A Scalable, Portable, and Memory-Efficient Lock-Free FIFO Queue

Ruslan Nikolaev
Virginia Tech, USA
rnikola@vt.edu

Abstract

We present a new lock-free multiple-producer and multiple-consumer (MPMC) FIFO queue design which is scalable and, unlike existing high-performant queues, very memory efficient. Moreover, the design is ABA safe and does not require any external memory allocators or safe memory reclamation techniques, typically needed by other scalable designs. In fact, this queue itself can be leveraged for object allocation and reclamation, as in data pools. We use FAA (fetch-and-add), a specialized and more scalable than CAS (compare-and-set) instruction, on the most contended hot spots of the algorithm. However, unlike prior attempts with FAA, our queue is both lock-free and linearizable.

We propose a general approach, SCQ, for bounded queues. This approach can easily be extended to support unbounded FIFO queues which can store an arbitrary number of elements. SCQ is portable across virtually all existing architectures and flexible enough for a wide variety of uses. We measure the performance of our algorithm on the x86-64 and PowerPC architectures. Our evaluation validates that our queue has exceptional memory efficiency compared to other algorithms and its performance is often comparable to, or exceeding that of state-of-the-art scalable algorithms.

2012 ACM Subject Classification Theory of computation → Concurrent algorithms

Keywords and phrases FIFO, queue, ring buffer, lock-free, non-blocking

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.28

Related Version A full version of the paper is available at `https://arxiv.org/abs/1908.04511`

1 Introduction

Creating efficient concurrent algorithms for modern multi-core architectures is challenging. Efficient and scalable lock-free FIFO queues proved to be especially hard to devise. Although elimination techniques address LIFO stack performance [6], their FIFO counterparts [18] are somewhat more restricted: a *dequeue* operation can eliminate an *enqueue* operation only if all preceding queue entries have already been consumed. Thus, FIFO elimination is more suitable in specialized cases and, typically, shorter queues. Relaxed versions of FIFO queues, which can reorder elements, were also proposed [9], but they cannot be used when a strict FIFO ordering is required. FIFO queues are important in many applications, especially in fixed-size data pools which use bounded *circular queues* (*ring buffers*).

Most correct linearizable and lock-free implementations of such queues rely on the compare-and-set (CAS) instruction. Although this instruction is powerful, it does not scale well as the contention grows. However, less powerful instructions such as fetch-and-add (FAA) are known to scale better. In Figure 1, we demonstrate execution time for FAA vs. CAS measured in a tight loop for the corresponding number of threads. The results are for an almost "ideal case" to simply emulate FAA in a CAS loop. As CAS loops in typical algorithms are more complex, the actual gap is bigger.

FAA may seem to be a natural fit for ring buffers when updating queue's head and tail pointers. The reality, however, is more nuanced when designing lock-free queues. Many straight-forward algorithms without explicit locks that use FAA, e.g., [10], are actually not lock-free [4] because it is possible to find an execution pattern where no thread can make

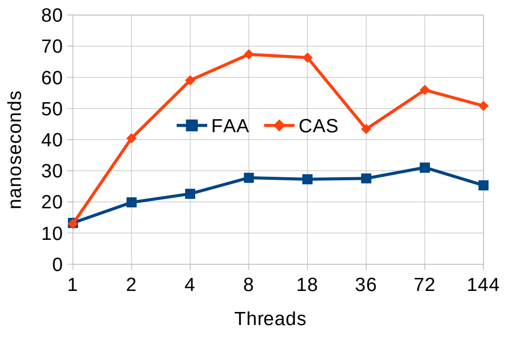


Figure 1 FAA vs. CAS on 4x18 Xeon E7-8880.

```
int Tail = 0, Head = 0;  // Queue's tail and head
void * Array[∞];                // An infinite array
void enqueue(void * p)
    while True do
        T = FAA(&Tail, 1);
        // Repeat the loop if the entry is
        // already invalidated by dequeue()
        if ( SWAP(&Array[T], p) = ⊥ )
            return;

void * dequeue()
    while True do
        H = FAA(&Head, 1);
        p = SWAP(&Array[H], ⊤);
        if ( p ≠ ⊥ ) return p;
        if ( Load(&Tail) ≤ H + 1 )
            return nullptr;              // Empty
```

Figure 2 Infinite array queue (susceptible to livelocks).

progress. Lack of true lock-freedom manifests in suboptimal performance when some threads are preempted since other threads are effectively blocked when the preemption happens in the middle of a queue operation. Additionally, such queues cannot be safely used in environments where blocking is not permitted. Even if FAA is not used, certain queues [23] fail to achieve linearizable lock-free behavior. A case in point: an open source lock-free data structure library, liblfds [12], simply falls back [13] to the widely known Michael & Scott's (M&S) FIFO lock-free queue [16] in their ring buffer implementation. While easy to implement, this queue does not scale well as we show in Section 7.

Despite the aforementioned challenges, the use of FAA is re-invigorated by recent concurrent FIFO queue designs [24, 19]. Unfortunately, [24, 19], despite their good performance, are not always memory efficient as we demonstrate in Section 7. Furthermore, such queues rely on memory allocators and safe memory reclamation schemes, thus creating a "chicken and egg" situation when allocating memory blocks. For example, if we simply want to recycle memory blocks or create a queue-based memory pool for allocation, reliance on an external memory allocator to allocate and deallocate memory is undesirable. Furthermore, typical system memory allocators, including jemalloc [2], are not lock-free. Lock-based allocators can defeat the purpose of creating a purely lock-free algorithm since they weaken overall progress guarantees. Moreover, in a number of use cases, such as within OS kernels, blocking can be prohibited or undesirable.

Contributions of the paper

- We introduce an approach of building ring buffers using indirection and two queues.
- We present our *scalable circular queue* (SCQ) design which uses FAA. To the best of our knowledge, it is the first ABA-safe design that is scalable, livelock-free and relies only on single-width atomic operations. It is inspired by CRQ [19] but prevents livelocks and uses our indirection approach. Although CRQ attempts to solve a similar problem, it is livelock-prone, uses double-width CAS (unavailable on PowerPC [7], MIPS [17], RISC-V [21], SPARC [20], and other architectures) and is not standalone; it uses M&S queue as an extra layer (LCRQ) to work around livelock situations.
- Since unbounded queues are also used widely, we present the LSCQ design (Section 5.3) which chains SCQ ring buffers in a list. LSCQ is more memory efficient than LCRQ.

2 Background

Lock-free algorithms

We consider an algorithm lock-free if at least one thread can make progress in a finite number of steps. In other words, individual threads may starve, but a preempted thread must not block other threads from making progress. In contrast, spin locks (either implicit, found in some incorrect algorithms, or explicit) will prevent other threads from making further progress if the thread holding the lock is scheduled out.

Atomic primitives

CAS (compare-and-set) is used universally by most lock-free algorithms. However, one downside of CAS is that it can fail, especially under large contention. Although specialized instructions such as FAA (fetch-and-add) and SWAP do not reduce memory contention directly, they are more efficiently implemented by hardware and never fail. FAA and SWAP are currently implemented by x86-64 [8], ARMv8.1+ [1], and RISC-V [21].

Safe memory reclamation (SMR)

Most non-trivial lock-free algorithms, including queues, require special treatment of memory blocks that need to be deallocated, as concurrent threads may still access memory referred to by pointers retrieved prior to the change in a corresponding lock-free data structure. For programming languages such as C/C++, where unmanaged code is prevalent, lock-free *safe memory reclamation* techniques such as hazard pointers [15] are used. The main high-level idea is that each accessed pointer must be protected by a corresponding API call. When done, the thread's pointer reservation can be reset. When SMR knows that a memory block can be returned safely to the OS, it triggers memory deallocation. Bounded SCQ does not need SMR, but certain other lock-free queues such as LCRQ rely on SMR by their design.

Infinite array queue

Figure 2 shows an infinite array queue, originally described for the LCRQ design [19]. This queue is susceptible to livelocks, but our infinite array queue as well as the SCQ design are inspired by it. Initially, the queue is empty, i.e., all its entries are set to a special $\perp$ value. *enqueue* uses FAA on `Tail` to retrieve a slot which will be used to place a new entry. An enqueuer will try to use this slot. However, if the previous value is not $\perp$, some dequeuer already modified it, and this enqueuer moves on to the next slot. *dequeue* uses FAA on `Head` to retrieve a slot which contains a previously produced entry. A dequeuer will insert another special value, $\top$, to indicate that the slot can no longer be used. If the previous value is not $\perp$, a corresponding enqueuer has already produced some entry, which is taken by this dequeuer. Otherwise, this dequeuer moves on to the next slot. A corresponding enqueuer, which arrives later, will be unable to use this slot.

3 Preliminaries

Assumptions

We assume that a program has k threads that can run on any number of physical CPU cores. For the purpose of this work, we will assume that the maximum (bounded) queue size is n. As no thread should block on another thread, we will further reasonably assume that $k \leq n$.

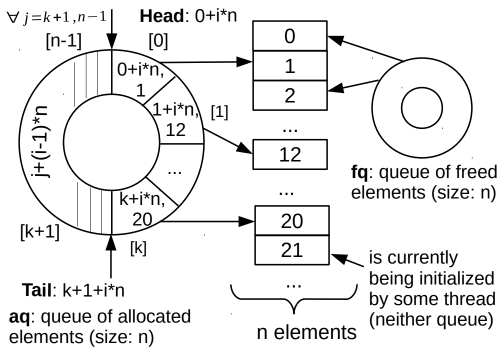

$\forall j = k+1, n-1$
Head: 0+i*n
[n-1]
[0]
0+i*n,
1
1+i*n,
12
[1]
...
k+i*n,
20
j+(i-1)*n
[k+1]
[k]
Tail: k+1+i*n
aq: queue of allocated elements (size: n)
0
1
2
...
12
...
20
21
...
n elements
fq: queue of freed elements (size: n)
is currently being initialized by some thread (neither queue)

Figure 3 Proposed data structure.

```
// data:  an array of pointers
// aq is initialized empty
// fq is initialized full
bool enqueue_ptr(void * ptr)
    int index = fq.dequeue();
    if ( index = ∅ ) return False;          // Full
    data[index] = ptr;
    aq.enqueue(index);
    return True;                          // Success

void * dequeue_ptr()
    int index = aq.dequeue();
    if ( index = ∅ ) return nullptr;      // Empty
    ptr = data[index];
    fq.enqueue(index);
    return ptr;                           // Success
```

Figure 4 Example: storing pointers.

For simplicity of our presentation, we will assume that the system memory model is sequentially consistent [11]. However, the actual implementations of the algorithms (including implementations used in Section 7) can rely on weaker memory models whenever possible.

Data structure

Our design is based on two key ideas. First, we use indirection, i.e., data entries are not stored in the queue itself. Instead, a queue entry simply records an index into the array of data. Second, we maintain two queues: **fq**, which keeps indices to unallocated entries of the array, and **aq**, which keeps allocated indices to be consumed. A producer thread dequeues an index from **fq**, writes data to the corresponding array entry, and inserts the index into **aq**. A consumer thread dequeues the index from **aq**, reads data from the array, and inserts the entry back into **fq**.

Both queues maintain `Head` and `Tail` references (Figure 3). They are incremented when new entries are enqueued (`Tail`) or dequeued (`Head`). At any point, these references can be represented as $j + i \times n$, where j is an *index* (position in the ring buffer), i is a *cycle*, and n is a ring buffer size (must be power of 2 in our implementation). For example, for `Head`, we can calculate index $j = (Head \mod n)$ and cycle $i = (Head \div n)$.

Queue entries mirror `Head` and `Tail` values. Each entry also records an index into the array, pointing to the data associated with the entry. Unlike `Head` and `Tail`, it suffices to just record *cycle* i, as entry positions are redundant. We instead record an index into the array that is of the same bit-length as the position.

ABA safety

The ABA problem is prevented by comparing cycles. As both `Head` and `Tail` are incremented sequentially, regardless of queue size, they will not wrap around until after the number of operations exceeds the CPU word's largest value, a reasonable assumption made by other ABA-safe designs as well.

Data entries and pointers

Data array entries can be of any type and size. It is not uncommon for programs to use a pair of queues with recyclable elements, e.g., queues analogous to **aq** and **fq**. In this case, a program can simply use indices instead of pointers. It is also possible to simply store

```
 1 int Tail = n, Head = n;              // Initialization
 2 forall entry_t Ent ∈ Entries[n] do
 3     Ent = { .Cycle: 0, .Index: 0 };
 4 void enqueue(int index)
 5     do
 6         T = Load(&Tail);
 7         j = Cache_Remap(T mod n);
 8         Ent = Load(&Entries[j]);
 9         if ( Cycle(Ent) = Cycle(T) )
10             CAS(&Tail, T, T + 1);      // Help to
11             goto 6;                    // move tail
12         if ( Cycle(Ent) + 1 ≠ Cycle(T) )
13             goto 6;         // T is already stale
14         New = { Cycle(T), index };
15     while !CAS(&Entries[j], Ent, New);
16     CAS(&Tail, T, T+1);     // Try to move tail

17 int dequeue()
18     do
19         H = Load(&Head);
20         j = Cache_Remap(H mod n);
21         Ent = Load(&Entries[j]);
22         if ( Cycle(Ent) ≠ Cycle(H) )
23             if ( Cycle(Ent) + 1 = Cycle(H) )
24                 return ∅;       // Empty queue
25             goto 19;      // H is already stale
26     while !CAS(&Head, H, H+1);
27     return Index(Ent);
```

Figure 5 Naive circular queue (NCQ).

(arbitrary) pointers as data entries. We can build a FIFO queue with data pointers using **aq** and **fq** queues as shown in Figure 4. A producer thread dequeues an entry from **fq**, initializes it with a pointer and inserts the entry into **aq**. A consumer thread dequeues the entry from **aq**, reads the pointer and inserts the entry back into **fq**. Note that *enqueue* does not need to check if a queue is full. It is only called when an available entry (out of n) exists (e.g., can be dequeued from **fq** if enqueueing to **aq**, or vice versa).

4 Naive Circular Queue (NCQ)

Let us first consider a simple algorithm, NCQ, which uses the presented data structure but borrows an idea of moving queue's tail on behalf of another thread from M&S queue [16]. It achieves performance similar to M&S queue but does not need double-width CAS to avoid the ABA problem. We use this queue as an extra baseline in Section 7.

Figure 5 shows the *enqueue* and *dequeue* operations. In the algorithm, we assume ordinary unsigned integer ring arithmetic when calculating cycles. Empty queues initialize all entries to cycle 0. Their `Head` and `Tail` are both n (cycle 1). Full queues initialize all entries to cycle 0 along with allocated entry indices. Their `Head` is 0 (cycle 0) and `Tail` is n (cycle 1).

Entries are always updated sequentially. To reduce contention due to false sharing, we remap queue entry positions by using a simple permutation function, *Cache_Remap*, that places two adjacent entries into different cache lines. The function remaps entries such that the same cache line will not be reused in the ring buffer as long as possible.

When dequeuing, we verify that `Head`'s cycle number matches the cycle number of the entry `Head` is pointing to. If `Head` is one cycle ahead, the queue is empty. Any other mismatches (i.e., an entry is ahead) imply that a producer has recycled this entry already. Therefore, other threads must have already consumed the entry and incremented `Head` since then (i.e., the previously loaded `Head` value is stale).

As discussed in Section 3, enqueueing is only possible when an available entry exists. To successfully enqueue an entry, `Tail` must be one cycle ahead of an entry it currently points to. `Tail`'s cycle number equals the entry's cycle number if another thread has already inserted an element but has not yet advanced `Tail`. The current thread helps to advance `Tail` to facilitate global progress. All other cycle mismatches (i.e., an entry is ahead) imply that the fetched `Tail` value is stale, as there has been at least an entire round (n enqueues) since it was last loaded.

```
// Threshold prevents livelocks
signed int Threshold = -1;            // Empty queue
void enqueue(int index)
    while True do
        T = FAA(&Tail, 1);
        if ( SWAP(&Entries[T], index) = ⊥ )
            Store(&Threshold, 2n − 1);
            return;

int dequeue()
    if ( Load(&Threshold) < 0 ) return ∅;
    while True do
        H = FAA(&Head, 1);
        index = SWAP(&Entries[H], ⊤);
        if ( index ≠ ⊥ ) return index;
        if ( FAA(&Threshold, -1) ≤ 0 )
            return ∅
        if ( Load(&Tail) ≤ H + 1 ) return ∅;
```

Figure 6 Infinite array queue. (We make it livelock-free by using a "threshold".)

5 Scalable Circular Queue (SCQ)

We will now consider a more elaborated design. Our scalable circular queue (SCQ) is partially inspired by CRQ [19] as well as by our data structure (Section 3). The major problem with CRQ is that it is not standalone due to its inherent susceptibility to livelocks. CRQ must be coupled with a truly lock-free FIFO queue (such as M&S queue [16]). If a livelock happens while enqueueing entries, a slow path is taken, where the current CRQ instance is "closed". Then a new CRQ instance is allocated and used to enqueue new entries. This approach replaces CRQ with a list of CRQs (LCRQ). As discussed in introduction, this design has to rely on a memory allocator and memory reclamation scheme.

SCQ is not only standalone and livelock-free, but also much more portable across different CPU architectures. Unlike CRQ that requires a special double-width CAS instruction, our SCQ algorithm only needs single-width CAS, available across virtually all modern architectures. SCQ enables support for PowerPC [7] where CRQ/LCRQ cannot be implemented [24, 19]. Similarly, SCQ enables support for MIPS [17], SPARC [20], and RISC-V [21] which, like PowerPC, do not support double-width CAS.

5.1 Infinite array queue

We start off with the presentation of our infinite array queue. We diverge from the original idea (Section 2) in two major ways.

First, we present a solution to livelocks caused by dequeuers by introducing a special "threshold" value that we describe below. Livelocks occur when dequeuers incessantly invalidate slots that enqueuers are about to use for their new entries. By using the threshold value, we do not carry over the livelock problem to the practical implementation as in case of CRQ, i.e., guarantee that at least one enqueuer as well as one dequeuer both succeed after a finite number of steps at any point of time. Algorithms with this property were previously called *operation-wise* lock-free [19]. This term represents a stronger version of lock-freedom.

Second, our queue reflects the design presented in Section 3, where we use indices into the array of data rather than pointers. This approach guarantees that *enqueue* always succeeds (neither **aq** nor **fq** ever ends up with more than n elements). Rather, both "full" and "empty" conditions are detected by the corresponding *dequeue* operation as in Figure 4. Consequently, *enqueue* does not need to be treated specially to detect full queues.

We note that a circular queue accommodates only a finite number of elements, n. Furthermore, per assumptions in Section 3, the number of concurrent enqueuers or dequeuers never exceeds n ($k \leq n$). We apply these restrictions to our infinite queue and present a modified algorithm in Figure 6.

Suppose that *enqueue* successfully inserts some entry into the queue and sets the threshold on Line 6 prior to completion. Let us consider the very last inserted entry for which the threshold is set. (The threshold can also be set by any concurrent *enqueue* for preceding entries if their Line 6 executes after last enqueuer's Line 5.) We will justify the threshold value later. If dequeuers are active at that moment, we have either of the two scenarios:

The last dequeuer is not ahead of the inserted entry. In this case, the last dequeuer is no farther than n slots to the left of the inserted entry (Figure 7). This is because we have at most n available slots which can be referenced by any concurrent enqueuers. None of the enqueuers in this region (i.e., after the last dequeuer) can fail because the corresponding slots are not being invalidated by the dequeuers. (Note that this argument is only applicable to the infinite array queue. We will further refine it for SCQ below.)

The last dequeuer gets ahead of the inserted entry (either initially, or in the process of dequeuing). Any preceding concurrent dequeuer either succeeds when its entry can be consumed (Line 13), or fails and retries. Since `Head` increases monotonically (Line 11), if a failed dequeuer ever retries, its new position can only be after the current position of the last dequeuer. However, since the inserted entry is the very last for which *enqueue* is complete (Line 6), all of the dequeuers located after the last dequeuer (inclusively) are doomed to fail unless some other concurrent enqueuer completes. (In the latter case, we recursively go back to the very beginning of our argument where we have chosen the last inserted entry.)

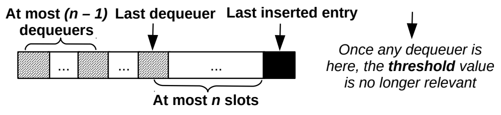

At most *(n – 1)* dequeuers
Last dequeuer
Last inserted entry
...
At most *n* slots
Once any dequeuer is here, the **threshold** value is no longer relevant

Figure 7 Threshold bound for livelock prevention.

In the second scenario, we are not concerned about the threshold value. In other words, Lines 14-15 that terminate dequeuers after the inserted entry do not cause any problems. All these dequeuers are guaranteed to fail one way or the other. Some preceding dequeuer, which is already in progress, will eventually fetch the inserted entry. However, in the first scenario, we want to make sure that Line 14 does not prematurely terminate dequeuers that are still trying to reach the inserted entry. Specifically, any failed attempt is penalized by decreasing the threshold value (Line 14).

To reach the last inserted entry, the last dequeuer, or any dequeuer that follows it later, will unsuccessfully traverse at most n slots (Figure 7). At the moment when the entry is inserted, we may also have up to $n-1$ dequeuers (since $k \leq n$) that are lagging behind (any distance away). When they fail, they are penalized by subtracting the threshold value. When they retry, the are guaranteed to be after the last dequeuer (Line 11). Thus, to guarantee that a dequeuer eventually reaches the inserted entry, the threshold must be $2n-1$.

Note that the threshold approach carefully avoids memory contention on the fast path in *dequeue*. Only *enqueue* typically updates this value through an ordinary memory write with a barrier. Moreover, if the threshold is still intact, it does not need to be updated.

5.2 SCQ algorithm

In Figure 8, we present our SCQ algorithm. It is based on the modified infinite array queue and (partially) CRQ [19]. It manages cycles differently in *dequeue*, making it possible to leverage a simpler atomic OR operation instead of CAS.

Every entry in the SCQ buffer consists of the *Cycle* and *Index* components. We also reserve one bit in each entry, *IsSafe*, that we describe below. This bit is similar to the

```
1  int Tail = 2n, Head = 2n;                       // Empty queue
2  signed int Threshold = -1;
3  forall entry_t Ent ∈ Entries[2n] do
4      Ent = { .Cycle=0, .IsSafe=1, .Index=$\perp$ };
5  void catchup(int tail, int head)                // Internal
6      while !CAS(&Tail, tail, head) do
7          head = Load(&Head);
8          tail = Load(&Tail);
9          if ( tail $\geq$ head )
10             break;
11 void enqueue(int index)
12     while True do
13         T = FAA(&Tail, 1);
14         j = Cache_Remap(T mod 2n);
15         Ent = Load(&Entries[j]);
16         if ( Cycle(Ent) < Cycle(T) and
             Index(Ent) = $\perp$ and
             (IsSafe(Ent) or Load(&Head) $\leq$ T) )
17             New = { Cycle(T), 1, index };
18             if ( !CAS(&Entries[j], Ent, New) )
19                 goto 15
20             if ( Load(&Threshold) $\neq 3n - 1$ )
21                 Store(&Threshold, $3n - 1$)
22             return;
23 int dequeue()
24     if ( Load(&Threshold) < 0 )                 // Check if
25         return $\varnothing$;                      // the queue is empty
26     while True do
27         H = FAA(&Head, 1);
28         j = Cache_Remap(H mod 2n);
29         Ent = Load(&Entries[j]);
30         if ( Cycle(Ent) = Cycle(H) )
               // Cycle can't change, mark as $\perp$
31             Atomic_OR(&Entries[j], { 0, 0, $\perp$ });
32             return Index(Ent);                  // Done
33         New = { Cycle(Ent), 0, Index(Ent) };
34         if ( Index(Ent) = $\perp$ )
35             New = { Cycle(H), IsSafe(Ent), $\perp$};
36         if ( Cycle(Ent) < Cycle(H) )
37             if ( !CAS(&Entries[j], Ent, New) )
38                 goto 29
39         T = Load(&Tail);                        // Check if
40         if ( T $\leq$ H + 1 ) // the queue is empty
41             catchup(T, H + 1);
42             FAA(&Threshold, -1);
43             return $\varnothing$;
44         if ( FAA(&Threshold, -1) $\leq$ 0 )
45             return $\varnothing$
```

Figure 8 Scalable circular queue (SCQ).

corresponding bit in CRQ.

SCQ leverages the high-level idea from the infinite array queue described above. However, SCQ replaces SWAP operations with CAS, as memory buffers are finite, and the same slot can be referenced by multiple cycles.

When discussing the threshold value, we previously assumed that no enqueuer can fail when all dequeuers are behind. In SCQ, this is no longer true, as the same entry can be occupied by some previous cycle. To bound the maximum distance between the last dequeuer and the last enqueuer, we ***double the capacity of the queue*** while still keeping the original number of elements. When using this approach, all the enqueuers after the last dequeuer can always locate an unused ($\perp$) slot no farther than $2n$ slots away from it. The threshold value should now become $(n - 1 + 2n) = 3n - 1$.

In the algorithm, we need a special value for $\perp$. We reserve the very last index, $2n - 1$, for this purpose. Since, in SCQ, n is a power of 2 number, $\perp$ will have all its index bits set to 1. As we show below, this allows *dequeue* to consume entries using an atomic OR operation. This value does not overlap with the actual data indices, which are still less than n.

SCQ also accounts for additional corner cases that are not present in the infinite queue:

A dequeuer arrives prior to its enqueuer counterpart, but the corresponding entry is already occupied. This happens when the entry is occupied by some other cycle. If this cycle is already newer (Line 36 is false), *dequeue* simply retries because the enqueuer counterpart is guaranteed to fail (Line 16). However, if the cycle is older, *dequeue* needs to mark it accordingly, so that when the enqueuer counterpart arrives, it will fail. For this purpose, we clear the *IsSafe* bit, as in CRQ. The key idea is that the enqueuer will have to additionally make sure that all active dequeuers are behind when *IsSafe* is set to 0 (Line 16) before inserting a new entry. Whenever *IsSafe* becomes 0, only enqueuers can change it back to 1. (Only Line 17 sets the bit to 1; Line 35 preserves bit's value, and Line 33 sets it to 0.)

Enqueuers attempt to use slots that are marked unsafe. If *IsSafe* on Line 16

```
void * ListHead = <empty SCQ>;
void * ListTail = ListHead;

void finalize_SCQ(SCQ * cq)
    Atomic_OR(&cq.Tail, {.Value=0, .Finalize=1});

void * dequeue_unbounded()
    while True do
        SCQ * cq = Load(&ListHead);
        void * p = cq.dequeue_ptr();
        if ( p ≠ nullptr ) return p;
        if ( cq.next = nullptr ) return nullptr;
        Store(&cq.aq.Threshold, $3n - 1$);
        p = cq.dequeue_ptr();
        if ( p ≠ nullptr ) return p;
        if ( CAS(&ListHead, cq, cq.next) )
            free_SCQ(cq);        // Dispose of cq

void enqueue_unbounded(void * p)
    while True do
        SCQ * cq = Load(&ListTail);
        if ( cq.next ≠ nullptr )
            CAS(&ListTail, cq, cq.next);
            continue;            // Move list tail
        // Finalizes & returns false if full
        if ( cq.enqueue_ptr(p, finalize=True) )
            return;
        ncq = alloc_SCQ();       // Allocate ncq
        ncq.init_SCQ(p);  // Initialize & put p
        if ( CAS(&cq.next, nullptr, ncq) )
            CAS(&ListTail, cq, ncq);
            return;
        free_SCQ(ncq);           // Dispose of ncq
```

Figure 9 Unbounded SCQ-based queue (LSCQ).

is 0, an enqueuer will additionally make sure that the dequeuer that needs to be accounted for has not yet started by reading and comparing the current `Head` value.

When dequeuing elements, if cycles match (Line 30), a dequeuer is guaranteed to succeed. The corresponding slot will not be recycled until an entry is consumed. The only thing that can change is the *IsSafe* bit. Unlike CRQ, to mark an entry as consumed, *dequeue* issues an atomic OR operation which sets all index bits to 1 while preserving entry's safe bit and cycle.

The *catchup* procedure is similar to the *fixState* procedure from CRQ and is used when the tail is behind the head. This allows to avoid unnecessary iterations in *enqueue* and reduces the risk of contention.

Finally, when comparing cycles, we use a common approach with signed integer arithmetic which takes care of potential wraparounds.

Optimization

Similarly to LCRQ, SCQ employs an additional optimization on dequeuers. If a dequeuer arrives prior to the corresponding enqueuer, it will not aggressively invalidate a slot. Instead, it will spin for a small number of iterations with the expectation that the enqueuer arrives soon. This alleviates unnecessary contention on the head and tail pointers, and consequently helps both dequeuers and enqueuers.

5.3 SCQ-based unbounded queue (LSCQ)

We follow LCRQ's main idea of maintaining a list of ring buffers in our LSCQ design. LSCQ is potentially more memory efficient than LCRQ, as it is based on livelock-free SCQs which do not end up being prematurely "closed" (Section 7). Since operations on the list are very rare, the cost is completely dominated by SCQ operations.

In Figure 9, we present the LSCQ algorithm. We intentionally ignore the memory reclamation problem (Section 2) which can be straight-forwardly solved by the corresponding techniques such as hazard pointers [15]. The presented unbounded queue can store any fixed-size data entries, including pointers (as in Figure 9), just like SCQ itself. When a ring buffer is full, we need to additionally "finalize" it, so that no new entries are inserted by the concurrent threads. As in CRQ, we reserve one bit in `Tail`. When **fq** does not have available entries (Line 3, Figure 4), we set the corresponding bit for **aq**'s `Tail`.

```
bool enqueue_ptr(void * ptr)
    // Add a full queue check before Line 12:
    T = Load(&Tail);
    if ( T ≥ Load(&Head) + 2n ) return False;
    ... Modified enqueue() ...
    // Add a full queue check in the loop after Line 22:
    if ( T+1 ≥ Load(&Head) + 2n ) return False;
```

Figure 10 SCQ for double-width CAS: checking for full queues.

We also modify *enqueue* for **aq** such that it fails when FAA on `Tail` returns a value with the "finalized" bit set. Thus, any concurrent thread that tries to insert entries after **aq** is being finalized, fails. In this case, we also need to place the entry back into **fq**. This cannot fail since **fq** is never finalized.

Before unlinking *cq* from the list (Line 14), *dequeue_unbounded* checks again that *cq*, which must already be finalized, is empty. Pending enqueuers may still access it. For the final check, the threshold must be reset so that slots for the pending enqueuers can be invalidated.

5.4 SCQ for double-width CAS

The x86-64 [8] and ARM64 [1] architectures implement double-width CAS, which atomically updates two contiguous words. We can leverage this capability to build SCQ which avoids indirection when storing arbitrary pointers. In this case, all entries consist of tuples. Each tuple stores two adjacent integers instead of just one integer. The *index* field of the first integer now simply indicates if the entry is occupied (0) or available ($\perp$). The second integer from the same tuple stores a pointer which is used in lieu of an index. Since pointers also need to be stored and retrieved, we change Lines 18, 31, and 37 to use double-width CAS accordingly.

This version of SCQ provides a fully compatible API such that architectures without double-width CAS can still implement the same queue through indirection. In this queue, *enqueue* becomes *enqueue_ptr*, and *dequeue* becomes *dequeue_ptr*.

If *enqueue_ptr* needs to identify full queues, additional changes are required. In Figure 10, we show a method which compares `Head` and `Tail` values. The comparison is relaxed and up to k ($k \leq n$) concurrent enqueuers can increment `Tail` spuriously. Thus, `Tail` can now be up to $3n$ slots ahead of `Head`. Since we previously assumed that number to be $2n$, we increase the threshold from $3n-1$ to $4n-1$. This method is imprecise and can only guarantee that at least n elements are in the queue, but the actual number varies. This is often acceptable, especially when creating unbounded queues (Section 5.3), which finalize full queues.

6 Correctness

We omit more formal linearizability arguments for NCQ due to its simplicity. SCQ's linearizability follows from the arguments we make in Sections 5.1 and 5.2, as well as from the corresponding CRQ linearizability derivations [19] because the SCQ design has many similarities with CRQ. Below we provide lock-freedom arguments for NCQ and SCQ.

▶ **Theorem 1.** *The NCQ algorithm is lock-free.*

Proof. The NCQ algorithm has two unbounded loops: one in *enqueue* (Lines 5-15) and the other one in *dequeue* (Lines 18-26).

If CAS fails in *enqueue* causing it to repeat, it means that the corresponding entry `Entries[j]` is changed by another thread executing *enqueue*, as *dequeue* does not modify entries. Consequently, that other thread is making progress, i.e., succeeding in the *enqueue* operation (Line 15).

If CAS fails in *dequeue* causing it to repeat, it means that the `Head` pointer is modified by another thread executing *dequeue*. Therefore, that other thread is making progress, i.e., succeeding in the *dequeue* operation (Line 26). ◀

▶ **Theorem 2.** *The SCQ algorithm is lock-free.*

Proof. The SCQ algorithm has two unbounded loops: one in *enqueue* (Lines 12-22) and the other one in *dequeue* (Lines 26-45).

If the condition on Line 16 of *enqueue* is false causing it to repeat the loop, then two possibilities exist. First, some dequeuer already invalidated the slot for this enqueuer (*Cycle(Ent)* and *IsSafe(Ent)* checks) because it arrived before the enqueuer. Alternatively, the entry is occupied by a prior enqueuer (i.e., $\neq \perp$) and is supposed to be consumed by some dequeuer which is yet to come.

In the latter case, the current enqueuer skips the occupied slot. There may also exist other concurrent enqueuers which will skip occupied slots as well. Since the total number of elements for *enqueue* is always capped, the queue will never have more than n entries. Thus, the enqueuers should be able to succeed unless dequeuers keep invalidating their slots.

The first case is more intricate. If none of the enqueuers succeed, dequeuers must not be able to invalidate new slots after a finite number of steps. Once dequeuers stop invalidating slots (using the threshold described in Section 5.1), at least one enqueuer can make further progress by catching up its `Tail` to the next available position and inserting a new element.

If the conditions on Lines 30, 40, or 44 of *dequeue* are false causing it to repeat the loop, then the following must be the reason for that. Line 30 can only be false if the entry is not yet initialized by the corresponding enqueuer. In this case, the dequeuer may potentially iterate and invalidate slots as many as $3n$ times until either Line 43 or 45 terminates the loop. Line 45 is guaranteed to eventually terminate the loop as long as no new entries are inserted by *enqueue* (i.e., enqueuers can be running, but none of them succeed). Any pending (almost completed) enqueuer may still cause Line 21 to increase the threshold value temporarily even though its entry was already consumed. However, this at most is going to happen for $k - 1$ pending enqueuers. After that, the threshold value will be depleted causing all active dequeuers to complete (Line 45). Since dequeuers are no longer running, at least one new enqueuer will be able to succeed. At that point, it will reset the threshold value (Line 21). After that, we recursively repeat this entire argument again to show that at least one following enqueuer will succeed in a finite number of steps. ◀

7 Evaluation

In this section, we evaluate our SCQ design against well-known or state-of-the-art algorithms. We use and extend the benchmark from [24] which already implements several algorithms.

In the evaluation, we show that SCQ achieves very high performance while avoiding limitations that are typical to other high-performant algorithms (i.e., livelock workarounds, memory reclamation, and portability). We have also found that state-of-the-art approaches, especially LCRQ, can have very high memory usage, a problem that does not exist in SCQ.

We present results for both bare-bones SCQ and the version that stores arbitrary pointers (SCQP). Bare-bones SCQ is relevant because queue elements in SCQ can be of any type, i.e., not necessarily pointers. We compare SCQ and SCQP against M&S FIFO lock-free queue (MSQUEUE) [16], combining queue (CCQUEUE) [3] – which is not a lock-free queue but is known to have good performance, LCRQ [19] – a queue that maintains a lock-free list of ring buffers (CRQ); CRQs cannot be used separately, as they are susceptible to livelocks, WFQUEUE – a recent scalable wait-free queue design [24]. These algorithms provide reasonable baselines as they represent well-known or state-of-the-art (in scalability) approaches. We also present NCQ as an additional baseline. NCQ uses the same data structure as SCQ, but its design is reminiscent of MSQUEUE. Finally, we provide FAA (fetch-and-add) throughputs to show the potential for scalability. FAA is not a real algorithm, it simply implements atomic increments on `Head` and `Tail` when calling *dequeue* and *enqueue* respectively. We skip a separate LSCQ evaluation since SCQ is already lock-free and can be used as is. LSCQ's costs are largely dominated by the underlying SCQ implementation.

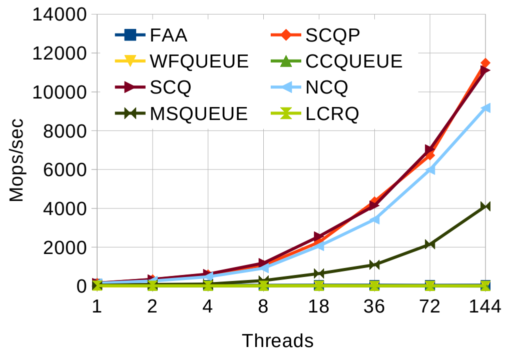
Performance differences between queues should be treated with great caution. For example, LCRQ has better throughput in a few tests, but it consumes a lot of memory under certain circumstances. Also, neither LCRQ nor WFQUEUE are deployable in all places where SCQ can be used, e.g., fixed-size data pools. As mentioned in introduction, this would trigger a "chicken and egg" situation, where data pools would need to depend on some external (typically non lock-free) memory allocator. Moreover, both LCRQ and WFQUEUE require safe memory reclamation by their design; the benchmark implements a specialized scheme for WFQUEUE and hazard pointers [15] for LCRQ and MSQUEUE. Finally, WFQUEUE needs special per-thread descriptors, which reduce the API transparency.

We run all experiments for up to 144 threads on a 72-core machine consisting of four Intel Xeon E7-8880 v3 CPUs with hyper-threading disabled, each running at 2.30 GHz and with a 45MB L3 cache. The machine has 128GB of RAM and runs Ubuntu 16.04 LTS. We use gcc 8.3 with the -O3 optimization flag. For SCQP, we use the double-width version (Section 5.4) as x86-64 can benefit from it. SCQP is compiled with clang 7.0.1 (-O3) because it generates faster code for our implementation (no such advantage for other queues).

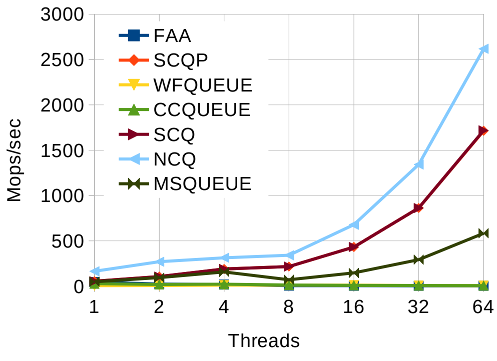
We also evaluate queues on the PowerPC architecture. We run experiments on an 8-core POWER8 machine. Each core has 8 threads, so we have 64 logical cores in total. We do not disable PowerPC's simultaneous multithreading because the machine does not have as many cores as our Xeon testbed. All cores are running at 3.0 Ghz and have an 8MB L3 cache. The machine has 64GB of RAM and runs Ubuntu 16.04 LTS. We use gcc 8.3 with the -O3 optimization flag. Since PowerPC does not support double-width CAS, it is impossible to implement the LCRQ algorithm there. In contrast, SCQ and SCQP both work well on PowerPC. For SCQP, we use the version with indirection and two queues.

We use jemalloc [2] to alleviate libc's malloc poor performance [14]. Each data point is measured 10 times for 10000000 operations in a loop, we present the average. The benchmark measures throughput in a steady/hot state and protects against occasional outliers. We use the default benchmark parameters from [24] to achieve optimal performance with LCRQ, CCQUEUE, and WFQUEUE. For SCQ and NCQ, we have chosen a relatively small ring buffer size, 2^{16} entries. (SCQ uses only half queue's capacity, $n = 2^{15}$ entries, as discussed in Section 5.2.) LCRQ uses 2^{12} entries in each CRQ to attain optimal performance. Unlike SCQ, LCRQ wastes a lot of memory in each CRQ due to cache-line padding. Most of our results for x86-64 have peaks for 18 threads because each CPU has 18 cores. Over-socket contention is expensive and results in performance drops. PowerPC has an analogous picture.

In Figure 11, we perform a simple experiment to measure the cost of *dequeue* on empty queues. MSQUEUE, NCQ, SCQ, and SCQP perform reasonably well on both x86-64 (Figure 11a) and PowerPC (Figure 11b) since they do not dequeue elements aggressively.



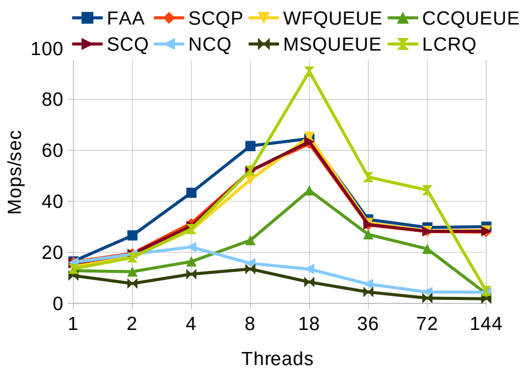
(a) 4x18-core Intel Xeon E7-8880



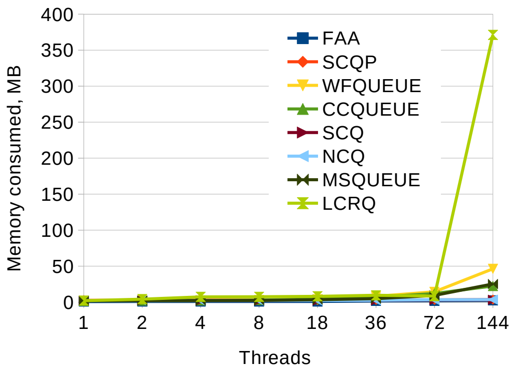
(b) 8x8-core POWER8

Figure 11 Empty queue test, throughput of the dequeue operation.

LCRQ, WFQUEUE, and CCQUEUE take a performance hit in this corner case. FAA is also slower than MSQUEUE, NCQ, SCQ, and SCQP because it still needs to modify an atomic variable. Slow dequeuing on empty queues was previously acknowledged by WFQUEUE's authors [24].



(a) Throughput (higher is better)

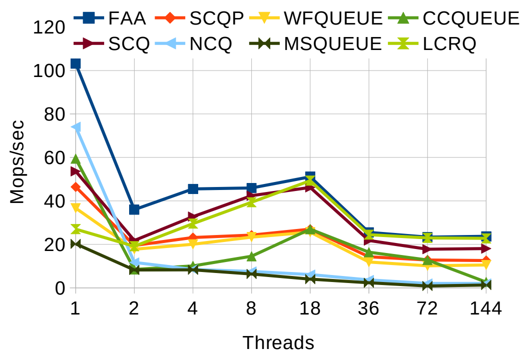


(b) Memory consumption (lower is better)

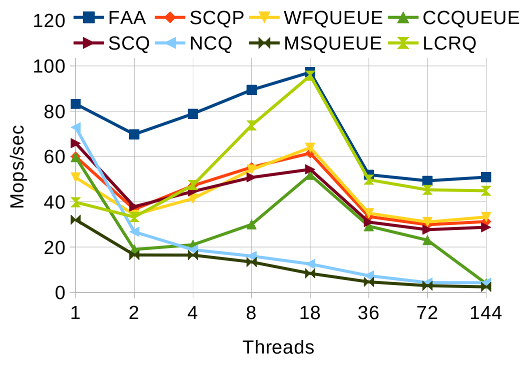
Figure 12 Memory efficiency test, 4x18-core Intel Xeon E7-8880 (standard malloc).

To evaluate memory efficiency, we run an experiment with 50% of *enqueue* and 50% of *dequeue* operations that are chosen by each thread randomly (Figure 12). For this test, we use libc's standard malloc to make sure that memory pages are unmapped more aggressively. We run the benchmark with its default configuration that uses tiny delays between operations. Using delays allows us to get more pronounced results while still showing a realistic execution scenario. It turns out that while, for the most part, LCRQ provides higher throughput (Figure 12a), it can also allocate a lot of memory while running (Figure 12b), up to ≈ 400MB. WFQUEUE's memory usage is also somewhat elevated (up to ≈ 50MB) and exceeds that of MSQUEUE and CCQUEUE for most data points. Conversely, SCQ, SCQP, and NCQ are very efficient; they only need a small (512K-1MB), fixed-size buffer that is allocated for circular queues. Overall, SCQ and SCQP win here as they both achieve great performance with very little memory overhead.

Since the design of SCQ is related to LCRQ, we were particularly interested in investigating LCRQ's high memory usage. As we suspected, LCRQ was "closing" CRQs frequently due to livelocks. To maintain good performance, LCRQ must use relatively large CRQs (each of them has 2^{12} entries). However, due to livelocks, CRQs need to be prematurely closed from time to time. Eventually, LCRQ ends up in a situation where it frequently allocates new CRQs, i.e., wasting memory greatly. Safe memory reclamation additionally impacts the timing of deallocation, i.e., CRQs are not deallocated immediately.



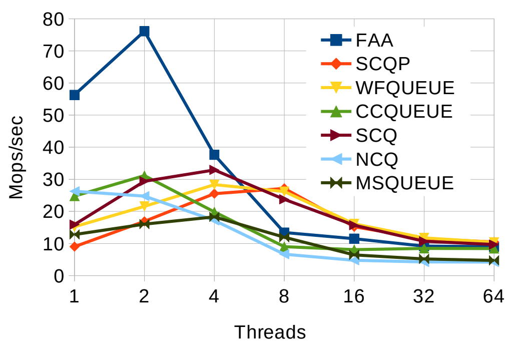
(a) enqueue-dequeue pairs



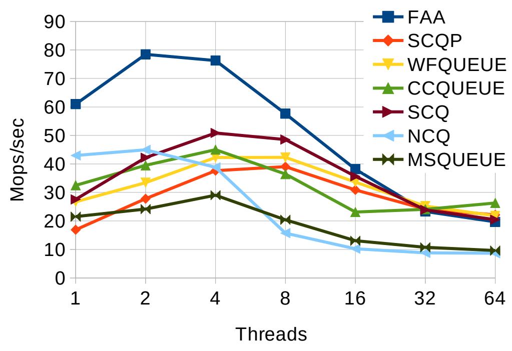
(b) 50% enqueues, 50% dequeues

Figure 13 Balanced load tests, 4x18-core Intel Xeon E7-8880.

Finally, we evaluate queues using operations by multiple threads in a tight loop. In Figures 13a and 14a, we present throughput for the x86-64 and PowerPC architectures respectively when using pairwise queue operations. In this experiment, every thread executes *enqueue* followed by *dequeue* in a tight loop. Since multiple concurrent threads are running simultaneously, the order of dequeued elements is not predetermined anyhow (even though enqueue and dequeue are paired). For x86-64, SCQ and LCRQ are both winners and have roughly similar performance. SCQP, WFQUEUE, and CCQUEUE (partially) attain half of their throughput on average. For PowerPC, SCQ is a winner: it generally outperforms all other algorithms. SCQP and WFQUEUE are roughly identical, except that WFQUEUE marginally outperforms SCQP for smaller concurrencies. CCQUEUE is generally worse, except very small concurrencies.



(a) enqueue-dequeue pairs



(b) 50% enqueues, 50% dequeues

Figure 14 Balanced load tests, 8x8-core POWER8.

In Figures 13b and 14b, we present results for an experiment which selects operations randomly: 50% of enqueues and 50% of dequeues. For x86-64, WFQUEUE and SCQP are almost identical. SCQP marginally outperforms SCQ when concurrency is high, probably due to (occasional) cache contention since entries are larger in double-width SCQP. CCQUEUE is typically slower than WFQUEUE, SCQP, or SCQ. LCRQ outperforms all of them most of the time. However, considering its memory utilization, LCRQ may not be appropriate in a number of cases as previously discussed. For PowerPC, SCQ is a winner: it generally outperforms all other algorithms. SCQP, WFQUEUE, and CCQUEUE are very close; WFQUEUE marginally outperforms SCQP in this test.

8 Related Work

Over the last couple of decades, different designs were proposed for concurrent FIFO queues as well as ring buffers. A classical Michael & Scott's lock-free FIFO queue [16] maintains a list of nodes. The list has the *head* and *tail* pointers. These pointers must be updated using CAS operations. The queue can be modified to avoid the ABA problem related to pointer updates. For that purpose, double-width CAS is used to store an ABA tag for each pointer.

Existing lock-free ring buffer designs that do not benefit from FAA (e.g., [22]) are typically not very scalable. Another approach [4], though uses FAA, needs a memory reclamation scheme and does not seem to scale as well as some other algorithms. Certain queues use FAA but are not linearizable. For example, [5] maintains a queue size and updates it with FAA. However, the queue may end up in inconsistent state as previously discussed in [19].

Certain bounded MPMC queues without explicit locks such as [10, 23] are relatively fast. However, these approaches are technically not lock-free as discussed in [4, 13]. Just as with explicit spin locks, lack of true lock-freedom manifests in suboptimal performance when threads are preempted, as remaining threads are effectively blocked when the preemption happens in the middle of a queue operation.

Alternative concurrent designs were also considered. CCQUEUE [3] is a special combining queue. The reasoning behind this queue is that it may be cheaper to execute operations sequentially. When performing an operation, a thread is added to a list; the thread at the head of the list completes operations on behalf of other threads from the list.

Finally, the use of FAA for queues is advocated by recent non-blocking FIFO queue designs [24, 19]. Unfortunately, [24, 19], despite their good performance, are not always memory efficient.

9 Conclusion

In this paper, we presented SCQ, a scalable lock-free FIFO queue. The main advantage of SCQ is that the queue is standalone, memory efficient, ABA safe, and scalable. At the same time, SCQ does not require a safe memory reclamation scheme or memory allocator. In fact, this queue itself can be leveraged for object allocation and reclamation, as in data pools.

We use FAA (fetch-and-add) for the most contended hot spots of the algorithm: `Head` and `Tail` pointers. Unlike prior attempts to build scalable queues with FAA such as CRQ, our queue is both lock-free and linearizable. SCQ prevents livelocks directly in the ring buffer itself without trying to work around the problem by allocating additional ring buffers and linking them together. SCQ is very portable and can be implemented virtually everywhere. It only needs single-width CAS. It is also possible to create unbounded queues based on SCQs which are more memory efficient than LCRQ.

We thank the anonymous reviewers for their valuable feedback.

SCQ's source code is available at `https://github.com/rusnikola/lfqueue`.

References

1 Arm Limited. ARM Architecture Reference Manual. `http://developer.arm.com/`, 2019.

2 Jason Evans. A scalable concurrent malloc(3) implementation for FreeBSD. In *Proceedings of the BSDCan Conference, Ottawa, Canada*, 2006.

3 Panagiota Fatourou and Nikolaos D. Kallimanis. Revisiting the Combining Synchronization Technique. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 257–266, New York, NY, USA, 2012. ACM.

4 Steven Feldman and Damian Dechev. A Wait-free Multi-producer Multi-consumer Ring Buffer. *ACM SIGAPP Applied Computing Review*, 15(3):59–71, October 2015.

5 Eric Freudenthal and Allan Gottlieb. Process Coordination with Fetch-and-increment. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, pages 260–268, 1991.

6 Danny Hendler, Nir Shavit, and Lena Yerushalmi. A Scalable Lock-free Stack Algorithm. In *Proceedings of the 16th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, pages 206–215, New York, NY, USA, 2004. ACM.

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

8 Intel. Intel 64 and IA-32 Architectures Developer's Manual. `http://www.intel.com/`, 2019.

9 Christoph M. Kirsch, Michael Lippautz, and Hannes Payer. Fast and Scalable, Lock-Free k-FIFO Queues. In *Proceedings of the 12th International Conference on Parallel Computing Technologies - Volume 7979*, pages 208–223, Berlin, Heidelberg, 2013. Springer-Verlag.

10 Alexander Krizhanovsky. Lock-free Multi-producer Multi-consumer Queue on Ring Buffer. *Linux J.*, 2013(228), 2013. URL: `http://dl.acm.org/citation.cfm?id=2492102.2492106`.

11 Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.

12 Liblfds. Lock-free Data Structure Library. `http://www.liblfds.org`.

13 Liblfds. Ringbuffer disappointment. `http://www.liblfds.org/wordpress/index.php/2016/04/29/ringbuffer-disappointment/`.

14 Lockless Inc. Memory Allocator Benchmarks. `https://locklessinc.com/benchmarks_allocator.shtml`, 2019.

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

16 Maged M. Michael and Michael L. Scott. Nonblocking Algorithms and Preemption-Safe Locking on Multiprogrammed Shared Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 51(1):1–26, May 1998.

17 MIPS. MIPS32/MIPS64 Rev. 6.06. `http://www.mips.com/products/architectures/`, 2019.

18 Mark Moir, Daniel Nussbaum, Ori Shalev, and Nir Shavit. Using Elimination to Implement Scalable and Lock-free FIFO Queues. In *Proceedings of the 17th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '05, pages 253–262, 2005.

19 Adam Morrison and Yehuda Afek. Fast Concurrent Queues for x86 Processors. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 103–112, New York, NY, USA, 2013. ACM.

20 Oracle. SPARC Architecture 2011. `http://www.oracle.com/`, 2019.

21 RISC-V Foundation. RISC-V Books. `http://riscv.org/risc-v-books/`, 2019.

22 Philippas Tsigas and Yi Zhang. 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*, SPAA '01, pages 134–143, 2001.

23 Dmitry Vyukov. Bounded MPMC queue. `http://www.1024cores.net/home/lock-free-algorithms/queues/bounded-mpmc-queue`.

24 Chaoran Yang and John Mellor-Crummey. A Wait-free Queue As Fast As Fetch-and-add. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, pages 16:1–16:13, New York, NY, USA, 2016. ACM.