wCQ: A Fast Wait-Free Queue with Bounded Memory Usage

Ruslan Nikolaev
rnikola@psu.edu
The Pennsylvania State University
University Park, PA, USA

Binoy Ravindran
binoy@vt.edu
Virginia Tech
Blacksburg, VA, USA

ABSTRACT

The concurrency literature presents a number of approaches for building non-blocking, FIFO, multiple-producer and multiple-consumer (MPMC) queues. However, only a fraction of them have high performance. In addition, many queue designs, such as LCRQ, trade memory usage for better performance. The recently proposed SCQ design achieves both memory efficiency as well as excellent performance. Unfortunately, both LCRQ and SCQ are only lock-free. On the other hand, existing wait-free queues are either not very performant or suffer from potentially unbounded memory usage. Strictly described, the latter queues, such as Yang & Mellor-Crummey's (YMC) queue, forfeit wait-freedom as they are blocking when memory is exhausted.

We present a wait-free queue, called wCQ. wCQ is based on SCQ and uses its own variation of fast-path-slow-path methodology to attain wait-freedom and bound memory usage. Our experimental studies on x86 and PowerPC architectures validate wCQ's great performance and memory efficiency. They also show that wCQ's performance is often on par with the best known concurrent queue designs.

CCS CONCEPTS

• **Theory of computation → Concurrent algorithms**.

KEYWORDS

wait-free; FIFO queue; ring buffer

ACM Reference Format:
Ruslan Nikolaev and Binoy Ravindran. 2022. wCQ: A Fast Wait-Free Queue with Bounded Memory Usage. In *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '22), July 11–14, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3490148.3538572

1 INTRODUCTION

The concurrency literature presents an array of efficient non-blocking data structures with various types of progress properties. *Lock-free* data structures, where *some* thread must complete an operation after a finite number of steps, have traditionally received substantial practical attention. *Wait-free* data structures, which provide even stronger progress properties by requiring that *all* threads complete any operation after a finite number of steps, have been less popular since they were harder to design and much slower than their lock-free alternatives. Nonetheless, the design methodologies have evolved over the years, and wait-free data structures have increasingly gained more attention because of their strongest progress property.

Wait-freedom is attractive for a number of reasons: (1) lack of starvation and reduced tail latency; (2) security; (3) reliability. Even if we assume that the scheduler is non-adversarial, applications can still be buggy or malicious. When having shared memory between two entities or applications, aside from parameter verification, bounding the number of operations in loops is desirable for security (DoS) and reliability reasons. Note that no theoretical bound exists for lock-free algorithms even if they generally have similar performance.

Creating efficient FIFO queues, let alone wait-free ones, is notoriously hard. Elimination techniques, so familiar for LIFO stacks [12], are not that useful in the FIFO world. Although [29] does describe FIFO elimination, it only works well for certain shorter queues. Additionally, it is not always possible to alter the FIFO discipline, as proposed in [16, 17].

FIFO queues are widely used in a variety of practical applications, which range from ordinary memory pools to programming languages and high-speed networking. Furthermore, for building efficient user-space message passing and scheduling, non-blocking algorithms are especially desirable since they avoid mutexes and kernel-mode switches.

FIFO queues are instrumental for certain synchronization primitives. A number of languages, e.g., Vlang, Go, can benefit from having a fast queue for their concurrency and synchronization constructs. For example, Go needs a queue for its buffered channel implementation. Likewise, high-speed networking and storage libraries such as DPDK [5] and SPDK [41] use *ring buffers* (i.e., bounded circular queues) for various purposes when allocating and transferring network frames, inter-process tracepoints, etc. Oftentimes, straight-forward implementations of ring buffers (e.g., the one found in DPDK) are erroneously dubbed as "lock-free" or "non-blocking", whereas those queues merely avoid *explicit* locks but still lack true non-blocking progress guarantees.

More specifically, such queues require a thread to reserve a ring buffer slot prior to writing new data. These approaches, as previously discussed [9, 24], are technically blocking since one stalled (e.g., preempted) thread in the middle of an operation can adversely affect other threads such that they will be unable to make further progress until the first thread resumes. Although this restriction is not entirely unreasonable (e.g., we may assume that the number of threads roughly equals the number of physical cores, and preemption is rare), such queues leave much to be desired. As with explicit spin locks, lack of true lock-freedom results in suboptimal performance unless preemption is disabled. (This can be undesirable or harder to do, especially in user space.) Also, such queues

SPAA '22, July 11–14, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9146-7/22/07.
https://doi.org/10.1145/3490148.3538572

cannot be safely used outside thread contexts, e.g., OS interrupts. Specialized libraries [23] acknowledged [24] this problem by rolling back to Michael & Scott's classical FIFO lock-free queue [26], which is correct and easily portable to all platforms but slow.

Typically, true non-blocking FIFO queues are implemented using *Head* and *Tail* references, which are updated using the compare-and-swap (CAS) instruction. However, CAS-based approaches do not scale well as the contention grows [30, 31, 45] since *Head* and *Tail* have to be updated inside a CAS loop that can fail and repeat. Thus, previous works explored fetch-and-add (F&A) on the contended parts of FIFO queues: *Head* and *Tail* references. F&A always succeeds and consequently scales better. Using F&A typically implies that there exist some ring buffers underneath. Thus, prior works have focused on making these ring buffers efficient. However, ring buffer design through F&A is not trivial when true lock- or wait-free progress is required. In fact, lock-free ring buffers historically needed CAS [20, 42].

Until recently, efficient ring buffers [30, 45] were livelock-prone, and researchers attempted to workaround livelock issues by using additional layers of slower CAS-based queues. SCQ [31] took a different approach by constructing a truly lock-free ring buffer that uses F&A. Unfortunately, SCQ still lacks stronger wait-free progress guarantees.

The literature presents many approaches for building wait-free data structures. Kogan & Petrank's *fast-path-slow-path* methodology [19] uses a lock-free procedure for the fast path, taken most of the time, and falls back to a wait-free procedure if the fast path does not succeed. However, the methodology only considers CAS, and the construction of algorithms that heavily rely on F&A for improved performance is unclear. (Substituting F&A with a more general CAS construct would erase performance advantages.)

To that end, Yang and Mellor-Crummey's (YMC) [45] wait-free queue implemented its own fast-path-slow-path method. But, as pointed out by Ramalhete and Correia [37], YMC's design is flawed in its memory reclamation approach which, strictly described, forfeits wait-freedom. This means that a user still has to choose from other wait-free queues which are slower, such as Kogan & Petrank's [18] and CRTurn [39] queues. These queues do not use F&A and scale poorly.

This prior experience reveals that solving multiple problems such as creating an unbounded queue, avoiding livelocks, and attaining wait-freedom in the same algorithm may not be an effective design strategy. Instead, a compartmentalized design approach may be more effective wherein at each step, we solve only one problem because it enables reasoning about separate components in isolation. For example, lock-free SCQ achieves great performance and memory efficiency. SCQ can be extended to attain wait-freedom.

In this paper, we present Wait-Free Circular Queue (or wCQ). wCQ uses its own variation of fast-path-slow-path methodology (Section 3) to attain wait-freedom and bound memory usage. When falling back to the slow path after bounded number of attempts on the fast path, wCQ's threads collaborate with each other to ensure wait-freedom. wCQ requires double-width CAS, which is nowadays widespread (i.e., x86 and ARM/AArch64). However, we also illustrate how wCQ can be implemented on architectures that lack such instructions including PowerPC and MIPS (see Section 4). We analyze wCQ's properties including wait-freedom and bounded memory usage (Section 5). Our evaluations on x86 and PowerPC architectures validate wCQ's excellent performance and memory efficiency (Section 6). Additionally, they show that wCQ's performance closely matches SCQ's.

wCQ is the first fast wait-free queue with bounded memory usage. Armed with wCQ, the final goal of creating an unbounded wait-free queue is also realistic, as existing slower wait-free queues (e.g., CRTurn [39]) can link faster wait-free ring buffers together.

2 PRELIMINARIES

For the sake of presentation, we assume a **sequentially consistent** memory model [22], as otherwise, the pseudo-code becomes cluttered with implementation-specific barrier code. Our implementation inserts memory barriers wherever strongly-ordered shared memory writes are necessary.

Wait-Free Progress. In the literature, several categories of non-blocking data structures are considered. In *obstruction-free* structures, progress is only guaranteed if one thread runs in isolation from others. In *lock-free* structures, *one* thread is always guaranteed to make progress in a finite number of steps. Finally, in *wait-free* structures, *all* threads are always guaranteed to make progress in a finite number of steps. In lock-free structures, individual threads may starve, whereas wait-freedom also implies starvation-freedom. Unsurprisingly, wait-free data structures are the most challenging to design, but they are especially useful for latency-sensitive applications which often have quality of service constraints.

Memory usage is an important property that is sometimes overlooked in the design of non-blocking data structures. All *truly* non-blocking algorithms must bound memory usage. Otherwise, no further progress can be made when memory is exhausted. Therefore, such algorithms are inherently blocking. True wait-freedom, thus, also implies bounded memory usage. Despite many prior attempts to design wait-free FIFO queues, the design of *high-performant* wait-free queues, which also have bounded memory usage, remains challenging – exactly the problem that we solve in this paper.

Read-Modify-Write. Lock-free and wait-free algorithms typically use read-modify-write (RMW) operations, which atomically read a memory variable, perform some operation on it, and write back the result. Modern CPUs implement RMWs via compare-and-swap (CAS) or a pair of load-link (LL)/store-conditional (SC) instructions. For better scalability, some architectures, such as x86-64 [15] and AArch64 [2], support specialized operations such as F&A (fetch-and-add), OR (atomic OR), and XCHG (exchange). The execution time of these specialized instructions is bounded, which makes them especially useful for wait-free algorithms. Since the above operations are unconditional, even architectures that merely implement them via LL/SC (but not CAS) may still achieve bounded execution time depending on whether their LL reservations occur in a wait-free manner. (Note that LL/SC can still fail due to OS events, e.g., interrupts.) x86-64 and AArch64 also support (double-width) CAS on two *contiguous* memory words, which we refer to as CAS2. CAS2 must not be confused with double-CAS, which updates two *arbitrary* words but is rarely available in commodity hardware.

Infinite Array Queue. Past works [30, 31, 45] argued that F&A scales *significantly* better than a CAS loop under contention and proposed

```
void *Array[INFINITE_SIZE];
int Tail = 0, Head = 0;
void Enqueue(void * p)
  while True do
    T = F&A(&Tail, 1);
    v = XCHG(&Array[T], p);
    if ( v = ⊥ ) return;
```

```
void *Dequeue()
  do          // While not empty
    H = F&A(&Head, 1);
    p = XCHG(&Array[H], ⊤);
    if ( p ≠ ⊥ ) return p;
  while Load(&Tail) > H + 1;
  return null;
```

Figure 1: Livelock-prone infinite array queue.

```
bool Enqueue_Ptr(void *p)
  int index = fq.Dequeue();
  if ( index = ∅ )              // Full
    return False;
  data[index] = p;
  aq.Enqueue(index);
  return True;                  // Done
```

```
void *Dequeue_Ptr()
  int index = aq.Dequeue();
  if ( index = ∅ )              // Empty
    return null;
  void *p = data[index];
  fq.Enqueue(index);
  return p;                     // Done
```

Figure 2: Storing pointers via indirection.

to use ring buffers. However, building a correct and performant ring buffer with F&A is challenging.

To help with reasoning about this problem, literature [13, 30] describe the concept of an "infinite array queue." The queue [30] assumes that we have an infinite array (i.e., memory is unlimited) and old array entries need not be recycled (Figure 1). The queue is initially empty, and its array entries are set to a reserved $\perp$ value. When calling *Enqueue*, *Tail* is incremented using F&A. The value returned by F&A will point to an array entry where a new element can be inserted. *Enqueue* succeeds if the previous value was $\perp$. Otherwise, some dequeuer arrived before the corresponding enqueuer, and thus the entry was modified to prevent the enqueuer from using it. In the latter case, *Enqueue* will attempt to execute F&A again. *Dequeue* also executes F&A on *Head*. F&A retrieves the position of a previously enqueued entry. To prevent this entry from being reused by a late enqueuer, *Dequeue* places a reserved $\top$ value. If the previous value is not $\perp$, *Dequeue* returns that value, which was previously inserted by *Enqueue*. Otherwise, *Dequeue* attempts to execute F&A again.

Note that this infinite queue is clearly susceptible to livelocks since all *Dequeue* calls may get well ahead of *Enqueue* calls, preventing any further progress. Prior queues, such as LCRQ [30], YMC [45], and SCQ [31] are all inspired by this infinite queue, and they all implement ring buffers that use finite memory. However, they diverge on how to address the livelock problem. LCRQ workarounds livelocks by "closing" stalled ring buffers. YMC tries to hit two birds with one stone by assisting unlucky threads while also ensuring wait-freedom (but the approach is flawed as previously discussed). Finally, SCQ uses a special "threshold" value to construct a lock-free ring buffer instead. In this paper, we use and extend SCQ's approach to create a wait-free ring buffer.

SCQ Algorithm. In SCQ [31], a program comprises of k threads, and the ring buffer size is n, which is a power of two. SCQ also reasonably assumes that $k \leq n$. wCQ, discussed in this paper, makes identical assumptions.

One of SCQ's key ideas is indirection. SCQ uses two ring buffers which store "allocated" and "freed" indices in **aq** and **fq**, respectively. Data entries (which are pointers or any fixed-size data) are stored in a separate array which is referred to by these indices. For example, to store pointers (Figure 2), *Enqueue_Ptr* dequeues an index from **fq**, initializes an array entry with a pointer, and inserts the index into **aq**. A counterpart *Dequeue_Ptr* dequeues this index from **aq**, reads the pointer, and inserts the index back into **fq**.

SCQ's *Enqueue* (for either **aq** or **fq**) need not check if the corresponding queue is full since the maximum number of indices is n. This greatly simplifies SCQ's design since only *Dequeue* needs to be handled specially for an empty queue. SCQ is *operation-wise* lock-free: at least one enqueuer and one dequeuer both succeed after a finite number of steps.

Figure 3 presents the SCQ algorithm. The algorithm provides a practical implementation of the infinite array queue that was previously discussed. Since memory is finite in practice, the same array entry can be used over and over again in a cyclic fashion. Thus, to distinguish recycling attempts, SCQ records *cycles*. XCHG is replaced with CAS as the same entry can be accessed through different cycles. LCRQ [30] uses a very similar idea.

Each ring buffer in SCQ keeps its *Head* and *Tail* references. Using these references, *Enqueue* and *Dequeue* can determine a position in the ring buffer ($j = Tail \mod n$) and a current cycle ($cycle = Tail \div n$). Each *entry* in SCQ ring buffers records *Cycle* and *Index* for the inserted entry. Since each entry has an implicit position, only the *Cycle* part needs to be recorded. (Note that *Index*'s bit-length matches the position bit-length since n is the maximum value for both; thus *Index* and *Cycle* still fit in a single word.) SCQ also reserves one bit in each entry: *IsSafe* handles corner cases when an entry is still occupied by some old cycle but some newer dequeuer needs to mark the entry unusable.

All entries are updated sequentially. To reduce false sharing, SCQ permutes queue positions by using *Cache_Remap*, which places two adjacent entries into different cache lines; the same cache line is not reused as long as possible.

When *Enqueue* successfully inserts a new index, it resets a special *Threshold* value to the maximum. The threshold value is decremented by *Dequeue* when the latter is unable to make progress. A combination of these design choices, justified in [31], allows SCQ to achieve lock-freedom directly inside the ring buffer itself.

The SCQ paper [31] provides a justification for the maximum threshold value when using the original infinite array queue as well as when using a practical finite queue (SCQ). For the infinite queue, the threshold is $2n - 1$, where n is the maximum number of entries. This value is obtained by observing that the last dequeuer is at most n slots away from the last inserted entry; additionally there may be at most $n - 1$ dequeuers that precede the last dequeuer.

Finite SCQ, however, is more intricate. To retain lock-freedom, it additionally requires to **double the capacity** of ring buffers, i.e., it allocates $2n$ entries while only ever using n entries at any point of time. Consequently, SCQ also needs to increase the threshold value to $3n - 1$ since the last dequeuer can be $2n$ slots away from the last inserted entry. We **retain same threshold and capacity** in wCQ.

Although not done in the original SCQ algorithm, for convenience, we distinguish two cases for *Dequeue* in Figure 3. When *Dequeue* arrives prior to its *Enqueue* counterpart, it places $\perp$. If *Dequeue* arrives on time, when the entry can already be *consumed*, it inserts $\perp_c$. The distinction between $\perp_c$ and $\perp$ will become useful

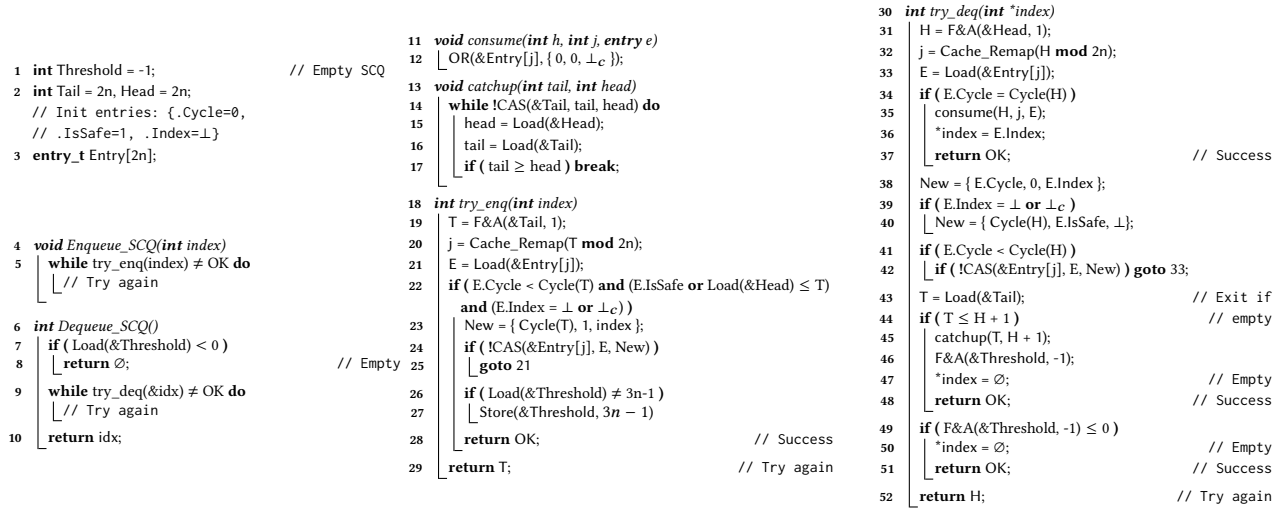


Figure 3: Lock-free circular queue (SCQ): *Enqueue* and *Dequeue* are identical for both aq and fq.

when discussing wCQ. SCQ assigns $\perp_c = 2n-1$ so that all lower bits of the number are ones ($2n$ is a power of two). This is convenient when consuming elements in Line 12, which can simply execute an atomic OR operation to replace the index with $\perp_c$ while keeping all other bits intact. We further assume that $\perp = 2n - 2$. Neither $\perp_c$ nor $\perp$ overlaps with any actual indices $[0..n-1]$.

Kogan & Petrank's method. This wait-free method [19] implies that a fast path algorithm (ideally with a performance similar to a lock-free algorithm) is attempted multiple times (MAX_PATIENCE in our paper). When no progress is made, a slow path algorithm will guarantee completion after a finite number of steps, though with a higher performance cost. The slow path expects collaboration from other threads. Periodically, when performing data structure operations, *every* thread checks if any other thread needs helping.

The original methodology dynamically allocates slow path helper descriptors, which need to be reclaimed. But dynamic memory allocation makes it trickier to guarantee bounded memory usage, as experienced by YMC. Also, it is not clear how to apply the methodology when using specialized (e.g., F&A) instructions. In this paper, we address these issues for SCQ.

3 WAIT-FREE CIRCULAR QUEUE (WCQ)

wCQ's key insight is to avoid memory reclamation issues altogether. Because wCQ only needs per-thread descriptors and the ring buffer itself, it does not need to deal with dynamic memory allocation. The original Kogan & Petrank's fast-path-slow-path methodology cannot be used as-is due to memory reclamation concerns as well as lack of F&A support. Instead, wCQ uses a variation of this methodology specifically designed for SCQ. All threads collaborate to guarantee wait-free progress.

Assumptions. Generally speaking, wCQ requires double-width CAS (CAS2) to properly synchronize concurrent queue alterations. wCQ also assumes hardware-implemented F&A and atomic OR to guarantee wait-freedom. However, these requirements are not very strict. (See Section 3.3 and Section 4 for more details.)

Data Structure. Figure 4 shows changes to the SCQ structures, described in Section 2. Each entry is extended to a *pair*. A pair comprises of the original entry *Value* and a special *Note*, discussed later. Each thread maintains per-thread state, the **thrdrec_t** record. Its private fields are only used by the thread when it attempts to *assist* other threads. In contrast, its shared fields are used to *request help*.

3.1 Bird's-Eye View

Figure 5 shows the *Enqueue_wCQ* and *Dequeue_wCQ* procedures. *Enqueue_wCQ* first checks if any other thread needs help by calling *help_threads*, after which it attempts to use the fast path to insert an entry. The fast path is identical to *Enqueue_SCQ*. When exceeding MAX_PATIENCE iterations, *Enqueue_wCQ* takes the slow path. In the slow path, *Enqueue_wCQ* requests help by recording its last *Tail* value that was tried (in *initTail* and *localTail*) and the *index* input parameter. The *initTail* and *localTail* are initially identical and only diverge later (see *slow_F&A* below). Additionally, *Enqueue_wCQ* sets extra flags to indicate an active enqueue help request. Since the entire request is to be read atomically, we use *seq1* and *seq2* to verify the integrity of reads. If *seq1* does not match *seq2*, no valid request exists for the thread. Each time a slow path is complete, *seq1* is incremented (Line 28). Each time a new request is made, *seq2* is set to *seq1* (Line 24). Subsequently, *enqueue_slow* is called to take the slow path.

A somewhat similar procedure is used for *Dequeue_wCQ*, with the exception that *Dequeue_wCQ* also needs to check if the queue is empty. After completing the slow path, the output result needs to be gathered. In SCQ, output is merely consumed by using atomic OR. In wCQ, *consume* is extended to mark all pending enqueuers, as discussed below.

```
struct phase2rec_t {
  int seq1;                          // = 1
  int *local;
  int cnt;
  int seq2;        };                // = 0
struct entpair_t {
  int Note;                          // = -1
  entry_t Value;        // = { .Cycle=0,
};   // .IsSafe=1, .Enq=1, .Index=⊥ }
entpair_t Entry[2n];
thrdrec_t Record[NUM_THRDS];
int Threshold = -1;            // Empty wCQ
int Tail = 2n, Head = 2n;

struct thrdrec_t {
 // === Private Fields ===
 int nextCheck;     // = HELP_DELAY
 int nextTid;            // Thread ID
 // === Shared Fields ===
 phase2rec_t phase2;     // Phase 2
 int seq1;                          // = 1
 bool enqueue;
 bool pending;             // = false
 int localTail, initTail;
 int localHead, initHead;
 int index;
 int seq2         };               // = 0
```

Figure 4: Auxiliary structures.

```
1  void consume(int h, int j, entry_t e)
2    if ( !e.Enq ) finalize_request(h);
3    OR(&Entry[j].Value, { 0, 0, 1, ⊥c });
4  void finalize_request(int h)
5    i = (TID + 1) mod NUM_THRDS;
6    while i != TID do
7      int *tail = &Record[i].localTail;
8      if ( Counter(*tail) = h )
9        CAS(tail, h, h | FIN);
10       return;
11     i = (i + 1) mod NUM_THRDS;
12 void Enqueue_wCQ(int index)
13   help_threads();
     // == Fast path (SCQ) ==
14   int count = MAX_PATIENCE;
15   while −−count ≠ 0 do
16     tail = try_enq(index);
17     if ( tail = OK ) return;
     // == Slow path (wCQ) ==
18   thrdrec_t *r = &Record[TID];
19   int seq = r->seq1;
20   r->localTail = tail;
21   r->initTail = tail;
22   r->index = index;
23   r->enqueue = true;
24   r->seq2 = seq;
25   r->pending = true;
26   enqueue_slow(tail, index, r);
27   r->pending = false;
28   r->seq1 = seq + 1;
29 int Dequeue_wCQ()
30   if ( Load(&Threshold) < 0 )
31     return ∅;                // Empty
32   help_threads();
     // == Fast path (SCQ) ==
33   int count = MAX_PATIENCE;
34   while −−count ≠ 0 do
35     int idx;
36     head = try_deq(&idx);
37     if ( head = OK ) return idx;
     // == Slow path (wCQ) ==
38   thrdrec_t *r = &Record[TID];
39   int seq = r->seq1;
40   r->localHead = head;
41   r->initHead = head;
42   r->enqueue = false;
43   r->seq2 = seq;
44   r->pending = true;
45   dequeue_slow(head, r);
46   r->pending = false;
47   r->seq1 = seq + 1;
     // Get slow-path results
48   h = Counter(r->localHead);
49   j = Cache_Remap(h mod 2n);
50   Ent = Load(&Entry[j].Value);
51   if ( Ent.Cycle = Cycle(h) and
       Ent.Index ≠ ⊥ )
52     consume(h, j, Ent);
53     return Ent.Index;          // Done
54 return ∅
```

Figure 5: Wait-free circular queue (wCQ).

Helping Procedures. Figure 6 shows the *help_threads* code, which is inspired by the method of Kogan & Petrank [19]. Each thread skips HELP_DELAY iterations using *nextCheck* to amortize the cost of *help_threads*. The procedure circularly iterates across all threads, *nextTid*, to find ones with a pending helping request. Finally, *help_threads* calls *help_enqueue* or *help_dequeue*. A help request is atomically loaded and passed to *enqueue_slow* and *dequeue_slow*.

3.2 Slow Path

Either a helpee or its helpers eventually call *enqueue_slow* and *dequeue_slow*. wCQ's key idea is that eventually all active threads assist a thread that is stuck if progress is not made. One of these threads will eventually succeed due to the underlying SCQ's lock-free guarantees. However, all helpers should repeat *exactly* the same procedure as the helpee. This can be challenging since the ring buffer keeps changing.

```
1  void help_threads()
2    thrdrec_t *r = &Record[TID];
3    if ( −−r->nextCheck ≠ 0 )
4      return;
5    thr = &Record[r->nextTid];
6    if ( thr->pending )
7      if ( thr->enqueue )
8        help_enqueue(thr);
9      else
10       help_dequeue(thr);
11   r->nextCheck = HELP_DELAY;
12   r->nextTid = (r->nextTid + 1) mod
       NUM_THRDS;
13 void help_enqueue(thrdrec_t *thr)
14   int seq = thr->seq2;
15   bool enqueue = thr->enqueue;
16   int idx = thr->index;
17   int tail = thr->initTail;
18   if ( enqueue and thr->seq1 = seq )
19     enqueue_slow(tail, idx, thr);
20 void help_dequeue(thrdrec_t *thr)
21   int seq = thr->seq2;
22   bool enqueue = thr->enqueue;
23   int head = thr->initHead;
24   if ( !enqueue and thr->seq1 = seq )
25     dequeue_slow(head, thr);
```

Figure 6: wCQ's helping procedures.

More specifically, multiple *enqueue_slow* calls are to avoid inserting the same element multiple times into different positions. Likewise, *dequeue_slow* should only consume one element. Figure 7 shows wCQ's approach for this problem.

In Figure 7, a special *slow_F&A* operation substitutes F&A from the fast path. The key idea is that for any given helpee and its helpers, the global *Head* and *Tail* values need to be changed only once per each iteration across all cooperative threads (i.e., a helpee and its helpers). To support this, each thread record maintains *initTail*, *localTail*, *initHead*, and *localHead* values. These values are initialized from the last tail and head values from the fast path accordingly. In the beginning, the init and local values are identical. The init value is a starting point for *all* helpers (Lines 17 and 23, Figure 6). The local value represents the last value in *slow_F&A*. To support *slow_F&A*, we redefine the global *Head* and *Tail* values to be *pairs* of counters with pointers rather than just counters. (The pointer component is initially **null**.) Fast-path procedures only use hardware F&A on counters leaving pointers intact. However, slow-path procedures use the pointer component to store the second phase request (see below). We use the fact that paired counters are monotonically increasing to prevent the ABA problem for pointers.

The local value is updated to the global counter (Line 25, Figure 7) by one of the cooperative threads. This value is compared against the prior value stored on stack (T or H) such that one and only one thread updates the local value. Since the global value also needs to be consistently changed, we use a special protocol. In the first phase, the local value is set to the global value with the INC flag (Line 25). Then, the global value is incremented (Line 32). In the second phase, the local value resets the INC flag in Line 34 (unless it was changed by a concurrent thread in Line 86 already).

Several corner cases for *try_enq_slow* need to be considered. One case is when an element is already inserted by another thread (Line 19, Figure 7). Another case is when the condition in Line 6 is true. If one helper skips the entry, we want other helpers to do the same since the condition can become otherwise false at some later point. To that end, *Note* is advanced to the current tail cycle, which allows Line 5 to skip the entry for later helpers.

Finally, we want to terminate future helpers if the entry is already consumed and reused for a later cycle. For this purpose, entries are inserted using the two-step procedure. We reserve an additional *Enq* bit in entries. First, the entry is inserted with Enq=0. Then the helping request is finalized by setting FIN in Line 14. Any

concurrent dequeuer that observes Enq=0 will help setting FIN (Line 2, Figure 5). FIN is set directly to the thread record's *localTail* value by stealing one bit to prevent any future increments with *slow_F&A*. Finally, either of the two threads will flip Enq to 1, at which point the entry can be consumed.

Similar corner cases exist in *try_deq_slow*, where *Note* prevents reusing a previously invalidated entry. *try_deq_slow* also uses FIN to terminate all dequeuers when the result is detected. *slow_F&A* takes care of synchronized per-request increments (by using another bit, called INC) and also terminates helpers when detecting FIN.

Since thresholds are to be decremented only once, *slow_F&A* decrements the threshold when calling it from *try_deq_slow*. (Decrementing *prior* to dequeueing is still acceptable.)

Second Phase Request. When incrementing the global counter in slow_F&A, a thread must update the local value to the previous global value. The thread tentatively sets the counter in Phase 1, but the INC flag must be cleared in Phase 2, at which point all cooperative threads will use the same local value. We request help from cooperative threads by using the *phase2* pointer. This pointer is set simultaneously when the global value increments (Line 32, Figure 7). The previous value will be recorded in the *phase2* request.

One corner case arises when the fast-path procedure unconditionally increments the global value while *phase2* is already set. This may cause Line 87 to sporadically fail even though Line 86 succeeds. In this case, Line 86 is repeated without any side effect until Line 87 succeeds or the *phase2* request is finalized by some thread. All fast-path threads will eventually be converging to help the thread that is stuck in the slow path.

Decrementing Threshold. One important difference with *try_deq* is that *try_deq_slow* decrements *Threshold* for every iteration inside *slow_F&A*. The reason for that is that *try_deq_slow* can also be executed concurrently by helpers. Thus, to retain the original threshold bound ($3n-1$), we must make sure that only *one* cooperative thread decrements the threshold value. The global *Head* value is an ideal source for such synchronization since it only changes once per *try_deq_slow* iteration across all cooperative threads. Thus, we decrement *Threshold prior* to the actual dequeue attempt. Note that *try_deq* is doing it after a *failure*, which is merely a performance optimization since the $3n-1$ bound is computed in the original SCQ algorithm regardless of the status of an operation. This performance optimization is obviously irrelevant for the slow path.

Bounding catchup. Since catchup merely improves performance in SCQ by reducing contention, there is no problem in explicitly limiting the maximum number of iterations. We do so to avoid unbounded loops in catchup.

Example. Figure 8 shows an enqueue scenario with three threads. Thread 3 attempts to insert index V3 but is stuck and requested help. Thread 3 does not make any progress on its own. (The corresponding thread record is denoted as T3.) Thread 1 and Thread 2 eventually observe that Thread 3 needs helping. This example shows that slow_F&A for both Thread 1 and Thread 2 converges to the same value (8). The global Tail value is only incremented once (by Thread 1). The corresponding entry is also only updated once (by Thread 1). For simplicity, we omit Cache_Remap's permutations.

3.3 Hardware Support

Our algorithm implies hardware F&A and atomic OR for wait-freedom. However, they are not strictly necessary since failing F&A in the fast path can simply force us to fall back to the slow path. There, threads eventually collaborate, bounding the execution time of F&A, which is still used for *Threshold*. Likewise, output can be consumed inside *dequeue_slow*, making it possible to emulate atomic OR with CAS while bounding execution time. We omit the discussion of these changes due to their limited practical value, as both wait-free F&A and OR are available on x86-64 and AArch64.

4 ARCHITECTURES WITH ORDINARY LL/SC

Outside of the x86(-64) and ARM(64) realm, CAS2 may not be available. In this section, we present an approach that we have used to implement wCQ on PowerPC [14] and MIPS [28]. Other architectures may also adopt this approach depending upon what operations are permitted between their LL and SC instructions.

We first note that CAS2 for global *Head* and *Tail* pairs can simply be replaced with regular CAS by packing a small thread index (in lieu of the *phase2* pointer) with a reduced (e.g., 48-bit rather than 64-bit) counter. However, this approach is less suitable for CAS2 used for entries (using 32-bit counters is risky due to a high chance of an overflow). Thus, we need an approach to store two words.

Typical architectures implement only a weak version of LL/SC, which has certain restrictions, e.g., memory alterations between LL and SC are disallowed. Memory reads in between, however, are still allowed for architectures such as PowerPC and MIPS. Furthermore, LL's reservation granularity is larger than just a memory word (e.g., L1 cache line for PowerPC [40]). This typically creates a problem known as "false sharing," when concurrent LL/SC on unrelated variables residing in the same cache line cause SC to succeed only for one variable. This often requires careful consideration by the programmer to properly align data to avoid false sharing.

In wCQ's slow-path procedures, both *Value* and *Note* components of entries need to be atomically loaded, but we only update one or the other variable at a time. We place two variables in the same reservation granule by aligning the first variable on the double-word boundary, so that only one LL/SC pair succeeds at a time. We use a regular Load operation between LL and SC to load the other variable atomically. We also construct an implicit memory fence for Load by introducing artificial data dependency, which prevents reordering of LL and Load. For the SC to succeed, the other variable from the same reservation granule must remain intact.

In Figure 9, we present two replacements of CAS2. We use corresponding versions that modify *Value* or *Note* components of entries. These replacements implement weak CAS semantics (i.e., sporadic failures are possible). Also, only single-word load atomicity is guaranteed when CAS fails. Both restrictions are acceptable for wCQ.

5 CORRECTNESS

Lemma 5.1. *wCQ's fast paths in Enqueue_wCQ and Dequeue_wCQ have a finite number of iterations for any thread.*

Proof. The number of iterations with loops containing *try_enq* or *try_deq* are limited by MAX_PATIENCE. Each *try_enq* has only one loop in Line 25, Figure 3. That loop will continue as long as

```
1  bool try_enq_slow(int T, int index, thrdrec_t *r)
2    j = Cache_Remap(T mod 2n);
3    Pair = Load(&Entry[j]);
4    Ent = Pair.Value;
5    if ( Ent.Cycle < Cycle(T) and Pair.Note < Cycle(T) )
6      if ( !(Ent.IsSafe or Load(&Head) ≤ T) or   (Ent.Index ≠ ⊥, ⊥c) )
7        N.Value = Ent;                          // Avert helper enqueuers from using it
8        N.Note = Cycle(T);
9        if ( !CAS2(&Entry[j], Pair, N) ) goto 3;
10       return False;                                          // Try again
       // Produce an entry: .Enq=0
11     N.Value = { Cycle(T), 1, 0, index };
12     N.Note = Pair.Note;
13     if ( !CAS2(&Entry[j], Pair, N) ) goto 3;               // The entry has changed
14     if ( CAS(&r->localTail, T, T | FIN) )                  // Finalize the help request
15       Pair = N;
16       N.Value.Enq = 1;
17       CAS2(&Entry[j], Pair, N);
18     if ( Load(&Threshold) ≠ 3n-1 ) Store(&Threshold, 3n-1);
19   else if ( Ent.Cycle ≠ Cycle(T) ) return False; // Not yet inserted by another thread
20   return True;                                                 // Success
21 bool slow_F&A(intpair *globalp, int *local, int *v, int *thld)
22   phase2rec_t *phase2 = &Record[TID].phase2;
     // Global Tail/Head are replaced with {.cnt, .ptr} pairs:
     // use only .cnt for fast paths; .ptr stores 'phase2'.
     // INVARIANT: *local, *v < *global ('global' has the next value)
     // VARIABLES: 'global', thread-record ('local'), on-stack ('v'):
     // 'local' syncs helpers so that 'global' increments only once
     // Increment 'local' (with INC, Phase 1), then 'global',
23   do            // finally reset INC (Phase 2). 'thld' is only used for Head.
24     int cnt = load_global_help_phase2(globalp, local);
25     if ( cnt = ∅ or !CAS(local, *v, cnt | INC) )           // Phase 1
26       *v = *local;
27       if ( *v & FIN ) return False;                  // Loop exits if FIN=1
28       if ( !(*v & INC) ) return True;                // Already incremented
29       cnt = Counter(*v);
30     else  *v = cnt | INC;                              // Phase 1 completes
31     prepare_phase2(phase2, local, cnt);              // Prepare help request
32   while !CAS2(globalp, { cnt, null }, { cnt + 1, phase2 });
     // Loops are finite, all threads eventually help the thread that is stuck
33   if ( thld ) F&A(thld, -1);
34   CAS(local, cnt | INC, cnt);
35   CAS2(globalp, { cnt + 1, phase2 }, { cnt + 1, null });
36   *v = cnt;                                            // Phase 2 completes
37   return True;                                                 // Success
38 void prepare_phase2(phase2rec_t *phase2, int *local, int cnt)
39   int seq = ++phase2->seq1;
40   phase2->local = local;
41   phase2->cnt = cnt;
42   phase2->seq2 = seq;
```

```
43 int try_deq_slow(int H, thrdrec_t *r)
44   j = Cache_Remap(H mod 2n);
45   Pair = Load(&Entry[j]);
46   Ent = Pair.Value;
     // Ready or consumed by helper (⊥c or value)
47   if ( Ent.Cycle = Cycle(H) and Ent.Index ≠ ⊥ )
48     CAS(&r->localHead, H, H | FIN);                  // Terminate helpers
49     return True;                                                 // Success
50   N.Note = Pair.Note;
51   Val = { Cycle(H), Ent.IsSafe, 1, ⊥ };
52   if ( Ent.Index ≠ ⊥, ⊥c )
53     if ( Ent.Cycle < Cycle(H) and  Pair.Note < Cycle(H) )   // Avert helper dequeuers
54       N.Value = Pair.Value;                                  // from using it
55       N.Note = Cycle(H);
56       r = CAS2(&Entry[j], Pair, N)
57       if ( !r ) goto 45;
58     Val = { Ent.Cycle, 0, Ent.Enq, Ent.Index };
59   N.Value = Val;
60   if ( Ent.Cycle < Cycle(H) )
61     if ( !CAS2(&Entry[j], Pair, N) )
62       goto 45;
63   T = Load(&Tail);                                       // Exit if queue
64   if ( T ≤ H + 1 )                                            // is empty
65     catchup(T, H + 1);
66   if ( Load(&Threshold) < 0 )
67     CAS(&r->localHead, H, H | FIN);
68     return True;                                                 // Success
69   return False;                                                // Try again
70 void enqueue_slow(int T, int index, thrdrec_t *r)
71   while slow_F&A(&Tail, &r->localTail, &T, null) do
72     if ( try_enq_slow(T, index, r) ) break;
73 void dequeue_slow(int H, thrdrec_t *r)
74   int *thld = &Threshold;
75   while slow_F&A(&Head,&r->localHead,&H, thld) do
76     if ( try_deq_slow(H, r) ) break;
77 int load_global_help_phase2(intpair *globalp, int *mylocal)
78   do          // Load globalp & help complete Phase 2 (i.e., make .ptr=null)
       // This loop is finite, see the outer loop and FIN
79     if ( *mylocal & FIN ) return ∅;                    // The outer loop exits
80     intpair gp = Load(globalp);                      // the thread that is stuck
81     phase2rec_t *phase2 = gp.ptr;
82     if ( phase2 = null ) break;                      // No help request, exit
83     int seq = phase2->seq2;
84     int *local = phase2->local;
85     int cnt = phase2->cnt;
       // Try to help, fails if 'local' was already advanced
86     if ( phase2->seq1 = seq ) CAS(local, cnt | INC, cnt);
       // No ABA problem (on .ptr) as .cnt increments monotonically
87   while !CAS2(globalp, { gp.cnt, phase2 }, { gp.cnt, null });
88   return gp.cnt;                                 // Return the .cnt component only
```

Figure 7: wCQ's slow-path procedures.

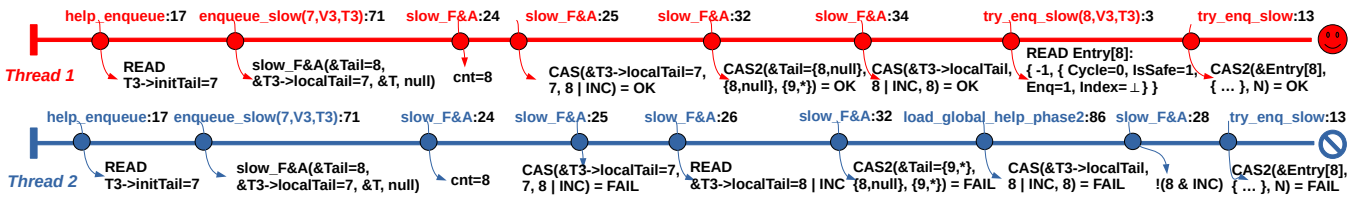


Figure 8: Enqueue slow-path execution example (ignoring CacheRemap's permutations).

Entry[j] is changed elsewhere. If several enqueuers contend for the same entry (wrapping around), we at most have $k \leq n$ concurrent enqueuers, each of which may cause a retry. Since each retry happens when another enqueuer succeeds and modifies the cycle, *Entry[j].Cycle* will eventually change after at most k unsuccessful attempts such that the loop is terminated (i.e., $E.Cycle < Cycle(T)$

```
1  bool CAS2_Value(entpair_t *Var,
       entpair_t Expect, entpair_t New)
2    Prev.Value = LL(&Var->Value);
3    Prev.Note = Load(&Var->Note);
4    if ( Prev ≠ Expect ) return False;
5    return SC(&Var->Value, New.Value);
6  bool CAS2_Note(entpair_t *Var,
       entpair_t Expect, entpair_t New)
7    Prev.Note = LL(&Var->Note);
8    Prev.Value = Load(&Var->Value);
9    if ( Prev ≠ Expect ) return False;
10   return SC(&Var->Note, New.Note);
```

Figure 9: CAS2 implementation for wCQ using LL/SC.

is no longer true). Likewise, with respect to contending dequeuers, *Entry[j].Cycle* can change for at most $k+(3n-1) \leq 4n-1$ iterations (due to the threshold), at which point all contending dequeuers will change *Entry[j]* such that the loop in *try_enq* terminates (either $E.Cycle \geq Cycle(T)$ or $Load(\&Head) > T$ for non-safe entries). Additionally, while dequeuing, for each given cycle, *Index* can change *once* to $\perp$ or $\perp_c$, and *IsSafe* can be reset (once). The argument is analogous for *try_deq*, which also has a similar loop in Line 42. *try_deq* also calls *catchup*, for which we explicitly limit the number of iterations in wCQ (it is merely an optimization in SCQ). □

Lemma 5.2. *Taken in isolation from slow paths, wCQ's fast paths are linearizable.*

Proof. Fast paths are nearly identical to the SCQ algorithm, which is linearizable. *catchup* is merely an optimization: it limits the number of iterations and does not affect correctness arguments. *consume* in wCQ, called from *try_deq*, internally calls *finalize_request* when Enq=0 (Line 2, Figure 5). The purpose of that function is to merely set the FIN flag for the corresponding enqueuer's local tail pointer (thread state) in the slow path, i.e., does not affect any global state. Enq=0 is only possible when involving slow paths. □

Lemma 5.3. *If one thread is stuck, all other threads eventually converge to help it.*

Proof. If the progress is not being made by one thread, all active threads will eventually attempt to help it (finishing their prior slow-path operations first). Some thread always succeeds since the underlying SCQ algorithm is (operation-wise) lock-free. Thus, the number of active threads in the slow path will reduce one by one until the very last stuck thread remains. All active helpers will then attempt to help that last thread, and one of them will succeed. □

Lemma 5.4. *slow_F&A does not alter local or global head/tail values as soon as a helpee or its helpers produce an entry.*

Proof. A fundamental property of *slow_F&A* is that it terminates the slow path in any remaining thread that attempts to insert the same entry as soon as the entry is produced (Lines 71 and 75, Figure 7). As soon as the entry is produced (Line 14), FIN is set. FIN is checked in Line 27 prior to changing either local or global head/tail further. In-flight threads, can still attempt to access previous ring buffer locations but this is harmless since previous locations are invalidated through the *Note* field. □

Lemma 5.5. *slow_F&A is wait-free bounded.*

Proof. CAS loops in *slow_F&A* and *load_global_help_phase2* (Lines 23-32 and Lines 78-87, Figure 7) are bounded since all threads will eventually converge to the slow path of one thread if no progress is being made according to Lemma 5.3. At that point, some helper will succeed and set FIN. After that, all threads stuck in *slow_F&A* will exit. □

Lemma 5.6. *slow_F&A decrements threshold only once per every global head change.*

Proof. In Figure 7, the threshold value is changed in Line 33. This is only possible after the global value was incremented by one (Line 32). □

Lemma 5.7. *Taken in isolation from fast paths, wCQ's slow paths are linearizable.*

Proof. There are several critical differences in the slow path when compared to the fast path. Specifically, *try_deq_slow* decrements the threshold value only once per an iteration across all cooperative threads (i.e., a helpee and its helpers). That follows from Lemma 5.6 since global *Head* changes once per such iteration – the whole point of *slow_F&A*.

Cooperative threads immediately stop updating global *Head* and *Tail* pointers when the result is produced (Lines 71 and 75, Figure 7), as it follows from Lemma 5.4.

Any cooperative thread always retrieves a value (the *v* variable) from *slow_F&A*, which is either completely new, or was previously observed by some other cooperative thread. As discussed in Section 3.2, a corner case arises when one cooperative thread already skipped a slot with a given cycle but a different cooperative thread (which is late) attempts to use that skipped slot with the same cycle. That happens due to the *IsSafe* bit or other conditions in Line 6, Figure 7. To guard against this scenario, the slow-path procedure maintains *Note*, which guarantees that late cooperative threads will skip the slot that was already skipped by one thread. (See Section 3.2 for more details.) □

Theorem 5.8. *wCQ's memory usage is bounded.*

Proof. wCQ is a statically allocated circular queue, and it never allocates any extra memory. Thread record descriptors are bounded by the total number of threads. □

Theorem 5.9. *wCQ is linearizable.*

Proof. Linearizability of the fast and slow paths in separation follows from Lemmas 5.2 and 5.7.

The fast path is fully compatible with the slow path. There are just some minor differences described below. The one-step procedure (Enq=1 right away) is used in lieu of the two-step procedure (Enq=0 and then Enq=1) when producing new entries. Likewise, the *Note* field is not altered on the fast path since it is only needed to synchronize cooperative threads, and we only have one such thread on the fast path. However, differences in Enq must be properly supported. To the end, the fast path dequeuers fully support the semantics expected by slow-path enqueuers with respect to the Enq bit. More specifically, *consume* always calls *finalize_request* for Enq=0 when consuming an element, which helps to set the corresponding FIN bit (Line 2, Figure 5) such that it does not need to wait until the slow-path enqueuer completes the Enq=0 to Enq=1 transition (Lines 14-17, Figure 7). □

Theorem 5.10. *Enqueue_wCQ/Dequeue_wCQ are wait-free.*

Proof. The number of iterations on the fast path (Figure 5) is finite according to Lemma 5.1. *enqueue_slow* (Line 26, Figure 5) terminates when *slow_F&A* (which is wait-free according to Lemma 5.5) returns false, which only happens when FIN is set for the tail pointer (Line 27, Figure 7). FIN is always set (Line 14, Figure 7) when some thread from the same helping request succeeds.

We also need to consider a case when a thread always gets unlucky and is unable to make progress inside *try_enq_slow* since entries keep changing. According to Lemma 5.3, in this situation, all other threads will eventually converge to help the thread that is stuck. Thus, the "unlucky" thread will either succeed by itself or the result will be produced by some helper thread, at which point the *slow_F&A* loop is terminated. (*Dequeue_wCQ* is wait-free by analogy.) □

6 EVALUATION

Since in Section 5 we already determined upper bounds for the number of iterations and they are reasonable, the primary goal of our evaluation is to make sure that this wCQ's stronger wait-free progress property does not come at a significant cost, as it is the case in present (true) wait-free algorithms such as CRTurn [37, 39]. Thus, our practical evaluation focuses on key performance attributes that can be quantitatively and fairly evaluated in *all* existing queues – memory efficiency and throughput. Specifically, our goal is to demonstrate that wCQ's performance in that respect is on par with that of SCQ since wCQ is based on SCQ.

We used the benchmark of [45] and extended it with the SCQ [31] and CRTurn [37, 39] implementations. We implemented wCQ and integrated it into the test framework.

We compare wCQ against state-of-the-art algorithms:

- MSQueue [26], a well-known Michael & Scott's lock-free queue which is not very performant.
- LCRQ [30], a queue that maintains a lock-free list of ring buffers. Ring buffers (CRQs) are livelock-prone and cannot be used alone.
- SCQ lock-free queue [31], a design which is similar to CRQ but is more memory efficient and avoids livelocks in ring buffers directly. wCQ is based on SCQ.
- YMC (Yang & Mellor-Crummey's) wait-free queue [45]. YMC is flawed (see the discussion in [37]) in its memory reclamation approach which, strictly described, forfeits wait-freedom. Since YMC uses F&A, it is directly relevant in our comparison against wCQ.
- CRTurn wait-free queue [37, 39]. This is a truly wait-free queue but as MSQueue, it is not very performant.
- CCQueue [8] is a combining queue, which is not lock-free but still achieves relatively good performance.
- FAA (fetch-and-add), which is not a true queue algorithm; it simply atomically increments *Head* and *Tail* when calling *Dequeue* and *Enqueue* respectively. FAA is only shown to provide a theoretical performance "upper bound" for F&A-based queues.

SimQueue [7] is another wait-free queue, but it lacks memory reclamation and does not scale as well as the fastest queues. Consequently, SimQueue is not presented here.

For wCQ, we set MAX_PATIENCE to 16 for *Enqueue* and 64 for *Dequeue*, which results in taking the slow path relatively infrequently.

To avoid cluttering, we do not separately present a combination of SCQ with MSQueue or wCQ with a slower wait-free unbounded queue since: (1) they *only* need that combination when an unbounded queue is desired (which is unlike LCRQ or YMC, where an outer layer is always required for progress guarantees) and (2) the costs of these unbounded queues were measured and found to be largely dominated by the underlying SCQ and wCQ implementations respectively. Memory reclamation overheads for queues are generally not that significant, especially when using more recent fast lock- and wait-free approaches [32–35, 38, 44]. As in [31, 45], we use customized reclamation for YMC and hazard pointers elsewhere (LCRQ, MSQueue, CRTurn).

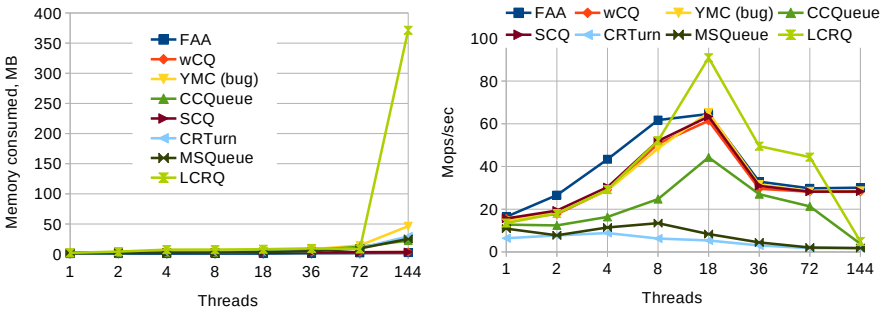
Since each queue has different properties and trade-offs, care must be taken when assessing their performance. For example, LCRQ can achieve higher performance in certain tests but it is not portable (e.g., cannot be implemented on PowerPC). Moreover, LCRQ is not as memory efficient as other queues such as SCQ. Likewise, YMC may exhibit performance similar to SCQ or wCQ, but its memory reclamation flaws need to be factored in as well.

We performed all tests on x86_64 and PowerPC machines. The x86_64 machine has 128GB of RAM and four Intel Xeon E7-8880 v3 (2.30GHz) processors, each with 18 cores with hyper-threading disabled. We used Ubuntu 18.04 LTS installation with gcc 8.4 (-O3 optimization). wCQ for x86_64 benefits from CAS2 and hardware-based F&A. The PowerPC machine has 64GB of RAM, 64 logical cores (each core has 8 threads), and runs at 3.0 Ghz (8MB L3 cache). We also used Ubuntu 18.04 LTS installation with gcc 8.4 (-O3 optimization). wCQ for PowerPC does not benefit from native F&A. Since CAS2 is unavailable on PowerPC, wCQ is implemented via LL/SC (see Section 4). LCRQ requires true CAS2, and its results are only presented for x86_64. On both x86_64 and PowerPC, we use jemalloc [6] due to its better performance [25].

The benchmark measures average throughput and protects against outliers. Each point is measured 10 times for 10,000,000 operations in a loop. The coefficient of variation, as reported by the benchmark, is small (< 0.01). We use a relatively small ring buffer (2^{16} entries) for wCQ and SCQ. For all other queues, we use the default parameters from [45] which appear to be optimal. Since contention across sockets is expensive, x86-64's throughput peaks for 18 threads (all 18 threads can fit just one physical CPU). Likewise, PowerPC's throughput peaks for 4-8 threads.

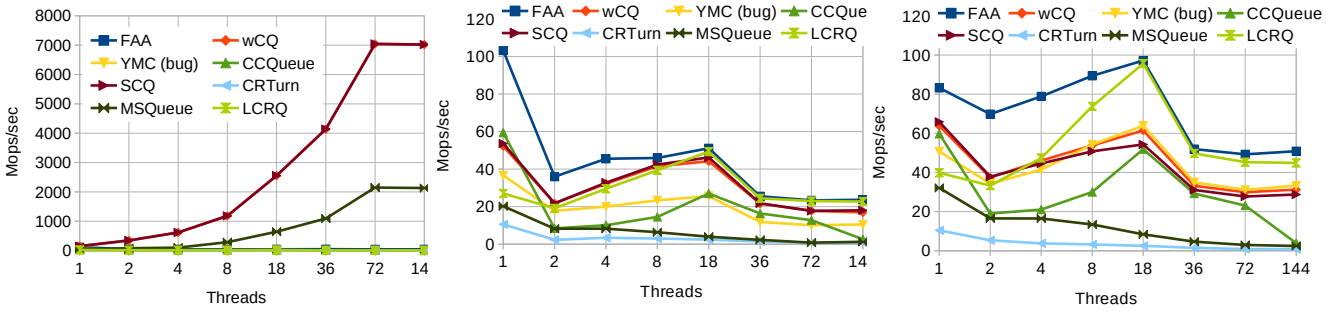
Our setup for all tests (except memory efficiency) is slightly different from [45], where authors always used tiny random delays between operations. This resulted in YMC's superior (relative) performance compared to LCRQ. However, it is unclear if such a setup reflects any specific practical scenario. Additionally, it degrades overall absolute throughput, whereas raw performance unhindered by any delays is often preferred. In our tests, contrary to [45], LCRQ is often on par or even superior to YMC, as it is also the case in [31].

As in [31], we first measured memory efficiency using standard malloc (this is merely to make sure that memory pages are unmapped more frequently, otherwise no fundamental difference in trends exists with jemalloc). This test additionally places tiny random delays between *Dequeue* and *Enqueue* operations, which we



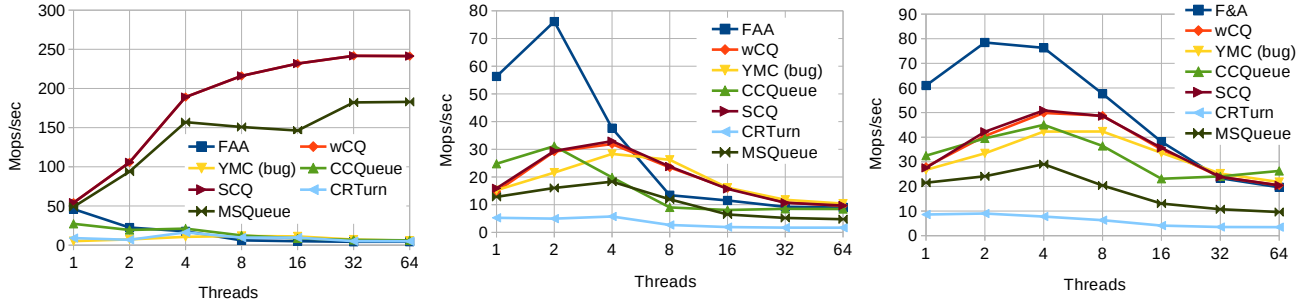
(a) Memory usage (b) Throughput

Figure 10: Memory test, x86-64 (Intel Xeon) architecture.



(a) Empty *Dequeue* throughput (b) Pairwise *Enqueue-Dequeue* (c) 50%/50% *Enqueue-Dequeue*

Figure 11: x86-64 (Intel Xeon) architecture.



(a) Empty *Dequeue* throughput (b) Pairwise *Enqueue-Dequeue* (c) 50%/50% *Enqueue-Dequeue*

Figure 12: PowerPC architecture.

empirically found to be helpful in amplifying memory efficiency artifacts. Operations are also chosen randomly, *Enqueue* for one half of the time, and *Dequeue* for the other half of the time. LCRQ appears to provide higher overall throughput, but its memory consumption grows very fast (≈ 380MB when the number of threads reaches 144). YMC's memory consumption also grows but slower (≈ 50MB). Finally, just like SCQ, wCQ is superior in this test. It only needs ≈ 1MB for the ring buffer as well as a negligible amount of memory for per-thread states. The reason for LCRQ's high memory consumption is discussed in [31]. The underlying ring buffers in these algorithms are livelock-prone and can prematurely be "closed." Each ring buffer, for better performance, needs to have at least 2^{12} entries, resulting in memory waste in certain cases.

Since performance on empty queues is known to vary [31, 45], we also measured *Dequeue* in a tight loop for an empty queue (Figures 11a and 12a). wCQ and SCQ both have superior performance due to their use of *Threshold* values. MSQueue also performs well, whereas other queues have inferior performance. In this test, FAA performs poorly since it still incurs cache invalidations due to RMW operations.

We then measured pairwise *Enqueue-Dequeue* operations by executing *Enqueue* followed by *Dequeue* in a tight loop. wCQ, SCQ,

and LCRQ have superior performance for x86-64 (Figure 11b). wCQ and SCQ both have superior performance for PowerPC (Figure 12b). In general, no significant performance difference is observed for wCQ vs. SCQ. YMC and other queues have worse performance.

We repeated the same experiment, but this time we selected *Enqueue* and *Dequeue* randomly. We executed *Enqueue* 50% of the time and *Dequeue* 50% of the time. wCQ's, SCQ's, and YMC's performance is very similar for x86-64 (Figure 11c). wCQ outperforms SCQ slightly due to reduced cache contention since wCQ's entries are larger. LCRQ typically outperforms all algorithms but can be vulnerable to increased memory consumption (Figure 10a shows this scenario). Moreover, LCRQ is only lock-free. The remaining queues have inferior performance. wCQ and SCQ outperform YMC and other queues for PowerPC (Figure 12c).

Overall, wCQ is the fastest wait-free queue. Its performance mostly matches the original SCQ algorithm, but wCQ enables wait-free progress. wCQ generally outperforms YMC, for which memory usage can be unbounded. LCRQ can sometimes yield better performance but it is only lock-free. LCRQ also sometimes suffers from high memory usage.

7 RELATED WORK

The literature presents a number of lock- and wait-free queues. Michael & Scott's FIFO queue [26] is a lock-free queue that keeps elements in a list of nodes. CAS is used to delete or insert new elements. Kogan & Petrank's wait-free queue [18] targets languages with garbage collectors such as Java. CRTurn [39] is a wait-free queue with built-in memory reclamation. Typically, none of these queues are high-performant. When feasible, certain batching lock-free approaches [27] can be used to alleviate the cost.

Though SimQueue [7] uses F&A and outperforms Michael & Scott's FIFO queue by aggregating concurrent operations with one CAS, it is still not that performant.

Ring buffers can be used for bounded queues. Typical ring buffers require CAS as in [20, 42]. Unfortunately, CAS scales poorly as the contention grows. Moreover, a number of approaches [10, 21, 43] are either not linearizable or not lock-free despite claims to the contrary, as discussed in [9, 24, 30]. Thus, designing high-performant, lock-free ring buffers remained a challenging problem.

Recently, a number of fast lock-free queues have been developed, which are inspired by ring buffers. LCRQ [30] implements high-performant, livelock-prone ring buffers that use F&A. To workaround livelocks, LCRQ links "stalled" ring buffers to Michael & Scott's lock-free list of ring buffers. This, however, may result in poor memory efficiency. SCQ [31], inspired by LCRQ, goes a step further by implementing a high-performant, lock-free ring buffer which can also be linked to Michael & Scott's lock-free list of ring buffers to create more memory efficient unbounded queues. Our proposed wCQ algorithm, inspired by SCQ, takes another step to devise a fully wait-free ring buffer. A slower queue (e.g., CRTurn) can be used as an outer layer for wCQ to implement unbounded queues. YMC [45], also partially inspired by LCRQ, attempted to create a wait-free queue. However, as discussed in [37], YMC is flawed in its memory reclamation approach, and is therefore not truly wait-free. Finally, LOO [11], another recent queue inspired by LCRQ, implements a specialized memory reclamation approach but is only lock-free.

Garbage collectors can alleviate reclamation problems but are not always practical, e.g., in C/C++. Furthermore, to our knowledge, there is no wait-free garbage collector with *limited* overheads and *bounded* memory usage. FreeAccess [3] and OrcGC [4] are efficient but they are only lock-free.

Finally, memory-boundness is an important goal in general, as evidenced by recent works. wCQ is close to the theoretical bound discussed in [1].

8 CONCLUSION

We presented wCQ, a fast wait-free FIFO queue. wCQ is the *first* high-performant wait-free queue for which memory usage is bounded. Prior approaches, such as YMC [45], also aimed at high performance, but failed to guarantee bounded memory usage. Strictly described, YMC is not wait-free as it is blocking when memory is exhausted.

Similar to SCQ's original lock-free design, wCQ uses F&A (fetch-and-add) for the most contended hot spots of the algorithm: *Head* and *Tail* pointers. Although native F&A is recommended, wCQ retains wait-freedom even on architectures without F&A, such as PowerPC or MIPS. wCQ requires double-width CAS to implement slow-path procedures correctly but can also be implemented on architectures with ordinary LL/SC (PowerPC or MIPS). Thus, wCQ largely retains SCQ's portability across different architectures.

Though Kogan-Petrank's method can be used to create wait-free queues with CAS [18], wCQ addresses unique challenges since it avoids dynamic allocation and uses F&A. We hope that wCQ will spur further research in creating *better performing* wait-free data structures with F&A.

Unbounded queues can be created by linking wCQs together, similarly to LCRQ or LSCQ, which use Michael & Scott's lock-free queue as an outer layer. The underlying algorithm need not be performant since new circular queues are only allocated very infrequently. Assuming that the underlying (non-performant) algorithm is truly wait-free with bounded memory usage such as CRTurn [37, 39], wCQ's complete benefits are retained even for unbounded queues.

Our slow_F&A idea can be adopted in other data structures that rely on F&A for improved performance.

AVAILABILITY

We provide wCQ's code at https://github.com/rusnikola/wfqueue.

An extended and up-to-date arXiv version of the paper is available at https://arxiv.org/abs/2201.02179.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their invaluable feedback.

A preliminary version of the algorithm previously appeared as a poster at PPoPP '22 [36].

This work is supported in part by AFOSR under grant FA9550-16-1-0371 and ONR under grants N00014-18-1-2022 and N00014-19-1-2493.

REFERENCES

[1] Vitaly Aksenov, Nikita Koval, and Petr Kuznetsov. 2022. Memory-Optimality for Non-Blocking Containers. https://arxiv.org/abs/2104.15003

[2] ARM. 2022. ARM Architecture Reference Manual. http://developer.arm.com/.

[3] Nachshon Cohen. 2018. Every Data Structure Deserves Lock-free Memory Reclamation. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 143 (Oct. 2018), 24 pages. https://doi.org/10.1145/3276513

[4] Andreia Correia, Pedro Ramalhete, and Pascal Felber. 2021. OrcGC: Automatic Lock-Free Memory Reclamation. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '21)*. ACM, 205–218. https://doi.org/10.1145/3437801.3441596

[5] DPDK Developers. 2022. Data Plane Development Kit (DPDK). https://dpdk.org/.

[6] Jason Evans. 2006. A scalable concurrent malloc(3) implementation for FreeBSD. In *Proceedings of the BSDCan Conference, Ottawa, Canada*. https://www.bsdcan.org/2006/papers/jemalloc.pdf

[7] Panagiota Fatourou and Nikolaos D. Kallimanis. 2011. A Highly-Efficient Wait-Free Universal Construction. In *Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures* (San Jose, California, USA) *(SPAA '11)*. ACM, New York, NY, USA, 325–334. https://doi.org/10.1145/1989493.1989549

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

[9] Steven Feldman and Damian Dechev. 2015. A Wait-free Multi-producer Multi-consumer Ring Buffer. *ACM SIGAPP Applied Computing Review* 15, 3 (Oct. 2015), 59–71. https://doi.org/10.1145/2835260.2835264

[10] Eric Freudenthal and Allan Gottlieb. 1991. Process Coordination with Fetch-and-increment. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems* (Santa Clara, California, USA) *(ASPLOS IV)*. 260–268. https://doi.org/10.1145/106972.106998

[11] Oliver Giersch and Jörg Nolte. 2022. Fast and Portable Concurrent FIFO Queues With Deterministic Memory Reclamation. *IEEE Trans. on Parallel and Distributed Systems* 33, 3 (2022), 604–616. https://doi.org/10.1109/TPDS.2021.3097901

[12] Danny Hendler, Nir Shavit, and Lena Yerushalmi. 2004. A Scalable Lock-free Stack Algorithm. In *Proceedings of the 16th ACM Symposium on Parallelism in Algorithms and Architectures* (Barcelona, Spain) *(SPAA '04)*. ACM, New York, NY, USA, 206–215. https://doi.org/10.1145/1007912.1007944

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

[14] IBM. 2005. PowerPC Architecture Book, Version 2.02. Book I: PowerPC User Instruction Set Architecture. http://www.ibm.com/developerworks/.

[15] Intel. 2022. Intel 64 and IA-32 Architectures Developer's Manual. http://www.intel.com/.

[16] Giorgos Kappes and Stergios V. Anastasiadis. 2021. POSTER: A Lock-Free Relaxed Concurrent Queue for Fast Work Distribution. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Virtual Event, Republic of Korea) *(PPoPP '21)*. ACM, New York, NY, USA, 454–456. https://doi.org/10.1145/3437801.3441583

[17] Christoph M. Kirsch, Michael Lippautz, and Hannes Payer. 2013. Fast and Scalable, Lock-Free k-FIFO Queues. In *Proceedings of the 12th International Conference on Parallel Computing Technologies - Volume 7979*. Springer-Verlag, Berlin, Heidelberg, 208–223. https://doi.org/10.1007/978-3-642-39958-9_18

[18] Alex Kogan and Erez Petrank. 2011. Wait-free Queues with Multiple Enqueuers and Dequeuers. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming* (San Antonio, TX, USA) *(PPoPP '11)*. ACM, New York, NY, USA, 223–234. https://doi.org/10.1145/1941553.1941585

[19] Alex Kogan and Erez Petrank. 2012. A Methodology for Creating Fast Wait-free Data Structures. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New Orleans, Louisiana, USA) *(PPoPP '12)*. ACM, New York, NY, USA, 141–150. https://doi.org/10.1145/2145816.2145835

[20] Nikita Koval and Vitaly Aksenov. 2020. POSTER: Restricted Memory-Friendly Lock-Free Bounded Queues. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, California) *(PPoPP '20)*. ACM, New York, NY, USA, 433–434. https://doi.org/10.1145/3332466.3374508

[21] Alexander Krizhanovsky. 2013. Lock-free Multi-producer Multi-consumer Queue on Ring Buffer. *Linux J.* 2013, 228, Article 4 (2013).

[22] Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.* 28, 9 (Sept. 1979), 690–691.

[23] Liblfds. 2022. Lock-free Data Structure Library. https://www.liblfds.org.

[24] Liblfds. 2022. Ringbuffer disappointment 2016-04-29. https://www.liblfds.org/slblog/index.html.

[25] Lockless Inc. 2022. Memory Allocator Benchmarks. https://locklessinc.com.

[26] Maged M. Michael and Michael L. Scott. 1998. Nonblocking Algorithms and Preemption-Safe Locking on Multiprogrammed Shared Memory Multiprocessors. *J. Parallel and Distrib. Comput.* 51, 1 (May 1998), 1–26. https://doi.org/10.1006/jpdc.1998.1446

[27] Gal Milman, Alex Kogan, Yossi Lev, Victor Luchangco, and Erez Petrank. 2018. BQ: A Lock-Free Queue with Batching. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures* (Vienna, Austria) *(SPAA '18)*. ACM, New York, NY, USA, 99–109. https://doi.org/10.1145/3210377.3210388

[28] MIPS. 2022. MIPS32/MIPS64 Rev. 6.06. http://www.mips.com/products/architectures/.

[29] Mark Moir, Daniel Nussbaum, Ori Shalev, and Nir Shavit. 2005. Using Elimination to Implement Scalable and Lock-free FIFO Queues. In *Proceedings of the 17th ACM Symposium on Parallelism in Algorithms and Architectures* (Las Vegas, Nevada, USA) *(SPAA '05)*. 253–262. https://doi.org/10.1145/1073970.1074013

[30] Adam Morrison and Yehuda Afek. 2013. Fast Concurrent Queues for x86 Processors. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Shenzhen, China) *(PPoPP '13)*. ACM, New York, NY, USA, 103–112. https://doi.org/10.1145/2442516.2442527

[31] Ruslan Nikolaev. 2019. A Scalable, Portable, and Memory-Efficient Lock-Free FIFO Queue. In *33rd International Symposium on Distributed Computing (DISC 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 146)*, Jukka Suomela (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 28:1–28:16. https://doi.org/10.4230/LIPIcs.DISC.2019.28

[32] Ruslan Nikolaev and Binoy Ravindran. 2019. Brief Announcement: Hyaline: Fast and Transparent Lock-Free Memory Reclamation. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (Toronto ON, Canada) *(PODC '19)*. ACM, New York, NY, USA, 419–421. https://doi.org/10.1145/3293611.3331575

[33] Ruslan Nikolaev and Binoy Ravindran. 2020. Universal Wait-Free Memory Reclamation. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, New York, NY, USA, 130–143. https://doi.org/10.1145/3332466.3374540

[34] Ruslan Nikolaev and Binoy Ravindran. 2021. Brief Announcement: Crystalline: Fast and Memory Efficient Wait-Free Reclamation. In *35th International Symposium on Distributed Computing (DISC 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 209)*, Seth Gilbert (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 60:1–60:4. https://doi.org/10.4230/LIPIcs.DISC.2021.60

[35] Ruslan Nikolaev and Binoy Ravindran. 2021. Snapshot-Free, Transparent, and Robust Memory Reclamation for Lock-Free Data Structures. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. ACM, New York, NY, USA, 987–1002. https://doi.org/10.1145/3453483.3454090

[36] Ruslan Nikolaev and Binoy Ravindran. 2022. POSTER: wCQ: A Fast Wait-Free Queue with Bounded Memory Usage. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Seoul, Republic of Korea) *(PPoPP '22)*. ACM, New York, NY, USA, 461–462. https://doi.org/10.1145/3503221.3508440

[37] Pedro Ramalhete and Andreia Correia. 2016. A Wait-Free Queue with Wait-Free Memory Reclamation (Complete Paper). https://github.com/pramalhe/ConcurrencyFreaks/blob/master/papers/crturnqueue-2016.pdf

[38] Pedro Ramalhete and Andreia Correia. 2017. Brief Announcement: Hazard Eras - Non-Blocking Memory Reclamation. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures* (Washington, DC, USA) *(SPAA '17)*. ACM, New York, NY, USA, 367–369. https://doi.org/10.1145/3087556.3087588

[39] Pedro Ramalhete and Andreia Correia. 2017. POSTER: A Wait-Free Queue with Wait-Free Memory Reclamation. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Austin, Texas, USA) *(PPoPP '17)*. ACM, New York, NY, USA, 453–454. https://doi.org/10.1145/3018743.3019022

[40] Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. 2012. Synchronising C/C++ and POWER. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. ACM, New York, NY, USA, 311–322. https://doi.org/10.1145/2254064.2254102

[41] SPDK Developers. 2022. Storage Performance Development Kit. https://spdk.io/.

[42] Philippas Tsigas and Yi Zhang. 2001. A Simple, Fast and Scalable Non-blocking Concurrent FIFO Queue for Shared Memory Multiprocessor Systems. In *Proceedings of the 13th ACM Symposium on Parallel Algorithms and Architectures* (Crete Island, Greece) *(SPAA '01)*. 134–143. https://doi.org/10.1145/378580.378611

[43] Dmitry Vyukov. 2022. Bounded MPMC queue. http://www.1024cores.net/home/lock-free-algorithms/queues/bounded-mpmc-queue.

[44] Haosen Wen, Joseph Izraelevitz, Wentao Cai, H. Alan Beadle, and Michael L. Scott. 2018. Interval-Based Memory Reclamation. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Vienna, Austria) *(PPoPP '18)*. ACM, New York, NY, USA, 1–13. https://doi.org/10.1145/3178487.3178488

[45] Chaoran Yang and John Mellor-Crummey. 2016. 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* (Barcelona, Spain) *(PPoPP '16)*. ACM, New York, NY, USA, Article 16, 13 pages. https://doi.org/10.1145/2851141.2851168

```
1  wCQ * LHead = <empty wCQ>, LTail = LHead;
2  void Enqueue_Unbounded(void * p)
3    wCQ * ltail = hp.protectPtr(HPTail, Load(&LTail));
     // Enqueue_Ptr() returns false if full or finalized
4    if ( !ltail->next and ltail->Enqueue_Ptr(p, finalize=True) )
5      hp.clear();
6      return;
7    wCQ * cq = alloc_wCQ();                                  // Allocate wCQ
8    cq->init_wCQ(p);                                       // Initialize & put p
9    enqueuers[TID] = cq;                                 // == Slow path (CRTurn) ==
10   for i = 0 .. NUM_THRDS-1 do
11     if ( enqueuers[TID] = null )
12       hp.clear();
13       return;
14     wCQ * ltail = hp.protectPtr(HPTail, Load(&LTail));
15     if ( ltail ≠ Load(&LTail) ) continue;
16     if ( enqueuers[ltail->enqTid] = ltail )
17       CAS(&enqueuers[ltail->enqTid], ltail, null);
18     for j = 1 .. NUM_THRDS do
19       cq = enqueuers[(j + ltail->enqTid) mod NUM_THRDS];
20       if ( cq = null ) continue;
21       finalize_wCQ(ltail);                  // Duplicate finalize is OK since
22       CAS(&ltail->next, null, cq);              // cq or another node follows
23       break;
24     wCQ * lnext = Load(&ltail->next);
25     if ( lnext ≠ null )
26       finalize_wCQ(ltail);                  // Duplicate finalize is OK since
27       CAS(&LTail, ltail, lnext);             // lnext or another node follows
28   enqueuers[TID] = null;
29   hp.clear();
30 bool dequeue_rollback(wCQ * prReq, wCQ * myReq)
31   deqself[TID] = prReq;
32   giveUp(myReq, TID);
33   if ( deqhelp[TID] ≠ myReq )
34     deqself[TID] = myReq;
35     return False;
36   hp.clear();
37   return True;

38 void finalize_wCQ(wCQ * cq)
39   OR(&cq->Tail, { .Value=0, .Finalize=1 });
40 void * Dequeue_Unbounded()
41   wCQ * lhead = hp.protectPtr(HPHead, Load(&LHead));
     // skip_last modifies the default behavior for Dequeue on
     // the last element in aq, as described in the text.
42   void * p = lhead->Dequeue_Ptr(skip_last=True);
43   if ( p ≠ last )
44     if ( p ≠ null or lhead->next = null )
45       hp.clear();
46       return p;
47   wCQ * prReq = deqself[TID];                  // == Slow path (CRTurn) ==
48   wCQ * myReq = deqhelp[TID];
49   deqself[TID] = myReq;
50   for i = 0 .. NUM_THRDS-1 do
51     if ( deqhelp[TID] != myReq ) break;
52     wCQ * lhead = hp.protectPtr(HPHead, Load(&LHead));
53     if ( lhead ≠ Load(&LHead) ) continue;
54     void * p = lhead->Dequeue_Ptr(skip_last=True);
55     if ( p ≠ last )
56       if ( p ≠ null or lhead->next = null )
57         if ( !dequeue_rollback(prReq, myReq) ) break;
58         return p;
59       Store(&lhead->aq.Threshold, 3n − 1);
60       p = lhead->Dequeue_Ptr(skip_last=True);
61       if ( p ≠ last and p ≠ null )
62         if ( !dequeue_rollback(prReq, myReq) ) break;
63         return p;
64     wCQ * lnext = hp.protectPtr(HPNext, Load(&lhead->next));
65     if ( lhead ≠ Load(&LHead) ) continue;
66     if ( searchNext(lhead, lnext) ≠ NOIDX ) casDeqAndHead(lhead, lnext);
67   wCQ * myCQ = deqhelp[TID];
68   wCQ * lhead = hp.protectPtr(HPHead, Load(&LHead));
69   if ( lhead = Load(&LHead) and myCQ = Load(&lhead->next) )
70     CAS(&LHead, lhead, myCQ);
71   hp.clear();
72   hp.retire(prReq);
73   return myCQ->Locate_Last_Ptr();
```

Figure 13: Adapting CRTurn to an unbounded wCQ-based queue design (high-level methods).

A APPENDIX: UNBOUNDED QUEUE

LSCQ and LCRQ implement unbounded queues by using an outer layer of M&S lock-free queue which links ring buffers together. Since operations on the outer layer are very rare, the cost is dominated by ring buffer operations. wCQ can follow the same idea.

Although the outer layer does not have to be performant, it still must be wait-free with bounded memory usage. However, M&S queue is only lock-free. The (non-performant) CRTurn wait-free queue [37, 39] does satisfy the aforementioned requirements. Moreover, CRTurn already implements wait-free memory reclamation by using hazard pointers in a special way. wCQ and CRTurn combined together would yield a fast queue with bounded memory usage.

Because CRTurn's design is non-trivial and is completely orthogonal to the wCQ presentation, its discussion is beyond the scope of this paper. We refer the reader to [37, 39] for more details about CRTurn. Below, we sketch expected changes to CRTurn (assuming prior knowledge of CRTurn) to link ring buffers rather than individual nodes. In Figure 13, we present pseudocode (assuming that entries are pointers) with corresponding changes to enqueue and dequeue operations in CRTurn. For convenience, we retain the same variable and function names as in [37] (e.g., *giveUp* that is not shown here). Similar to [37], we assume memory reclamation API based on hazard pointers (the *hp* object).

The high-level idea is that we create a wait-free queue (list) of wait-free ring buffers, where *LHead* and *LTail* represent corresponding head and tail of the list. *Enqueue_Unbounded* will first attempt to insert an entry to the last ring buffers as long as *LTail* is already pointing to the last ring buffer. Otherwise, it allocates a new ring buffer and inserts a new element. It then follows CRTurn's procedure to insert the ring buffer to the list. The only difference is that when helping to insert the new ring buffer, threads will make sure that the previous ring buffer is finalized.

Dequeue_Unbounded will first attempt to fetch an element from the first ring buffer. wCQ's Dequeue for **aq** needs to be modified to detect the very last entry in a *finalized* ring buffer. (Note that it can only be done for finalized ring buffers, where no subsequent entries can be inserted.) Instead of returning the true entry, *Dequeue_Ptr* returns a special **last** value. This approach helps to retain CRTurn's wait-freedom properties as every single ring buffer contains at least one entry. Helper methods must also be modified accordingly.