast-path, it obtains $f(k)$ using its own FAA on H, which is trivially within its execution interval. If D_k dequeues from $L[k]$ via the slow-path, it publishes dequeue request r with an id $r.id$ obtained using FAA on H by its last failing fast-path dequeue. We know that $H = r.id$ before its FAA returned $r.id$. Because the helpers of a dequeue request only search for candidate cells starting at $r.id$, we know $r.id < f(k)$. Since $H < f(k)$ before the FAA returning the request id that D_k stores in $r.id$, H increments monotonically, and D_k ensures $H > f(k)$ before it returns, we know the FAA or CAS that moves H past $f(k)$ must happen during the execution interval of D_k. Since E_k happens before or during D_k's execution and the FAA or CAS that moves H past $f(k)$ happens during D_k's exection, D_k's linearization point is always within its execution interval.

Procedure P linearizes a dequeue $\bar{D}$ that returns `EMPTY` at its read of T, where $T \leq i$, $Q[i]$ is the cell $\bar{D}$ or its helper is visiting. Since $\bar{D}$'s helper only helps when $\bar{D}$'s dequeue request is pending. $\bar{D}$'s linearization point is always within its execution interval.

We now prove that P's linearization of W conforms to a correct sequential execution history of a FIFO queue. We denote the linearization point of an operation op as $e_{j(op)}$. We denote the precedence ordering between two operations as $op_1 \prec op_2$. If $op_1 \prec op_2$, we have $j(op_1) \leq j(op_2)$. Note that operations linearized at the same e_j are ordered based on the order of P's steps.

[5] $f(k)$ is known as i in `help_enq`.

Lemma 4.1. *For $k \in \{1, 2, ...\}$, P's linearization of W satisfies: (1) $E_k \prec E_{k+1}$, (2) $E_k \prec D_k$, (3) $D_k \prec D_{k+1}$, and (4) $H_L = T_L$ at every $e_{j(\bar{D})}$.*

Proof. Condition (1) means that a later enqueue E_{k+1} cannot be linearized before an earlier enqueue E_k. This is obviously true since P linearizes E_k at the point T moves past $f(k)$ and T increments monotonically.

Condition (2) means that a dequeue is always linearized after its matching enqueue. This is obviously true since P linearizes D_k after E_k (line 261).

We show that condition (3) holds using contradiction. Suppose $j(D_{k+1}) < j(D_k)$, which means a later dequeue D_{k+1} is linearized before an earlier dequeue D_k. Since we linearize D_k at the earliest point after $e_{j(E_k)}$ and $H > f(k)$, we have $H > f(k+1)$ at $e_{j(D_{k+1})}$. Since $j(D_k) > j(D_{k+1}) \geq j(E_{k+1}) \geq j(E_k)$, we must have $H \leq f(k)$ at $e_{j(D_{k+1})}$, otherwise D_k would have been linearized at $e_{j(D_{k+1})}$ on line 264. $H > f(k+1)$ and $H \leq f(k)$ cannot both be true simultaneously at $e_{j(D_{k+1})}$ since f is monotonic: a contradiction.

We show that condition (4) holds using contradiction. Suppose $H_L < T_L = t$ at $e_{j(\bar{D})}$. Then there exists a minimal h such that $h \leq H_L$ and D_h is not linearized at $e_{j(\bar{D})}$. This means we have an enqueued value in L that has not been dequeued when a dequeue returns `EMPTY`. We now prove that such a case is not possible. Assume $e_{j(\bar{D})}$ (or its helper) reads T and sees $T < i$ ①, where $Q[i]$ is the cell inspected by a `help_enq` being performed on behalf of $\bar{D}$. We know E_t is linearized when $T_L = t$, hence $f(t) \leq T$ ②. We also know $H < f(h)$ ③ at $e_{j(\bar{D})}$, otherwise P would have linearized D_h at $e_{j(\bar{D})}$ on line 264. Because $h \leq H_L < T_L = t$ and f is monotonic, $f(h) < f(t)$ ④. Combining ①, ②, ③, and ④, we have $H < i$ at $e_{j(\bar{D})}$ ⑤. If $\bar{D}$ visited $Q[i]$ using the fast-path, $\bar{D}$ would have performed a FAA that updated H to $i+1$; however, from ⑤ $H < i$—a contradiction. Consider the case $\bar{D}$ (or its helpers) visited $Q[i]$ using the slow-path. On its slow-path, a dequeue always uses the value x returned from its last failed fast-path attempt on H as its request id; following the FAA, we have $H = x + 1$. There are two cases we must consider for $\bar{D}$: $x \geq f(h)$ and $x < f(h)$; we consider each in turn. If $x \geq f(h)$ and $H = x + 1$, then $H > f(h)$; however, from ③, $H < f(h)$—a contradiction. If $x < f(h)$, $\bar{D}$ (or its helper) must have visited $Q[f(h)]$, i.e., $L[h]$, before visiting $Q[i]$ (Invariant 10). Then $\bar{D}$ (or its helper) must either claimed a value from $L[h]$ or marked it as unusable. If $\bar{D}$ (or its helper) claimed a value from $L[h]$, $\bar{D}$ does not return `EMPTY`, a contradiction. If it marked $L[h]$ unusable, D_h may not dequeue from $L[h]$ later, a contradiction. □

Theorem 4.2. *The queue presented in Listings 3- 5 is linearizable.*

Proof. Lemma 4.1 shows that procedure P linearizes an arbitrary execution history W of our wait-free queue to a correct sequential execution of a FIFO queue, thus our queue is linearizable. □

4.2 Wait Freedom

We prove that operations on our queue are wait-free.

Lemma 4.3. *Each enqueue operation in Listing 3 completes in a bounded number of steps.*

Proof Sketch. An enqueue e will attempt at most `PATIENCE` times on the fast-path. On the slow-path, an enqueue request r will definitely complete when all other threads become its helpers. In the worst case, other threads may help $n-1$ operations by other threads before helping r, where n is the total number of threads. Every time an enqueuer fails to enqueue into a cell $Q[i]$ on the

slow-path, its contender c, the dequeuer that marks $Q[i]$ unusable, will help its peer. If c's peer does not need help or it completes its peer's pending request, it sets its peer to the next thread. If c fails to help its peer, it is because $Q[i].enq$ was updated to $\top$ or another request r' by another helper h. In both cases, h will update its peer to the next thread. Therefore, every time the enqueuer fails, some thread will update its peer to the next thread. After an enqueue fails $(n-1)^2$ times on the slow-path, all other threads will become its helper and it will complete. □

Lemma 4.4. *Each dequeue operation in Listing 4 completes in a bounded number of steps.*

Proof Sketch. Similar to enqueues, a dequeue will certainly complete when all other threads becomes its helper on the slow-path. In the worst case, all other threads may help $n-1$ operations by other threads before it helps a slow-path dequeue d. Every time a dequeuer successfully dequeues a value, it helps its peer and updates its peer to the next thread. All dequeuers will help d after d fails to claim $(n-1)^2$ values. Since an enqueue will succeed after it tries $(n-1)^2$ cells in the worst case, a dequeuer may traverse $(n-1)^2$ cells to find a value. Therefore, a slow-path dequeue will definitely complete after visiting $(n-1)^4$ cells. □

Lemma 4.5. *The memory reclamation scheme in Listing 5 completes in a bounded number of steps.*

Proof Sketch. The cleanup process only visits handles of p threads, where p is a finite number, to identify the lowest numbered `Segment` in use. The number of segments to be freed between the front and the first segment in use is finite. Therefore, an invocation of cleanup will complete in a bounded number of steps.

Dequeue augmented with memory reclamation makes only one call to cleanup. Since the work of cleanup is bounded and other work of the dequeue is bounded by Lemma 4.4, the work of the augmented dequeue is also bounded and thus wait free. □

Theorem 4.6. *The queue implementation presented in Listings 3- 5 is wait-free.*

Proof. Lemmas 4.3, 4.4 show that enqueue and dequeue are wait-free without memory reclamation. Lemma 4.5 shows that after adding memory reclamation to dequeue it is still wait free. Therefore, the queue implementation is wait-free since all of its operations are wait-free. □

5. Evaluation

To understand the performance characteristics of the fast and slow paths in our wait-free queue operations, we study two versions of our code that differ in the `PATIENCE` threshold we set. We set `PATIENCE` to 10 in the first version. We view this as a practical threshold that should allow most operations to complete on the fast-path, maximizing performance. We set `PATIENCE` to 0 in the second version, which means that each operation only attempts the fast-path once before switching to the slow-path. This version emphasizes the performance of the slow-path in our queue, which provides a lower bound on the queue's throughput on most architectures. We refer to these two versions of our wait-free queue as *WF-10* and *WF-0*.

We compare our wait-free queue to several representative queue implementations in the literature. We use Morrison and Afek's LCRQ [19] as the representative of prior FAA-based queues; it delivers the best performance among prior concurrent queue implementations because it avoids the CAS retry problem by using FAA. We chose CC-Queue by Fatourou et al. [8] as the representative of queues based on the *combining* principle. We also compare with Michael and Scott's classic non-blocking MS-Queue [17]. Note that LCRQ and MS-Queue are lock-free queues, and CC-Queue is a blocking queue; none provides a progress guarantee as strong as our wait-free queue. Previous wait-free queues sacrifice performance to provide strong progress guarantee and hence deliver lower performance than these queues. In particular, the wait-free queue constructed by Kogan and Petrank [13] using the fast-path-slow-path strategy, delivers the same performance as MS-Queue which is the fast-path in their implementaion. The wait-free queue by Fatourou and Kallimanis [7] using the P-Sim universal construction is slower than CC-Queue, which is also of their design.

We also include a microbenchmark that simulates enqueue and dequeue operations with FAA primitives on two shared variables: one for enqueues and the other for dequeues. This simple microbenchmark provides a practical upper bound for the throughput of all queue implementations based on FAA.

5.1 Experiment Setup

Platforms We evaluated the performance of aforementioned queue implementation and the FAA microbenchmark on four platforms based on the following processors: Intel Haswell, Intel Knight's Corner Xeon Phi, AMD Magny-Cours, and IBM Power7. Table 1 lists the key characteristics of these platforms. Since the Power7 architecture does not directly implement FAA, we emulate FAA using a retry loop that employs `load-linked` and `store-conditional` (LL/SC). It is important to note that implementing FAA with LL/SC sacrifices the wait-freedom of our queue because of the potential for unbounded retries.

We compiled each code with `GCC 4.9.2` on all platforms except on our Intel Knight's Corner Xeon Phi where we used `GCC 4.7.0`, which was the only GNU compiler available. We used `-O3` as our optimization level without any special optimization flags.

In our experiments, we used a *compact* mapping of software to hardware threads; namely, each software thread is mapped to the hardware thread that is closest to previously mapped threads. For example, on a dual-socket system with a pair of 18-core Intel Haswell processors, where each core supports two threads using simultaneous multithreading (SMT), we map the first two of the 72 threads to the two SMT threads on the first core, the next two threads to the second core on the same processor, and the last 36 threads to the second processor.

Implementation We implemented our queue algorithm, CC-Queue, and MS-Queue in C. Our implementations of CC-Queue and MS-Queue employ fences as necessary to execute correctly on the Power7, which is a weakly ordered architecture. We use an LCRQ implementation provided to us by Morrison [20]. We evaluate these algorithms using a framework very similar to that of Fatourou and Kallimani. Unlike prior work, which didn't reclaim any memory, assuming that a 3rd party garbage collector would handle the matter, we consider memory reclamation to be an integral responsibility of the queue algorithms since garbage collection is not available in all environments, including Fatourou and Kallimani's framework previously used to test the performance of queue algorithms. For LCRQ and MS-Queue, and CC-Queue, which lacked support for memory reclamation, we added it.

To LCRQ and MS-Queue, we added implementations of the *hazard pointer* scheme [16] to reclaim memory. Since CC-Queue is a blocking queue, memory can be safely reclaimed after each operation without the use of a lock-free memory reclamation scheme. The segment size in our queue, N, we used in our experiments is 2^{10}. The segment size in LCRQ is 2^{12}, which is the size that yields the best performance when running with all cores.

Processor Model	Clock Speed	# of Processors	# of Cores	# of Threads	CC Protocol	Native FAA
Intel Xeon E5-2699v3 (Haswell)	2.30 GHz	2	36	72	snooping	yes
Intel Xeon Phi 3120	1.10 GHz	1	57	228	directory	yes
AMD Opteron 6168 (Magny-Cours)	0.80 GHz	4	48	48	directory	yes
IBM Power7 8233-E8B	3.55 GHz	4	32	128	snooping	no

Table 1. Summary of experimental platforms.

Benchmark All experiments employ an initially-empty queue to which threads apply a series of enqueue and dequeue operations. We use the following two benchmarks to evaluation the performance of the queue algorithms.

- enqueue-dequeue pairs: in each iteration a thread executes an enqueue followed by a dequeue; the benchmark executes 10^7 pairs partitioned evenly among all threads;
- 50% enqueues: in each iteration a thread decides uniformly at random to execute an enqueue or dequeue with equal odds; the benchmark executes 10^7 operations partitioned evenly among all threads;

As in prior studies of queue performance, each thread performs a random amount of "work" (between 50 and $100ns$) between operations to avoid artifical *long run scenarios* [17], in which a thread completes a series of operations when it has the shared object in its L1 cache without being interrupted by other threads. Long run scenarios can over-optimisticly bias the evaluation result due to an unrealistically low cache miss rate. The execution time of this "work" is excluded from the performance numbers we report in this paper. Surprisingly, this random "work" between operations actually *improves* performance on the Haswell-based system used for our experiments. We have been unable to obtain enough information about Haswell's coherence protocol and how it performs arbitration under contention to explain this effect.

During tests, we keep the activities of other processes on the system to a minimum to avoid interference. All software threads are pinned to a different hardware thread to minimize interference by the OS scheduler. Our benchmark uses the `jemalloc` [6] memory allocator to avoid requesting memory pages from the OS on every allocation.

Methodology To provide an experimental result that is statistically rigorous, we follow the methodology suggested by Georges et al. [9]. In each invocation of a process, we perform at most 20 benchmark iterations. During each process invocation i, we determine the iteration s_i in which a steady-state performance is reached, i.e., once the *coefficient of variation* (COV) of the most recent 5 iterations ($s_i - 4$, $s_i - 3$, ... s_i) falls below the threshold 0.02. If the COV never drops below 0.02 for any 5 consecutive iterations in the 20 iterations, we choose the 5 consecutive iterations that have the lowest COV under steady-state. We compute the mean $\bar{x}_i$ of the 5 benchmark iterations under steady-state

$$\bar{x}_i = \sum_{j=s_i-4}^{s_i} x_{ij}.$$

Then we perform 10 invocations of the same binary and compute the 95% *confidence interval* over the mean of the 10 invocations $\bar{x}_i (i = 1, 2, ...10)$. Since the number of measurements n is relatively small ($n = 10$ in our case), we compute the confidence interval under the assumption that the distribution of the transformed value

$$t = \frac{\bar{x} - \mu}{s/\sqrt{n}}$$

where μ is the population mean and s is the sample standard deviation, follows *Student's t*-distribution with $n - 1$ degrees of freedom. Georges et al. [9] describe in detail how to compute confidence intervals.

5.2 Experimental Results

Figure 2 shows results of experiments with the queue implementations and the FAA-based microbenchmark on all four platforms. For the 50%-enqueue benchmark, we omit the results on Intel Knight's Corner Xeon Phi and IBM Power7 since they are similar to those of the enqueue-dequeue pair benchmark. We describe the experimental result in two parts, considering the the performance of a *single core* and a *multi-core* execution respectively.

Single core performance The single core performance of algorithms shown in Figure 2 and marked with an $*$, uses one thread on Magny-Cours, two threads on Haswell, and four threads on Knight's Corner Xeon Phi and Power7.

The FAA microbenchmark outperforms all queue implementations because it contains only a FAA for each operation. Moreover, because all threads on the same core share the same L1 cache, the variable updated by FAA is always in cache.

CC-Queue outperforms other queues in sequential executions. Because CC-Queue reuses the same node for every enqueue and dequeue pair, it does not incur any cache misses without a contending thread. However, this advantage disappears with two or more threads.

For the enqueue-dequeue pairs benchmark, WF-10 outperforms LCRQ by about 65% in a single thread execution on both Haswell and Magny-Cours. For the 50%-enqueues benchmark, WF-10 outperforms LCRQ by about 35% in a single thread execution on both Haswell and Magny-Cours. This is because of the low overhead of our memory reclamation scheme compares favorably with the hazard pointer scheme we added to LCRQ. LCRQ executes at least one memory fence in each operation.

Multi-core performance Using 228 threads of a Intel Knight's Corner Xeon Phi, the performance of the fast version of our wait-free queue, WF-10, achieves about $150\times$ the performance of MS-Queue and about $30\times$ the performance of CC-Queue. MS-Queue performance suffers from the CAS retry problem described earlier, and CC-Queue's combining technique serializes the execution of operations that could have been performed in parallel.

Although the FAA implementation used in our experiment on IBM Power7 suffers from retries of LL/SC pairs, our fast version still achieves about $3.3\times$ the performance of MS-Queue and about $2.8\times$ the performance of CC-Queue. This shows that our queue is still faster than other queue implementaions even on systems that do not support FAA in hardware, if one is willing to sacrifice the wait-free property.

For the enqueue-dequeue pairs benchmark, WF-10 and LCRQ performs almost the same on both Haswell and Magny-Cours machine. On Intel Haswell, AMD Magny-Cours, and IBM Power7, WF-10 and LCRQ deliver throughput comparable to the FAA microbenchmark, which provides a practical bound on the throughput for all FAA-based queues. Surprisingly, WF-0, which executes the slow-path more frequently performs better than WF-10 in some cases. Since the slow-path dequeue in our algorithm does not perform FAA, it relieves contention on the head and tail indices, which improves performance.

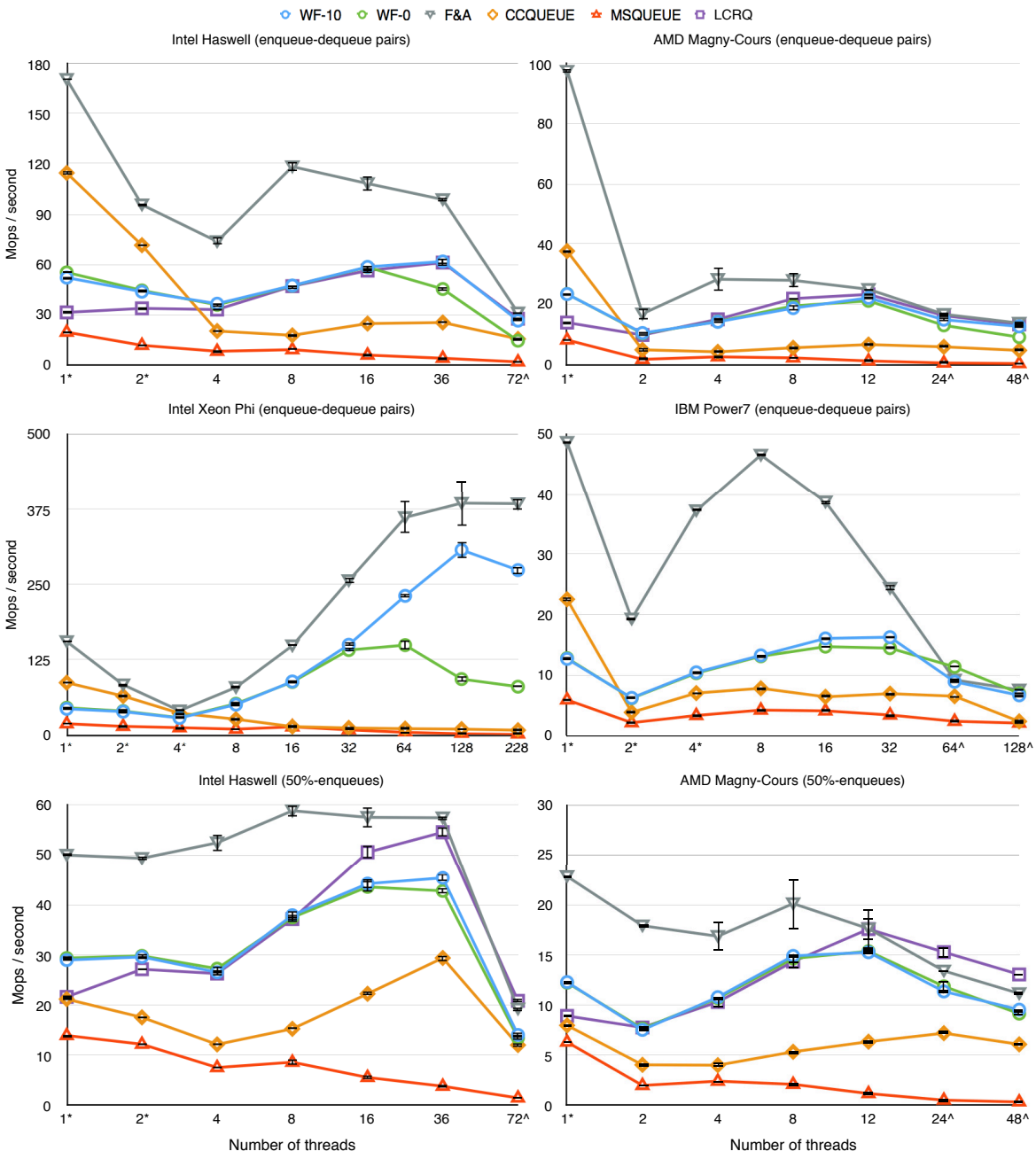

Figure 2. Throughput of different queues on all four platforms. There is no implementation of LCRQ for Intel Xeon Phi and IBM Power7 because the architectures lack support for CAS2. *: executions where all threads run on the same core. ^: executions that involve multiple processors. The error bar around each data point marks the bounds for its 95% confidence interval.

At high concurrency levels, LCRQ outperforms our wait-free queues in the 50%-enqueues benchmark by about 50% on Intel Haswell and 35% on AMD Magny-Cours; this is because our wait-free queue is more costly when trying to dequeue from an empty queue. To return an empty value, a dequeue needs to inspect its enqueue peer first which elevates traffic between cores. Moreover, when a dequeue cannot return empty because enqueues have performed FAA but have yet to put values in the queue, the dequeue will consume more cells before it completes.

# of threads	36	72	144	288
% of slow-path enqueues	0.002	0.004	0.024	0.028
% of slow-path dequeues	1.536	4.047	2.888	3.239
% of empty dequeues	0.002	0.002	0.003	0.003

Table 2. Breakdown of different execution passes of *WF-0* on Intel Haswell. The 144 and 288 threads are oversubscribed workloads.

Breakdown of different execution paths Table 2 shows a breakdown of different execution paths on our Intel Haswell system for

the 50%-enqueues benchmark. Using more software threads than the 72 hardware threads increases the percentage of enqueues completed on the slow-path but decreases the percentage of dequeues completed on the slow-path. In all cases, the percentage of slow-path enqueues and dequeues is very low; more than 99% of enqueues succeed with a single fast-path attempt and more than 95% dequeues succeed with a single fast-path attempt. Our wait-free queue rarely returns `EMPTY`.

In all cases, the percentage of dequeues that return `EMPTY` is less than 0.01%. If a dequeue in our wait-free queue obtains an empty cell, it performs many steps before it reads T. It is very likely that T has been increased to be larger the index of the cell a dequeuer visits when the dequeuer reads T, causing many dequeues that could have returned `EMPTY` to fail.

6. Conclusions

Our design for a *linearizable* and *wait-free* FIFO queue delivers high performance over a range of highly-threaded multi-core systems. On a Power7, our queue cannot provide a wait-free progress guarantee because `fetch-and-add` is implemented using LL/SC, which might require unbounded retries. Nevertheless, at the top thread count on all but a Knight's Corner Xeon Phi coprocessor, the throughput of our queue in operations per second equals or exceeds the throughput of `fetch-and-add` operations per second performed by a tight loop in a microbenchmark. Although the performance of our wait-free queue does not equal the throughput of `fetch-and-add` on the Xeon Phi, it dominates the throughput of other queues by a margin almost twice as wide as the gap between our queue's throughput and that of the `fetch-and-add` microbenchmark. Not only does our queue dominate the throughput of others, it also provides a wait-free progress guarantee that the other queues do not. On a Knight's Corner Xeon Phi, we believe that the gap between the throughput of `fetch-and-add` and our queue is due to low per-thread performance; investigating this is the subject of ongoing work.

In summary, our performance experiments show that our wait-free queue design is fast and faster than queues with weaker non-blocking progress guarantees. Our queue design provides an instance of a wait-free data structure fast enough to use in practice.

Acknowledgments

We thank our shepherd Adam Morrison for proofreading the paper and giving helpful comments. This work was supported in part by the DOE Office of Science cooperative agreement DE-SC0008883. Experiments were performed on systems that include a dual-socket Intel Haswell system on loan from Intel and an IBM Power 755 acquired with support from NIH award NCRR S10RR02950 in addition to an IBM Shared University Research (SUR) Award in partnership with Cisco, Qlogic and Adaptive Computing.

References

[1] J. Alemany and E. W. Felten. Performance Issues in Non-blocking Synchronization on Shared-memory Multiprocessors. In *Proceedings of the Eleventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '92, pages 125–134, New York, NY, USA, 1992. ACM. doi: 10.1145/135419.135446.

[2] J. Anderson and M. Moir. Universal constructions for large objects. In J.-M. Hlary and M. Raynal, editors, *Distributed Algorithms*, volume 972 of *Lecture Notes in Computer Science*, pages 168–182. Springer Berlin Heidelberg, 1995. doi: 10.1007/BFb0022146.

[3] T. A. Brown. Reclaiming Memory for Lock-Free Data Structures: There Has to Be a Better Way. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, PODC '15, pages 261–270, New York, NY, USA, 2015. ACM. doi: 10.1145/2767386.2767436.

[4] P. Chuong, F. Ellen, and V. Ramachandran. A Universal Construction for Wait-free Transaction Friendly Data Structures. In *Proceedings of the 22nd Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 335–344, New York, NY, USA, 2010. ACM. doi: 10.1145/1810479.1810538.

[5] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569–, Sept. 1965. doi: 10.1145/365559.365617.

[6] J. Evans. Scalable memory allocation using `jemalloc`. `http://on.fb.me/1KmCoyj`, January 2011.

[7] P. Fatourou and N. D. Kallimanis. A Highly-efficient Wait-free Universal Construction. In *Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 325–334, New York, NY, USA, 2011. ACM. doi: 10.1145/1989493.1989549.

[8] P. Fatourou and N. D. Kallimanis. Revisiting the Combining Synchronization Technique. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 257–266, New York, NY, USA, 2012. ACM. doi: 10.1145/2145816.2145849.

[9] A. Georges, D. Buytaert, and L. Eeckhout. Statistically Rigorous Java Performance Evaluation. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 57–76, New York, NY, USA, 2007. ACM. doi: 10.1145/1297027.1297033.

[10] T. L. Harris. A Pragmatic Implementation of Non-blocking Linked-Lists. In *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01, pages 300–314, London, UK, UK, 2001. Springer-Verlag.

[11] M. Herlihy. Wait-free Synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, Jan. 1991. doi: 10.1145/114005.102808.

[12] M. P. Herlihy and J. M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990. doi: 10.1145/78969.78972.

[13] A. Kogan and E. Petrank. Wait-free Queues with Multiple Enqueuers and Dequeuers. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 223–234, New York, NY, USA, 2011. ACM. doi: 10.1145/1941553.1941585.

[14] A. Kogan and E. Petrank. A Methodology for Creating Fast Wait-free Data Structures. *SIGPLAN Not.*, 47(8):141–150, Feb. 2012. doi: 10.1145/2370036.2145835.

[15] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.*, 28(9):690–691, Sept. 1979. doi: 10.1109/TC.1979.1675439.

[16] M. M. Michael. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, June 2004. doi: 10.1109/TPDS.2004.8.

[17] M. M. Michael and M. L. Scott. Simple, Fast, and Practical Non-blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, pages 267–275, New York, NY, USA, 1996. ACM. doi: 10.1145/248052.248106.

[18] M. Moir. Laziness pays! Using lazy synchronization mechanisms to improve non-blocking constructions. *Distributed Computing*, 14(4):193–204, 2001. doi: 10.1007/s004460100063.

[19] A. Morrison and Y. Afek. Fast Concurrent Queues for x86 Processors. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 103–112, New York, NY, USA, 2013. ACM. doi: 10.1145/2442516.2442527.

[20] TAU Multicore Computing Group. Fast concurrent queues for x86 processors. `http://mcg.cs.tau.ac.il/projects/lcrq/`.

[21] S. Timnat and E. Petrank. A Practical Wait-free Simulation for Lock-free Data Structures. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 357–368, New York, NY, USA, 2014. ACM. doi: 10.1145/2555243.2555261.