

POSTER: 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, existing wait-free queues are either not very scalable or suffer from potentially unbounded memory usage. We present a wait-free queue, wCQ, which uses its own variation of the fast-path-slow-path methodology to attain wait-freedom and bound memory usage. wCQ is memory efficient and its 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

1 Introduction

Wait-free data structures require that *all* threads complete any operation after a finite number of steps. Wait-free algorithms have evolved over the years, and they have increasingly gained more attention due to their strongest non-blocking progress property.

Creating efficient FIFO queues, let alone wait-free ones, is notoriously hard [4]. 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 [3, 4, 7] 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. Only recently, SCQ [4] implemented a fast non-blocking ring buffer via 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 [2] 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. To that end, Yang and Mellor-Crummey's (YMC) [7] wait-free queue implemented its own fast-path-slow-path method. But, as pointed out by Ramalheite and Correia [6], YMC's design is flawed in its memory reclamation approach which, strictly described, forfeits wait-freedom. Thus, a user still has to choose from other wait-free queues which do not use F&A and are slower, e.g., Kogan & Petrank's [1] queue.

We present a wait-free circular queue (wCQ) which extends SCQ by using its own fast-path-slow-path method. wCQ uses double-width CAS, available on x86 and AArch64.

2 Algorithm Descriptions

wCQ's key insight is to avoid memory reclamation altogether. Since wCQ only allocates 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.

Figure 1 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 SCQ). *Enqueue_wCQ* then takes the slow path, where it requests help by recording its last *Tail* value that was tried (in *initTail* and *localTail*) and the *index* input parameter. *initTail* and *localTail* are initially identical but diverge later.

A somewhat similar procedure is used for *Dequeue_wCQ*, which additionally checks if the queue is empty. After completing the slow path, the output result needs to be gathered.

*Most of the work was done while the author was at Virginia Tech.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

PPoPP '22, April 2–6, 2022, Seoul, Republic of Korea

© 2022 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-9204-4/22/04.

<https://doi.org/10.1145/3503221.3508440>

```

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,  $\perp$  });
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();
14   // == Fast path (SCQ) ==
15   int count = MAX_PATIENCE;
16   while --count  $\neq$  0 do
17     tail = try_enq(index);
18     if ( tail = OK ) return;
19   // == Slow path (wCQ) ==
20   thrdrec_t *r = &Record[TID];
21   int seq = r->seq1;
22   r->localTail = tail;
23   r->initTail = tail;
24   r->index = index;
25   r->enqueue = true;
26   r->seq2 = seq;
27   r->pending = true;
28   enqueue_slow(tail, index, r);
29   r->pending = false;
30   r->seq1 = seq + 1;
31   int Dequeue_wCQ()
32   if ( Load(&Threshold) < 0 )
33     return  $\emptyset$ ; // Empty
34   help_threads();
35   // == Fast path (SCQ) ==
36   int count = MAX_PATIENCE;
37   while --count  $\neq$  0 do
38     int idx;
39     head = try_deq(&idx);
40     if ( head = OK ) return idx;
41   // == Slow path (wCQ) ==
42   thrdrec_t *r = &Record[TID];
43   int seq = r->seq1;
44   r->localHead = head;
45   r->enqueue = false;
46   r->seq2 = seq;
47   r->pending = true;
48   dequeue_slow(head, r);
49   r->pending = false;
50   r->seq1 = seq + 1;
51   // Get slow-path results
52   h = Counter(r->localHead);
53   j = Cache_Remap(h mod 2n);
54   Ent = Load(&Entry[j].Value);
55   if ( Ent.Cycle = Cycle(h) and
56       Ent.Index  $\neq$   $\perp$  )
57     consume(h, j, Ent);
58   return Ent.Index; // Done
59   return  $\emptyset$ 

```

Figure 1. Wait-free circular queue (wCQ).

In SCQ, the output is merely *consumed* by using atomic OR (i.e., Line 3 only). In wCQ, we additionally mark all pending enqueueers (Line 2). A special bit, *Enq*, is used internally for the slow path to support a two-step insertion [5].

help_threads circularly iterates across all threads and loads a request, which is passed to *enqueue_slow/dequeue_slow*.

wCQ's key idea for the slow path 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 must repeat *exactly* the same procedure as the helpee. This can be challenging since the ring buffer keeps changing.

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. We introduce a special *slow_F&A* operation, which 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. 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. (Pointers are initially **null**.) Fast path procedures use F&A on counters leaving pointers intact. However, slow path procedures use the pointer component to store the second phase request [5].

To retain SCQ's 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 (*slow_F&A*) across all cooperative threads. We decrement *Threshold* prior to the actual dequeue attempt.

3 Evaluation

Our evaluation [5] shows that wCQ is the fastest wait-free queue; its performance is close to the SCQ algorithm. wCQ generally outperforms YMC, for which memory usage can be unbounded. Certain lock-free algorithms (e.g., LCRQ) can yield better performance but they lack wait-freedom.

4 Conclusion

We presented wCQ, the *first* high-performant wait-free queue for which memory usage is bounded. Similar to SCQ's lock-free design, wCQ uses F&A for the most contended hot spots of the algorithm: *Head* and *Tail* pointers. Kogan-Petrank's method can be used for wait-free queues with CAS [1], but wCQ had to design its own method to support F&A and avoid dynamic allocation. We hope that wCQ's method will spur further research in creating wait-free data structures.

We thank the anonymous reviewers for their invaluable feedback. 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. Complete details of the algorithm, analysis, and evaluation are available in [5]. We provide wCQ's code at <https://github.com/rusnikola/wfqueue>.

References

- [1] Alex Kogan and Erez Petrank. 2011. Wait-free Queues with Multiple Enqueueers and Dequeueers. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP '11)*. 223–234.
- [2] Alex Kogan and Erez Petrank. 2012. A Methodology for Creating Fast Wait-free Data Structures. In *Proceedings of the 17th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP '12)*. 141–150.
- [3] 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 (PPoPP '13)*. 103–112.
- [4] Ruslan Nikolaev. 2019. A Scalable, Portable, and Memory-Efficient Lock-Free FIFO Queue. In *Proceedings of the 33rd International Symposium on Distributed Computing (DISC 2019)*, Vol. 146. 28:1–28:16.
- [5] Ruslan Nikolaev and Binoy Ravindran. 2022. wCQ: A Fast Wait-Free Queue with Bounded Memory Usage (full paper, arXiv). <https://arxiv.org/abs/2201.02179>
- [6] Pedro Ramalhete and Andreia Correia. 2016. A Wait-Free Queue with Wait-Free Memory Reclamation. <https://github.com/pramalhe/ConcurrencyFreaks/blob/master/papers/crturqueue-2016.pdf>
- [7] Chaoran Yang and John Mellor-Crummey. 2016. A Wait-free Queue As Fast As Fetch-and-add. In *Proceedings of the 21st ACM Symposium on Principles and Practice of Parallel Programming (PPoPP '16)*. 16:1–16:13.