The State-of-the-Art LCRQ Concurrent Queue Algorithm Does NOT Require CAS2

Raed Romanov
Higher School of Economics
St. Petersburg, Russia
raid_r@mail.ru

Nikita Koval
JetBrains
Amsterdam, The Netherlands
ndkoval@ya.ru

Abstract

Concurrent queues are, arguably, one of the most important data structures in high-load applications, which require them to be extremely fast and scalable. Achieving these properties is non-trivial. The early solutions, such as the classic queue by Michael and Scott, store elements in a concurrent linked list. Reputedly, this design is non-scalable and memory-inefficient. Modern solutions utilize the `Fetch-and-Add` instruction to improve the algorithm's scalability and store elements in arrays to reduce the memory pressure. One of the most famous and fast such algorithms is LCRQ. The main disadvantage of its design is that it relies on the atomic `CAS2` instruction, which is unavailable in most modern programming languages, such as Java, Kotlin, or Go, let alone some architectures.

This paper presents the LPRQ algorithm, a *portable* modification of the original LCRQ design that eliminates all `CAS2` usages. In contrast, it performs the synchronization utilizing only the standard `Compare-and-Swap` and `Fetch-and-Add` atomic instructions. Our experiments show that LPRQ provides the same performance as the classic LCRQ algorithm, outrunning the fastest of the existing solutions that do not use `CAS2` by up to `1.6`×.

CCS Concepts: • **Computing methodologies** → **Concurrent algorithms**; *Shared memory algorithms.*

Keywords: concurrent queue, ring buffer, lock-free

1 Introduction

Queues are one of the most fundamental and practical data structures in concurrent programming. A tremendous amount of research has been gained into developing fast and efficient solutions. The first *lock-free* algorithm was suggested by Michael and Scott almost three decades ago [16], starting the era of building efficient non-blocking data structures. Roughly, their design bases on a linked list structure, maintained via `Head` and `Tail` pointers; `Enqueue(..)` and `Dequeue()` update them via atomic `Compare-and-Set (CAS)` instruction. Recently, this design has been changed drastically, utilizing the `Fetch-and-Add (FAA)` instruction to improve the algorithm's scalability and storing elements in arrays to reduce the memory pressure. One of the most famous algorithms of this family is `LCRQ` [18], suggested by Morrison and Afek in 2013.

Infinite array queue. The LCRQ design was inspired by a straightforward queue algorithm, which manipulates an infinite array with positioning counters for `Enqueue(..)` and `Dequeue()` operations. To add a new element, `Enqueue(..)` increments the `Tail` counter via `Fetch-and-Add`, obtaining the value `t` right before the increment. After that, it stores the element into the reserved cell `A[t]` and finishes. The `Dequeue()` operation works symmetrically, incrementing its `Head` counter and extracting the element from the reserved cell `A[h]`. Figure 1 below shows how the data structure changes when inserting and extracting an element.

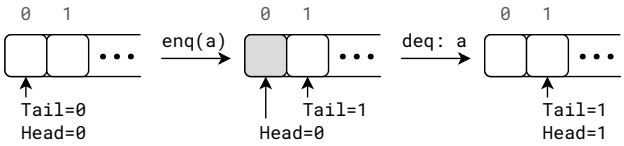


Figure 1. Infinite array queue structure.

Due to concurrency, it is possible for `Enqueue(..)` to increment `Tail` and pause, so `Dequeue()` may reserve a cell that is still empty. To make progress, `Dequeue()` poisons the empty cell by moving its state to a special ⊥ value and restarts. After that, `Enqueue(..)` observes that the cell is broken and also restarts; it installs the element via `CAS` to synchronize. Listing 1 presents the corresponding pseudocode.

The LCRQ design. Afek and Morrison elaborated this idea and developed an extremely efficient LCRQ algorithm, using *ring buffers* under the hood, which they call CRQ-s (Concurrent Ring Queues). Roughly, CRQ is a bounded queue with relaxed semantics regarding the conditions under which `Enqueue(..)` can fail. Similarly to the infinite array queue, `Enqueue(..)` increments its `Tail` counter and processes the

PPoPP '23, February 25-March 1, 2023, Montreal, QC, Canada

ACM 979-8-4007-0015-6/23/02…$15.00
https://doi.org/10.1145/3572848.3577485

```
1  Head, Tail: Long // 64-bit counters
2  A: E[] // Infinite array for elements of type E
3
4  fun Enqueue(item: E) = while (true) {
5    t := Tail.FAA(1)
6    if (A[t].CAS(null, item)) return
7  }
8
9  fun Dequeue(): E = while (true) {
10   if (Head >= Tail) return null // empty?
11   h := Head.FAA(1)
12   if (A[h] == null && A[h].CAS(null, ⊥)) continue
13   return A[h]
14 }
```

Listing 1. Infinite array queue algorithm.

cell `A[t % R]`, where `R` is the size of the ring buffer. Symmetrically, `Dequeue()` increments `Head` and processes the cell `A[h % R]`. Once the current ring buffer is full, `Enqueue(..)` closes it for further insertions and creates a new CRQ, constructing a linked list of them. Even though the high-level idea is similar, making the array circular changes the synchronization completely.

In the infinite array queue, each cell can be processed by exactly one `Enqueue(..)` and exactly one `Dequeue()`, making the synchronization between them straightforward. In the ring buffer structure, each `R`-s `Enqueue(..)` and `Dequeue()` attempt manipulates the same cell in the ring buffer of size `R`. This leads to several potential issues, such as adding an element to the middle of the queue or out-of-order removals. To solve the races, LCRQ equips each cell with a 64-bit *epoch* counter. Intuitively, it ensures that `Dequeue()` extracts the element only from its `h / R` epoch, while `Enqueue(..)` never installs the element if the conjugate `Dequeue()` of the same epoch has already processed and skipped the cell.

LCRQ non-portability. The main problem of the LCRQ design is its portability. Each cell is equipped with an epoch counter, which LCRQ updates atomically along with the cell state via `CAS2`. While this instruction gains popularity in modern hardware architectures, most programming languages, such as Java, Kotlin, or Go, do not provide such a primitive, making it impossible to implement LCRQ in them.

Several approaches were suggested to eliminate the `CAS2` usages. Ramalhete [22] proposed a simple lock-free queue that never reuses ring buffers, allocating a new one when reaching the end of the internal array. This design reminiscences the infinite array queue and does not use epochs, allowing only one `Enqueue`-`Dequeue` pair to process the cell. Yang and Mellor-Crummey [27] extended this approach to provide wait-freedom. These works and our experiments show that LCRQ surpasses both solutions.

Another way to get rid of `CAS2` is to manipulate 32-bit values and 32-bit epochs, which can be packed into a single 64-bit word and updated atomically by `CAS`. Nikolaev [20] uses such queues to build his SCQ ring buffer algorithm, maintaining an array of elements and two queues of 32-bit indices in this array, referencing free cells for `Enqueue(..)` and occupied cells for `Dequeue()`. Feldman and Dechev [5] approached the problem another way, integrating the epoch field into elements. Besides that, their algorithm stores epochs in empty cells, which is technically impossible in most languages with garbage collection, such as Java or C#. Compared to LCRQ, both these solutions are significantly less efficient.

Our contribution. In this paper, we present the LPRQ concurrent queue algorithm, a *portable* modification of LCRQ, which does not require `CAS2`. Essentially, we apply two techniques to the original LCRQ design. First, we get rid of `CAS2` in `Dequeue()`, making the `Enqueue(..)` operation to be responsible for maintaining epochs. Yet, it should place the element only in a correct epoch, which LCRQ solves via `CAS2`. Instead, LPRQ "locks" the cell by installing a *thread-unique* token, checking the epoch and replacing the token with the element after that. From the implementation side, this thread-unique token is usually a reference to the currently running thread (e.g., `java.lang.Thread` instance on the JVM).

We have implemented the LPRQ algorithm in C++ and compared it against the original LCRQ solution and various queues that do not require `CAS2`. Our experiments show that our LPRQ algorithm provides the same performance as LCRQ, surpassing the best of other solutions by up to `1.6`×.

2 Preliminaries

Memory model and atomic operations. For simplicity, we assume a sequentially consistent memory model. Besides the plain reads and writes, we use `Compare-and-Set (CAS)` and `Fetch-and-Add (FAA)` atomic operations, which are accessible in all modern programming languages. In the LCRQ algorithm discussion, we also use the non-portable `CAS2` instruction, which updates two *contiguous* (not arbitrary) memory locations; it is also known as `cmpxchg16b` on x86.

Memory reclamation. For simplicity, we assume that the environment is provided with a garbage collector. For manual memory reclamation, one can use a technique such as Hazard Pointers [15] or Hazard Eras [23].

Nullability. The original LCRQ algorithm and our LPRQ modification assume that the element type `E` is provided with a special `null` value, which is never inserted into the queue. We use it to specify empty array cells. Returning `null` in `Dequeue()` indicates that the queue is empty.

Thread tokens. Our PRQ algorithm assumes that the environment provides *thread-unique* tokens, denoted as $\top_{threadId}$, which can be stored in the same locations as elements. In short, `Enqueue(..)` installs such a token in the cell before storing the element to ensure that it performs the operation in a correct epoch. In practice, thread tokens can be implemented by reserving a single bit of the value to distinguish them from elements. In case this bit is set, the others store the index of the thread which owns the token. Otherwise

the other bits store the element. On some platforms, which do not allow "bit stealing", a reference to the currently running thread can be used as a thread token. E.g., on the JVM `java.lang.Thread` instance can be efficiently obtained via `Thread.currentThread()`.

3 The LPRQ Algorithm

We begin by discussing the original LCRQ design, which uses the non-portable `CAS2` instruction. Then, we eliminate its usage in two steps. First, we get rid of `CAS2` in `Dequeue()`, making the `Enqueue(..)` operation to be responsible of maintaining per-cell epochs. Next, we get rid of `CAS2` in `Enqueue(..)` by using *thread-unique tokens* to "lock" the cell and perform the rest of the synchronization safely.

3.1 The LCRQ Overview

The key ingredient behind LCRQ is a special *ring buffer* (CRQ) data structure, which can be closed for further insertions. In brief, CRQ manipulates a pre-allocated array of constant size and two positioning counters for `Enqueue(..)` and `Dequeue()` operations. When CRQ is full, `Enqueue(..)` closes it to prevent further insertions and allocates a new CRQ, constructing a linked list of them.

LCRQ on top of ring buffers. Listing 2 presents the high-level LCRQ implementation that manipulates ring buffers (CRQ-s). Our LPRQ algorithm leverages the same design, replacing the non-portable CRQ ring buffer implementation with the new PRQ (Portable Ring Queue) one.

Similarly to the Michael-Scott queue [16], the linked list is specified via `Head` and `Tail` pointers (line 2). To insert an element, `Enqueue(..)` reads the last CRQ (line 6) and tries to add the element into it (line 7). On success, the operation finishes immediately. Otherwise, the current CRQ is full and closed for further insertions. This way, `Enqueue(..)` allocates a new CRQ instance with the element inserted right away (line 9). Then, it tries to add this new CRQ right after the current one in a way similar to the Michael-Scott queue (lines 10–15). In case a concurrent `Enqueue(..)` has already added a new CRQ, the current operation restarts.

The `Dequeue()` operation manipulates the `Head` CRQ and tries to extract an element from it, finishing on success (lines 19–21). In case the extraction fails, the current CRQ is empty, while the queue may still contain non-empty CRQ-s. Thus, `Dequeue()` reads the next CRQ, returning `null` if it is not present (line 23). Otherwise, concurrent `Enqueue(..)`-s could add elements to the current CRQ between we observed it empty and read `crq.Next`. Therefore, `Dequeue()` tries to extract an element from the current queue again (line 25), finishing on success (line 26). Otherwise, the current CRQ is empty and closed, so `Dequeue()` updates the `Head` pointer to the next CRQ and restarts (line 28).

High-level ring buffer design. We move our attention to the CRQ implementation. Figure 2 shows the ring buffer

```
1  class LCRQ<E> {
2    Head, Tail: CRQ
3
4    fun Enqueue(item: E) = while (true) {
5      // fast-path: add to the current CRQ
6      crq := tail
7      if (crq.Enqueue(item)) return
8      // slow-path: Tail is full, add new CRQ
9      newTail := CRQ(); newTail.Enqueue(item)
10     if (crq.Next.CAS(null, newTail)) {
11       Tail.CAS(crq, newTail)
12       return
13     } else {
14       Tail.CAS(crq, crq.Next)
15     }
16   }
17
18   fun Dequeue(): E = while (true) {
19     crq := Head
20     res := crq.Dequeue()
21     if (res != null) return res
22     // failed, is this queue empty?
23     if (crq.Next == null) return null
24     // `crq` is closed but may store elements
25     res := crq.Dequeue()
26     if (res != null) return res
27     // `crq` is empty, update HEAD and restart
28     Head.CAS(crq, crq.Next)
29   }
30 }
31 class CRQ<E> {
32   Next: CRQ<E> = null
33   // enqueues item and returns true
34   // or closes CRQ and returns false
35   fun Enqueue(item: E): Bool
36   fun Dequeue(): E // returns null if CRQ is empty
37 }
```

Listing 2. The high-level LCRQ [18] implementation manipulates CRQ-s. The LPRQ high-level design is identical with a difference only in replacing CRQ with PRQ from Listing 4.

structure that bases on an array `A` of size `R` to store elements equipped with `Head` and `Tail` positioning counters; their values modulo `R` specify indices for the next element insertion and extraction. Initially, all array cells are empty and store `null`. The current size of the ring buffer equals `(Tail - Head)` and never exceeds `R`.

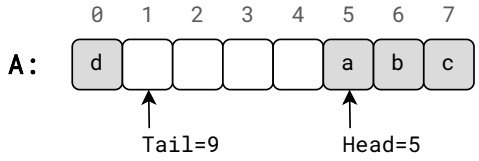


Figure 2. An example of a ring buffer of size $R = 8$. `Head % R` points to the next element to be extracted, while `Tail % R` points to the next cell to put an element.

To add an element, `Enqueue(..)` first increments its `Tail` counter via the atomic `Fetch-and-Add` instruction, obtaining index `t` right before the increment and placing the element to the cell `A[t % R]` if it is empty. Symmetrically,

`Dequeue()` increments `Head` via `FAA` and extracts the element from `A[h % R]`. This design reminds the infinite array queue in Listing 1.

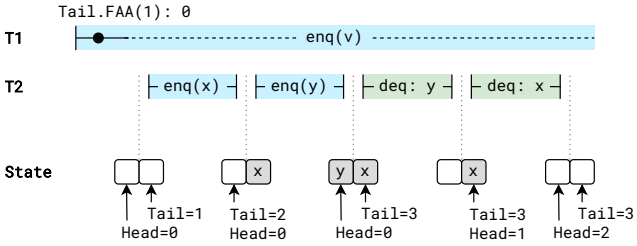
Cell epochs. For further discussion, we need to introduce *operation index* and *operation cycle* notions. When an operation increments `Head` or `Tail` via `FAA`, it returns the value `i` of the counter right before the increment. We call this value `i` the *index* of the operation, while `i / R` is the operation *cycle*.

It is crucial for `Dequeue()` to take element only from the conjugate `Enqueue(..)` of the same operation cycle. To demonstrate that, we consider two examples on a ring buffer of size `R = 2`, showing that extracting an element that was put on another cycle results in the FIFO semantics violation. First, we consider the execution presented in Figure 3a. The thread $T1$ invokes `Enqueue(..)`, which increments `Tail` and gets preempted by the scheduler. The execution switches to the thread $T2$, which begins by inserting x and y. Specifically, $enq(x)$ obtains the index 1 and puts x into the 1-st cell on cycle 0, while $enq(y)$ obtains the index 2 and puts y into the 0-th cell on cycle 1. Then, $T2$ performs two `Dequeue()`-s, extracting x and y in reverse order. The problem is that $deq : y$ (0-th cell on cycle 0) retrieves the element that was put on a higher cycle 1; thus, extracting it from the future.

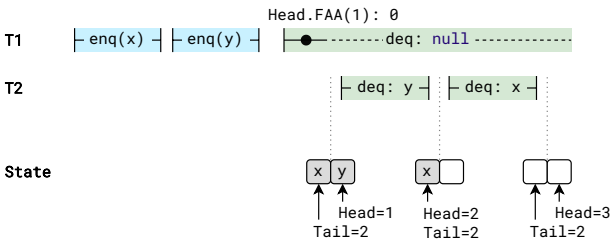
A similar issue occurs when `Dequeue()` retrieves an element that was placed on a lower cycle; we present such an execution in Figure 3b. First, $T1$ inserts x and y on cycle 0. After that, it invokes `Dequeue()`, which increments `Head` and gets preempted. The execution switches to $T2$, which performs two `Dequeue()`-s. (Recall that `Head` has already been incremented by $T1$.) The first `Dequeue()` obtains the index 1 and extracts y from the 1-st cell on cycle 0. The second operation obtains index 2, taking x from the 0-th cell on cycle 1. Similar to the previous example, the elements are extracted in reverse order, violating the FIFO queue semantics.

One way to ensure that an element can be taken only on the correct cycle is to associate each cell with an *epoch*. When `Enqueue(..)` puts an element into a cell, it also updates the cell epoch to its operation cycle. In turn, `Dequeue()` extracts the element only if the cell epoch matches its cycle.

Figure 4 shows how this scheme fixes the incorrect execution in Figure 3b. The execution begins by adding x and y on cycle 0. After that, $T1$ invokes `Dequeue()`, which increments `Head` and gets preempted. The execution switches to $T2$, which performs two `Dequeue()`-s. As in the previous example, the first `Dequeue()` obtains the index 1 and extracts y from the 1-st cell on cycle 0. However, the next `Dequeue()` in $T2$ cannot retrieve x anymore, as the 0-th cell epoch is 0, while the operation cycle is 1. Thus, this `Dequeue()` restarts, finds the ring buffer empty, and returns `null`. After the execution switches back to $T1$, the corresponding `Dequeue()` finishes by extracting x from the 0-th cell.



(a) Dequeuing from the future breaks the FIFO semantics.



(b) Dequeuing from the past breaks the FIFO semantics.

Figure 3. These two examples show that `Dequeue()` must take element only from the conjugate `Enqueue(..)`, cycles of which coincide. Here, two threads access a ring buffer of size $R = 2$.

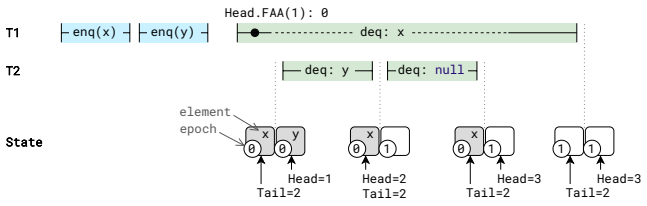


Figure 4. Epochs help to keep FIFO order of elements.

The CRQ algorithm. We need to update the cell epoch and its state atomically; otherwise, `Dequeue()` may observe an element of incorrect epoch and take it, breaking the FIFO semantics. CRQ implements this atomic update via the hardware `CAS2` instruction.

Listing 3 presents the CRQ pseudocode. The cell data structure has a 64-bit `Value` (line 5) and a 64-bit `SafeAndEpoch` (line 2) fields; the latter packs a special `Safe` flag along with the `Epoch`. Besides the `Head` and `Tail` counters, CRQ additionally stores the `Closed` flag (line 9)[1], which indicates whether the queue is closed for insertions, and a pointer to the next CRQ in the linked list of them, which is used by LCRQ in Listing 2. `Enqueue(..)` and `Dequeue()` operations proceed in iterations (see the loops at lines 13 and 34). Each iteration starts by performing `FAA` on `Tail` (line 14) or `Head` (line 35), obtaining the iteration *index* and its *cycle*. On each

[1]Originally, CRQ reserves a higher bit of `Tail` to store the `Closed` flag. For simplicity, we put it into a separate variable.

```
1  struct Cell<E> {
2    SafeAndEpoch: packed { // 64 bits
3      Safe: Bool = true, Epoch: Long = 0
4    }
5    Value: E = null
6  }
7
8  Head: Long = 0, Tail: Long = 0
9  Closed: Bool = false
10 A: Cell<E>[R]
11 Next: CRQ<E> = null // pointer to the next CRQ
12
13 fun Enqueue(item: E): Bool = while (true) {
14   t := Tail.FAA(1)
15   if (Closed) return false
16   cycle := t / R; i := t % R
17   <safe, epoch> := A[i].SafeAndEpoch
18   value := A[i].Value
19
20   if (value == null && // the cell is empty
21       // and enqueue is not overtaken
22       epoch <= cycle && (safe || Head <= t)) {
23     // enqueue transition
24     if (A[i].CAS2(<<safe, epoch>, null>,
25                   <<true, cycle>, item>))
26       return true
27   }
28   // is the queue full?
29   if (t - Head >= R) {
30     Closed := true
31     return false
32   }
33 }
34 fun Dequeue(): E = while (true) {
35   h := Head.FAA(1)
36   cycle := h / R; i := h % R
37   while (true) { // try update the cell state
38     <safe, epoch> := A[i].SafeAndEpoch
39     value := A[i].Value
40
41     when { // transitions according to Figure 5
42       epoch == cycle && value != null -> {
43         // dequeue transition
44         if (A[i].CAS2(<<safe, epoch>, value>,
45                       <<safe, cycle + 1>, null>))
46           return value
47       }
48       epoch <= cycle && value == null -> {
49         // empty transition
50         if (A[i].CAS2(<<safe, epoch>, value>,
51                       <<safe, cycle + 1>, value>))
52           break
53       }
54       epoch < cycle && value != null -> {
55         // unsafe transition
56         if (A[i].CAS2(<<safe, epoch>, value>,
57                       <<false, epoch>, value>))
58           break
59       }
60       // else epoch > cycle
61       else -> break  // deq is overtaken
62     }
63   }
64   // is the queue empty?
65   if (Tail <= h + 1) return null
66 }
```

Listing 3. The CRQ algorithm by Morrison and Afek [18].

iteration, the operation reads the cell state (lines 17–18, 38–39) and performs an update transition, restarts, or finishes. Figure 5 illustrates possible cell state transitions.

Consider an iteration of Dequeue() with index h and cycle c. If case the cell's epoch is greater than c (line 61), this Dequeue() attempt has been overtaken between the counter increment and reading the cell epoch. In this case, this Dequeue() attempt fails, and the operation restarts. Otherwise, one of the following transitions is performed:

- If the cell contains an element and the epoch matches the iteration cycle c, a *dequeue transition* is performed. Dequeue() tries to extract the element and increment the cell epoch, via CAS2 (line 44), finishing on success.
- If the cell is empty, an *empty transition* is performed, preventing Enqueue(..) to put an element on cycle c. For that, Dequeue() atomically increases the cell epoch to $c + 1$ if the cell is empty (line 50).
- At last, if the cell contains an element, but the epoch is lower than c, an *unsafe transition* is performed. To safely skip this cell, Dequeue() has to prevent installing an element on cycle c. The trick is to reset the Safe bit (line 56), making Enqueue() eligible to install an element only if Head is lower than the global cell index (line 22).

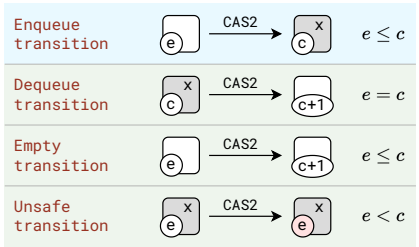


Figure 5. Cell state transitions in CRQ: c is the operation cycle, e is the cell epoch before the transition.

Whenever CAS2 that performs a transition fails, the cell update procedure restarts (loop at line 37). At the same time, if Dequeue() completes an iteration without extracting an element, it reads Tail to check if the queue is empty (line 65), returning null in this case and restarting if the queue might have elements.[2]

Now consider an iteration of Enqueue(..). After performing FAA on Tail, Enqueue(..) checks if the queue is closed (line 15). In this case, it immediately returns false. Then, if

[2] As proposed by Morrison and Afek [18], Dequeue() may also advance Tail up to the current Head before returning null. We omit this optimization for simplicity.

the cell is empty and `Enqueue(..)` is not overtaken (lines 20–22), it tries to perform an *enqueue transition*, returning `true` on success. On failure, it checks whether the queue is potentially full and, therefore, should be closed (line 29). If so, `Enqueue(..)` updates the `Closed` flag to `true` (line 30), preventing further insertions, and returns `false`. In case this CRQ has remaining space, `Enqueue(..)` proceeds to the next iteration.

The *enqueue transition* (line 24) puts the element to the cell and advances the epoch up to the iteration cycle. Notably, the operation must verify that it is not overtaken by `Dequeue()` of a higher cycle (line 22). Additionally, in case the `Safe` bit is set, `Enqueue(..)` cannot rely on the cell epoch to ensure that the operation has not been overtaken. Instead, it reads the current `Head` value and compares it to the iteration index (line 22). On successful transition, `Enqueue(..)` also sets the `Safe` bit to `true`, thus, recovering the cell.

3.2 The PRQ Algorithm

To construct our PRQ algorithm, we apply two transformations to CRQ. First, we shift cycle numbers and modify transition preconditions, which allow to replace all `CAS2` usages in `Dequeue()` with plain `CAS`-s. Second, we propose an efficient way to emulate `CAS2` in `Enqueue(..)`, which it uses to place the element in the correct epoch, with a sequence of `CAS`-s. Unlike traditional `k-CAS` algorithms [8], our approach does not use descriptors and involves neither allocation nor memory indirection.

***Elimination of* `CAS2` *in* `Dequeue()`.** We apply the following changes to the original CRQ algorithm:

- We initialize `Head` and `Tail` with R, so the operations start from cycle 1 (unlike 0 in CRQ).
- `Enqueue(..)` is eligible to place the element only when the cell epoch is *strictly* lower than the operation cycle.
- When `Enqueue(..)` places the element, it raises the cell epoch to the operation cycle. Thus, successful `Enqueue(..)`-s always increase the cell epoch.
- When `Dequeue()` extracts the element, it keeps the cell epoch unchanged. The only exception is the empty transition, which raises the epoch to the operation cycle.

These changes effectively turn all `CAS2` in `Dequeue()` to *double-compare single-swap* operations, changing only the `Value` field on successful retrieval, and updating `SafeAndEpoch` in case of empty and unsafe transitions. We need to add one more ingredient to implement them via plain `CAS`-s. Intuitively, `Dequeue()` should read a consistent snapshot of the cell state before updating it. Like in CRQ, it reads `SafeAndEpoch` followed by `Value` (lines 38–39), additionally verifying that `SafeAndEpoch` has not been changed. If so, a correct cell snapshot has been obtained, as epochs always increase, and the `Safe` flag resets only when `Enqueue(..)` increases the cell epoch. If the snapshot has not been obtained, `Dequeue()` tries to do that again.

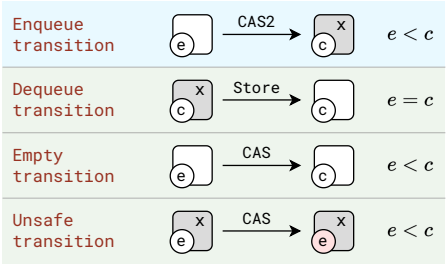


Figure 6. Cell state transitions in the modified CRQ algorithm: c is the operation cycle, e is the cell epoch before the transition.

Figure 6 summarizes the cell state transitions in the modified CRQ algorithm, which we discussed above. We present the corresponding pseudocode in Appendix A.

***`CAS2` emulation in* `Enqueue(..)`.** With the changes discussed above, only `Enqueue()` requires `CAS2` to place its element and increase the epoch atomically (line 24 in Listing 3). We provide a smart way to simulate this `CAS2` with a three-step procedure illustrated in Figure 7:

1. *Cell reservation.* To place the element and increase the epoch, `Enqueue(..)` first "locks" the cell by installing a *thread-unique token* $\top_{curThread}$ into it. (Regarding the implementation, a reference to the currently running thread can be used as a thread token.)
2. *Epoch promotion.* Next, it increases the cell epoch up to the operation cycle.
3. *Element publishing.* Finally, `Enqueue(..)` replaces its $\top_{curThread}$ with the element.

The key idea is that other operations are not allowed to change the cell epoch when it stores a thread token, so when `Enqueue(..)` replaces it with the element on the last step, it is guaranteed that the cell epoch has not been changed.

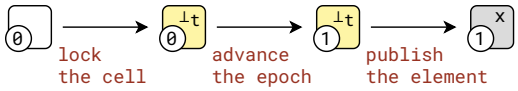


Figure 7. `CAS2` emulation in `Enqueue(..)` to place the element correctly. Here, thread t inserts x on cycle 1.

The pseudocode. Listing 4 presents the pseudocode of our PRQ algorithm. As for the data structure, it is the same as one of CRQ (Listing 3); the only difference is that PRQ initializes `Head` and `Tail` with R instead of 0 (line 2).

The `Enqueue(..)` operation begins by incrementing its `Tail` counter, obtaining index `h` (line 5). Then, it checks whether the cell is empty or locked and the operation is not overtaken, restarting in this case (lines 13–15). Otherwise, it tries to place the element following the three-step procedure described above (lines 18–29).

The `Dequeue()` operation begins by incrementing its `Head` counter and obtaining index `h` (line 41). Next, it snapshots the cell, obtaining its epoch, the `Safe` flag, and the cell value (lines 44–47), followed by one of the listed transitions:

```
1  // initially
2  Head = R; Tail = R
3
4  fun Enqueue(item: E): Bool = while (true) {
5    t := Tail.FAA(1)
6    if (Closed) return false
7
8    cycle := t / R; i := t % R
9
10   <safe, epoch> := A[i].SafeAndEpoch
11   value := A[i].Value
12
13   if (value is null or ⊤ && // not occupied
14       // and enqueue is not overtaken
15       epoch < cycle && (safe || Head <= t)) {
16
17     // lock the cell with the thread token
18     if (!A[i].Value.CAS(value, ⊤curThread))
19       goto checkOverflow
20
21     // advance the epoch
22     if (!A[i].SafeAndEpoch.CAS(<safe, epoch>,
23                                <true, cycle>)) {
24       A[i].Value.CAS(⊤curThread, null) // clean up
25       goto checkOverflow
26     }
27
28     // publish item
29     if (A[i].Value.CAS(⊤curThread, item))
30       return true
31   }
32
33   checkOverflow: // is the queue full?
34   if (t - Head >= R) {
35     Closed := true
36     return false
37   }
38 }
39
40 fun Dequeue(): E = while (true) {
41   h := Head.FAA(1)
42   cycle := h / R; i := h % R
43   while (true) { // try update the cell state
44     <safe, epoch> := A[i].SafeAndEpoch
45     value := A[i].Value
46     if (<safe, epoch> != A[i].SafeAndEpoch)
47       continue // inconsistent view of the cell
48
49     when {
50       epoch == cycle && value is not null or ⊤ -> {
51         // dequeue transition
52         A[i].Value := null
53         return value
54       }
55       epoch <= cycle && value is null or ⊤ -> {
56         // empty transition
57         // unlock the cell
58         if (value is ⊤ &&
59             !A[i].Value.CAS(value, null))
60           continue
61         // advance the epoch
62         if (A[i].SafeAndEpoch.CAS(<safe, epoch>,
63                                   <safe, cycle>))
64           break
65       }
66       epoch < cycle && value is not null or ⊤ -> {
67         // unsafe transition
68         if (A[i].SafeAndEpoch.CAS(<safe, epoch>,
69                                   <false, epoch>))
70           break
71       }
72       // epoch > cycle
73       else -> break // deq is overtaken
74     }
75   }
76   // is the queue empty?
77   if (Tail <= h + 1) return null
78 }
```

Listing 4. The PRQ algorithm. Yellow-highlighted operations emulate CAS2.

- *Dequeue transition* is performed in case the cell is occupied and the epoch matches the iteration cycle (line 50) — it replaces the stored element with `null` (line 52) and finishes (line 53). As the epoch cannot be changed while the cell is occupied, it is safe to use an ordinary store instead of a CAS here.
- *Empty transition* is applied either when the cell is empty or when it is locked and the epoch is lower than the operation cycle (line 55). For that, `Dequeue()` first replaces the thread token with `null` (line 59), advancing the epoch to the operation cycle after that (line 62). In case any of these CAS-s fails, the cell update procedure restarts.
- *Unsafe transition* is performed on an occupied cell if its epoch is lower than the operation cycle. Like in CRQ, `Dequeue()` resets the `Safe` bit (line 68) and restarts.

Notably, when `Enqueue(..)` restarts, it also reads the `Head` counter to verify that the queue is not full (line 34), closing it (line 35) and returning `false` otherwise.

4 Correctness

Similar to LCRQ, our LPRQ algorithm constructs a linked list of ring buffers, which follow the semantics of a tantrum queue [18]. Specifically, `Enqueue(..)` can non-deterministically refuse to insert an element and close the ring buffer. After it is closed, all further insertion attempts are bound to fail. To show the LPRQ correctness, we prove that our PRQ ring buffer is a linearizable tantrum queue. As the high-level part LPRQ, which manipulates ring buffers (see Listing 2), is identical to the one of LCRQ, we do not analyze it.

PRQ linearizability. To prove that PRQ is a linearizable tantrum queue, we follow the approach by the LCRQ authors [18]. Consider an execution E of operations $< op : res >$, where $op \in \{enq, deq\}$ and returns res. We assume that an operation op is *active* between its first and last events in E. We denote the result of the last `Head.FAA(1)` / `Tail.FAA(1)` call performed by the `Dequeue()` / `Enqueue(..)` operation op as $index(op)$.

First, we show that each successful `Enqueue(x)` invocation is paired with `Dequeue()` of the same *index*, which returns `x`. Essentially, it guarantees that `Dequeue()`-s retrieve elements of correct epochs, and successfully added elements do not disappear.

Lemma 4.1. *Consider a successful* `Enqueue(x)` *operation* $< enq : true >$ *with* $index(enq) = i$. *If the execution E contains a* `Dequeue()` *operation deq, which performs* `FAA` *on* `Head` *that returns i, the following statements are correct:*

(1) $index(deq) = i$,
(2) deq returns x,
(3) if deq obtains its index before enq, deq is still active when enq performs its last `FAA` *on* `Tail`.

Proof. Consider the execution of *enq* after it has performed the last `FAA` on `Tail`, obtaining the index i. The `CAS` that installs the thread token in the cell $A[i \% R]$ (line 18), the `CAS` that advances the epoch (line 22), and the `CAS` that places its element x (line 29) must succeed. In case one of the `CAS`-s failed, the operation would restart, performing a new `FAA` on `Tail` and obtaining a higher index.

To show that `Enqueue(..)` places the element in a correct epoch, we prove that other operations do not change the cell epoch between it locks the cell (line 18) and replaces the thread token with the element (line 29). Indeed, when another `Enqueue(..)` advances the epoch (line 22), it first installs its thread token (line 18), which would prevent the element publishing. Similarly, when `Dequeue()` advances the epoch (line 62), it first "unlocks" the cell (line 59), which also would result in unsuccessful element publishing.

The discussion above shows that at some point in E, the cell state is `{ SafeAndEpoch : <true, i / R>, Value : x}`. Next, we prove that only `Dequeue()` with index i can take x, and other operations cannot change the cell epoch until this `Dequeue` retrieves the element. Because the cell epoch is always advanced, it is unique for each published element. Thus, after the read of a consistent snapshot of the cell state (lines 44–47), `Dequeue()` can observe x only with the epoch i/R, and so it will perform the dequeue transition (line 52) only if its cycle matches the `Enqueue(..)`'s cycle.

Now we show that the lemma statements are correct. Obviously, $index(deq) \geq i$, as `Head.FAA(1)` returned i at some point in E, according to the lemma. Suppose that $index(deq) > i$. In this case, one of the following events must happen in *deq* after it obtained the index i and before it obtains $i + 1$:

(a) *deq* performs empty transition (line 62)
(b) *deq* performs unsafe transition (line 68)
(c) *deq* observes `SafeAndEpoch.Epoch > i / R` (line 73)

We show that assuming any of these events leads to a contradiction. In case of (a), *deq* raises the epoch to i/R. So *enq* could not raise it to the same value with its second `CAS` (line 22). In case of (b), *deq* resets the `Safe` bit (line 68). Succeeding of *enq* implies that some *enq'* (maybe *enq*) with $index(enq') \leq i$ observed and recovered the unsafe cell (line 22). For that, it must observe $\texttt{Head} \leq index(enq')$, which is impossible, as $\texttt{Head} > i$. Before *enq* published the element, the cell epoch was less than i/R. After that, the epoch became i/R and could not rise while the cell was occupied. So (c) is also impossible. As result, *deq* can only finish with $index(deq) = i$, returning the element x (line 53), which implies the (1) and (2) lemma statements.

At last, (3) follows by the fact that *deq* cannot complete before *enq* raises the cell epoch and installs the element, which happens after its last `Tail.FAA(1)`. □

Theorem 4.2. *PRQ is a linearizable tantrum queue.*

Proof. We construct a sequential history of the operations H, processing E sequentially, one event at a time. Then we prove that H is a linearization of the concurrent history induced by E. Further, we say "linearize" as a shortcut for "append to H".

First, we provide linearization points for $< enq : \cdot >$ and $< deq : \texttt{null} >$ operations:

- $< enq : \texttt{false} >$. Unsuccessful `Enqueue(..)`-s either observe that the ring buffer is already closed by reading the `Closed` flag (line 6) or close the queue by setting the flag to `true` (line 35). We linearize them at one of these points.
- $< enq : \texttt{true} >$. Successful `Enqueue(..)`-s linearize on their last `Tail.FAA(1)` (line 5).
- $< deq : \texttt{null} >$. Unsuccessful `Dequeue()`-s linearize on their last read of the opposite `Tail` counter (line 77), detecting that the ring buffer is empty and finishing.

Now we need to linearize successful `Dequeue()`-s. For that, we maintain a set S of active and not yet linearized `Dequeue()` operations. When the *deq* operation performs its last `Head.FAA` (line 41), we either:

- append *deq* to H if $index(deq) < \texttt{Tail}$, or
- put *deq* into S if $index(deq) \geq \texttt{Tail}$, linearizing it later.

We linearize `Dequeue()`-s from S right after `Enqueue(..)`-s of the same index. Specifically, when an `Enqueue(..)` operation *enq* linearizes, we remove the *deq* operation with $index(deq) = index(enq)$ from S (if one exists), and append it to H.

Next, we show that H meets the tantrum queue sequential specification. Successful $< deq : \texttt{res} >$ and $< enq : \texttt{true} >$ linearize in ascending order of their indices, correctly transferring elements, as Lemma 4.1 shows. When $< deq : null >$ linearizes, $\texttt{Head} \geq \texttt{Tail}$, and all successful `Dequeue()`-s with indices lower than `Tail` are already linearized. At last, unsuccessful `Enqueue()`-s ($< enq : \texttt{false} >$) must never precede successful ones. We guarantee it by reading the `Closed` flag on each operation attempt (line 6), so all further `Enqueue()`-s are bound to fail.

By the H construction and Lemma 4.1, linearization points of all operations lie within their execution intervals in E. This

completes the proof of the theorem, showing that PRQ is a linearizable tantrum queue. □

Progress Guarantees. Now we discuss the PRQ and LPRQ progress guarantees. First, we show that both `Enqueue(..)` and `Dequeue()` in the PRQ implementation (Listing 4) are obstruction-free.

Theorem 4.3. *PRQ is obstruction-free.*

Proof. The `Enqueue(..)` operation performs a finite number of attempts, either successfully adding the element, or closing the ring buffer when `Tail` reaches `Head + R`.

Similarly, `Dequeue()` performs finite number of attempts, either retrieving an element or returning `null` when the `Head` counter reaches `Tail`. □

We can enhance the progress conditions to operation-wise lock-freedom following the approaches in [18, 20]. Specifically, `Enqueue(..)` could close the current ring buffer after a constant number of unsuccessful attempts. In this case, the LPRQ algorithm is lock-free, similar to LCRQ (we omit the formal proof, as [18] contains it).

5 Evaluation

We have successfully implemented the proposed LPRQ algorithm in C++ and compared it against the original LCRQ solution, as well as the FAAArrayQueue by Ramalhete [22] and LSCQ by Nikolaev [20], which also leverage `Fetch-and-Add` and do not use `CAS2`. At last, we add CC-queue [4] to the comparison, which is the fastest flat-combining-based queue.

Benchmarks. We use two workloads to benchmark the algorithms. In both cases, multiple threads share a concurrent queue to transfer elements — we measure the throughput of the system. In the first "enqueue-dequeue pairs" workload, inspired by [22], each thread performs `Enqueue(..)` followed by `Dequeue()`. The second "producer-consumer" benchmark separates the threads into producers and consumers, so each thread either only sends elements or retrieves them. We measure the throughput of this workload with different producer-consumer ratios: `1:1` (the numbers of producers and consumers are the same), `1:2` (the number of consumers is twice as the number of producers), and `2:1`. In all workloads, the queues are initially empty.

Environment. We run the experiments on a `c6i.32xlarge` AWS instance [11], powered by two 3.5 GHz Intel Xeon 8375C (Ice Lake) 32-core processors, for 128 hardware threads in total; the L1, L2, and L3 cache sizes are 48KB, 1.25MB, and 54MB, respectively.

Methodology. We measure the time it takes to transfer 10 million elements through the queue and count the system throughput. To provide stable and statistically significant results, we run each experiment 10 times and count the standard deviation. We also simulate some amount of work between operations to make the workload more realistic. Specifically, we perform 8 uncontended loop cycles on average, generating a pseudorandom number in each iteration. Our internal experiments show that changing the work size does not affect performance trends. In the "producer-consumer" benchmark, we keep the work balanced, so when the number of producers is greater than the number of consumers, the latter perform less work.

Implementation details. Based on minimal tuning, we find the ring buffer size of `1024` optimal for all the algorithms; this matches the prior work [22, 27]. In all implementations, we put `Head` and `Tail` counters on separate cache lines to avoid *false sharing* and made `Dequeue()` "optimistic", so it tries to extract an element first, checking for queue emptiness only on failure, thus, reducing contention on the counters.

Regarding memory management, we use jemalloc [3] memory allocator and Hazard Pointers [15] reclamation technique. To verify that the memory reclamation scheme has no significant impact on the results, we ran an additional experiment without memory reclamation — the performance trends stayed the same.

As for the memory model, our implementation relies on the SC-DRF (sequential consistency for data-race-free programs) memory model, provided by Java, C++11, and other real-world weak memory models. We use atomic variables provided in the standard library to achieve data race freedom.

Results analysis. We present the benchmark results in Figure 8. The first thing that immediately catches the eye is that our `LPRQ` algorithm shows the same performance as LCRQ across all benchmarks. One may see that counter-intuitive, as `LCRQ` performs a single `CAS2` instruction when installing an element into the cell, while `LPRQ` makes three consecutive `CAS`-s. This difference in the algorithm does not affect performance, as in practice, performing extra atomic instructions on the same cache line induces insignificant overhead. We have collected statistics on cache misses to confirm that; please see Appendix B for the data.

Now we compare LPRQ with other queues that do not use `CAS2`. First, our solution outperforms the flat-combining-based `CC-Queue` in all workloads by up to `6×`. The LSCQ algorithm by Nikolaev is slower by `1.3×` on average and up to `1.6×` in the "1:2 producer-consumer" scenario. Finally, the `FAAArrayQueue` solution by Ramalhete shows similar performance on most of the workloads, slightly degrading when consumers prevail; LPRQ is faster by up to `1.6×`.

Artifact. One may easily reproduce the experiments we have described. An artifact that includes all the queue implementations and the benchmarks is publicly available [24]. Essentially, the artifact contains the following items:

- source code of all considered queue algorithms and benchmarks;

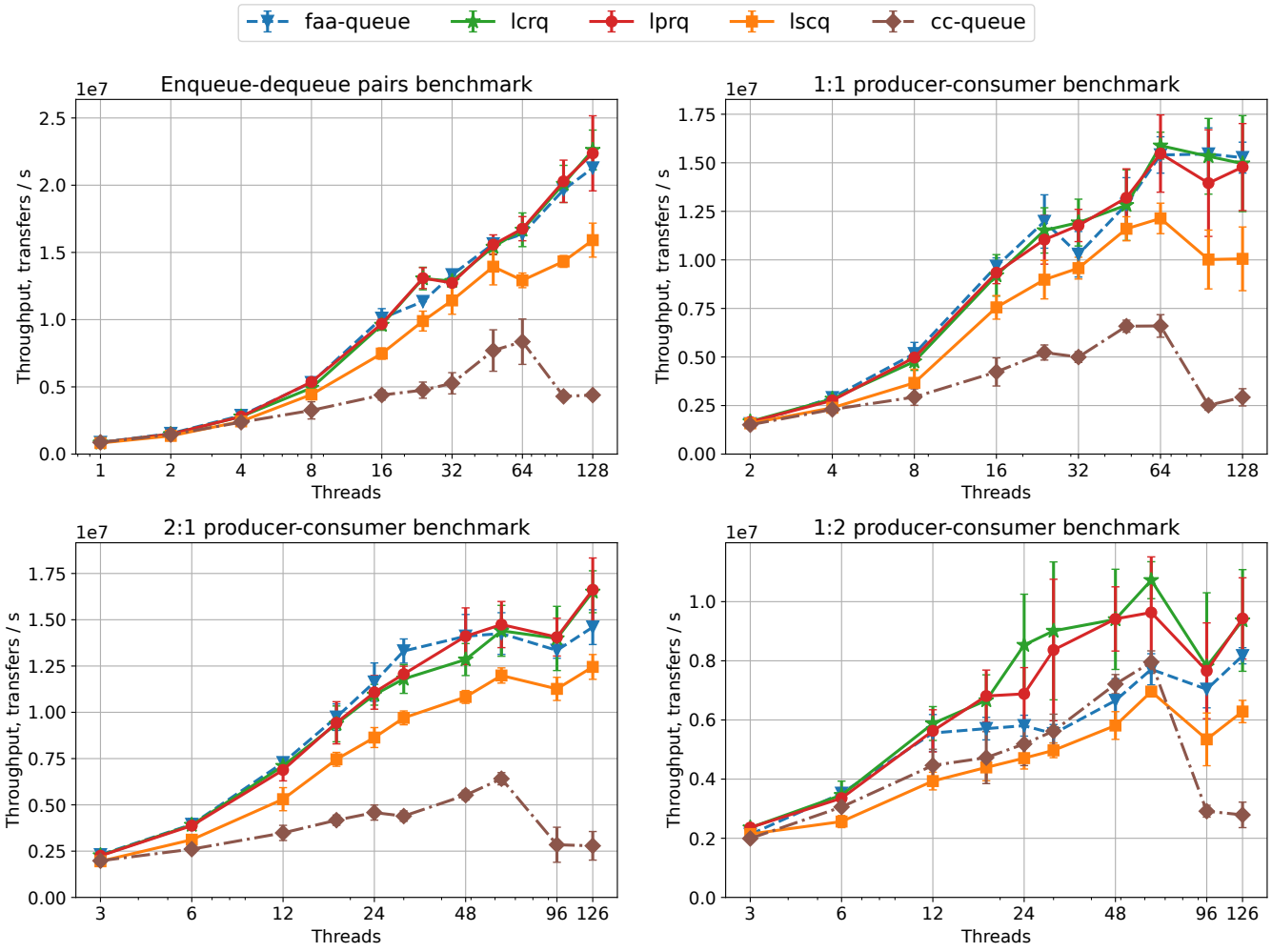


Figure 8. Evaluation of the proposed LPRQ algorithm compared to LCRQ [18], `FAAArrayQueue` [22], and LSCQ [20] that leverage `Fetch-and-Add` and the ring buffer structure. In addition, we add CC-Queue [4] that uses the flat combining technique for the comparison. The benchmarks show the system throughput when sharing the queue to transfer elements across multiple threads; see the text for details. **Higher is better.**

- script that runs the benchmarks and produces graphs similar to the ones in the paper;
- Docker image configuration, which encapsulates all required dependencies.

Apart from the experiments we presented in the paper, our software allows evaluation under different settings. For example, one can evaluate how the ring size affects the performance or test other producers-consumers ratios. Furthermore, the software is able to collect additional metrics, such as the number of allocated segments, which may be useful for investigating the properties of the algorithms deeper.

6 Related Work

Michael and Scott proposed the first lock-free queue [16] that maintains a concurrent linked list of nodes, each of which stores an element. The synchronization scheme leverages the classic `CAS`-loop design: all `Dequeue()`-s read the `Head` pointer and try to update it by `CAS`, finishing on success and restarting on failure; similar logic applies to `Enqueue(..)`. The disadvantage of this scheme is that under high contention, only one operation succeeds, while others retry as their `CAS` attempts fail, leading to sequential processing with high synchronization cost. Several works [10, 12, 14, 17, 21] reduce contention on the `Head` and `Tail` pointers by performing the synchronization at another location on `CAS` failure.

At the same time, bounded queues gained significant popularity, as they can naturally be implemented on a preallocated array to store elements with positioning counters for `Enqueue(..)` and `Dequeue()`. Importantly, these counters can be updated by unconditional `Fetch-and-Add` instruction, which significantly improves the algorithm's scalability. Yet, implementing such a queue is non-trivial. Certain

solutions [6, 7] lack linearizability, as discussed in [1, 18], while others do not provide non-blocking progress guarantee [13, 19]. In result, many algorithms [2, 25, 26] update `Head` and `Tail` via `CAS` to achieve both linearizability and non-blocking progress guarantee.

In 2013, Morrison and Afek [18] proposed a non-blocking *ring buffer* implementation, which is, essentially, a relaxed bounded queue that allows `Enqueue(..)`-s to fail spuriously. By constructing a linked list of them, they build the LCRQ unbounded queue algorithm. Regarding synchronization, LCRQ equips each cell with a special *epoch*, updating it together with the cell state via the `CAS2` instruction. This is the most scalable concurrent queue algorithm we are aware of at the point of writing the paper.

As the `CAS2` instruction is not available in most modern programming languages, several works tried to eliminate it. The most straightforward idea is not to reuse the segments of the queue, reclaiming them when `Head` exceeds R. A simple yet efficient lock-free algorithm by Ramalhete [22], as well as a more tricky Yang and Mellor-Crummey's wait-free queue [27] successfully employ this approach.

Feldman and Dechev [5] approach the problem from a different angle, integrating the epoch field into elements. Besides that, their algorithm stores epochs in empty cells, which is technically impossible in most languages with garbage collection, such as Java or C#. In the case of manual memory reclamation, storing epochs in elements and dereferencing the latter in `Dequeue()` makes the element reclamation highly non-trivial. Nikolaev proposed a novel SCQ ring buffer implementation [20], which does not require the non-portable `CAS2` instruction. The key idea is to store 32-bit values and maintain 32-bit epochs in the original LCRQ algorithm, packing them in a single 64-bit word and updating them by the standard `CAS`. Instead of storing elements in this queue, Nikolaev stores them in a separate fixed-size array, maintaining two queues of 32-bit indices of free cells for `Enqueue(..)` and of occupied cells for `Dequeue()`.

At last, other techniques, such as flat combining [9], can be used to construct an efficient queue. The fastest implementation that leverages flat combining is proposed by Fatourou and Kallimanis [4]; later works [18, 27] show that their solution scales much worse than LCRQ.

7 Discussion

In this work, we present the `LPRQ` concurrent queue algorithm, a portable modification of the famous LCRQ design that originally relies on the `CAS2` primitive, which is unavailable in most modern programming languages. Our design shows the same performance as the original LCRQ algorithm and improves the existing solutions that do not leverage `CAS2` by up to `1.6`×. We believe that our work will influence the community to include the LPRQ design in the standard libraries of modern programming languages and frameworks.

References

[1] Guy E Blelloch, Perry Cheng, and Phillip B Gibbons. 2003. Scalable room synchronizations. *Theory of computing systems* 36, 5 (2003), 397–430.

[2] Robert Colvin and Lindsay Groves. 2005. Formal verification of an array-based nonblocking queue. In *10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)*. IEEE, 507–516.

[3] Jason Evans, Mozilla Foundation, and Facebook Inc. 2016. jemalloc memory allocator. http://jemalloc.net/

[4] Panagiota Fatourou and Nikolaos D. Kallimanis. 2012. Revisiting the Combining Synchronization Technique. *SIGPLAN Not.* 47, 8 (feb 2012), 257–266. https://doi.org/10.1145/2370036.2145849

[5] Steven Feldman and Damian Dechev. 2015. A Wait-Free Multi-Producer Multi-Consumer Ring Buffer. *SIGAPP Appl. Comput. Rev.* 15, 3 (oct 2015), 59–71. https://doi.org/10.1145/2835260.2835264

[6] Eric Freudenthal and Allan Gottlieb. 1991. Process coordination with fetch-and-increment. *ACM SIGOPS Operating Systems Review* 25, Special Issue (1991), 260–268.

[7] Allan Gottlieb, Boris D Lubachevsky, and Larry Rudolph. 1983. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 5, 2 (1983), 164–189.

[8] Timothy L. Harris, Keir Fraser, and Ian A. Pratt. 2002. A Practical Multi-word Compare-and-Swap Operation. In *Distributed Computing*, Dahlia Malkhi (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 265–279.

[9] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. 2010. Flat Combining and the Synchronization-Parallelism Tradeoff. In *Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures* (Thira, Santorini, Greece) *(SPAA '10)*. Association for Computing Machinery, New York, NY, USA, 355–364. https://doi.org/10.1145/1810479.1810540

[10] Moshe Hoffman, Ori Shalev, and Nir Shavit. 2007. The Baskets Queue. In *Principles of Distributed Systems*, Eduardo Tovar, Philippas Tsigas, and Hacène Fouchal (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 401–414.

[11] Amazon Web Services Inc. 2021. Introducing Amazon EC2 C6i instances. https://aws.amazon.com/about-aws/whats-new/2021/10/amazon-ec2-c6i-instances/

[12] Alex Kogan and Erez Petrank. 2011. Wait-Free Queues with Multiple Enqueuers and Dequeuers. *SIGPLAN Not.* 46, 8 (feb 2011), 223–234. https://doi.org/10.1145/2038037.1941585

[13] Alexander Krizhanovsky. 2013. Lock-Free Multi-Producer Multi-Consumer Queue on Ring Buffer. *Linux J.* 2013, 228, Article 4 (apr 2013).

[14] Edya Ladan-Mozes and Nir Shavit. 2004. An Optimistic Approach to Lock-Free FIFO Queues. In *Distributed Computing*, Rachid Guerraoui (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 117–131.

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

[16] 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 (1998), 1–26.

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

[18] Adam Morrison and Yehuda Afek. 2013. Fast Concurrent Queues for X86 Processors. *SIGPLAN Not.* 48, 8 (feb 2013), 103–112. https://doi.org/10.1145/2517327.2442527

[19] Francesco Nigro. 2019. MpmcUnboundedXaddArrayQueue. https://github.com/JCTools/JCTools/blob/6966302c5657b22400dbebe9e54019f739eca03e/jctools-core/src/main/java/org/jctools/queues/MpmcUnboundedXaddArrayQueue.java

[20] Ruslan Nikolaev. 2019. A Scalable, Portable, and Memory-Efficient Lock-Free FIFO Queue. In *33rd International Symposium on Distributed Computing, DISC 2019, October 14-18, 2019, Budapest, Hungary (LIPIcs, Vol. 146)*, Jukka Suomela (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 28:1–28:16. https://doi.org/10.4230/LIPIcs.DISC.2019.28

[21] Or Ostrovsky and Adam Morrison. 2020. Scaling concurrent queues by using HTM to profit from failed atomic operations. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 89–101.

[22] Pedro Ramalhete. 2016. FAAArrayQueue - MPMC lock-free queue. http://concurrencyfreaks.blogspot.com/2016/11/faaarrayqueue-mpmc-lock-free-queue-part.html

[23] 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*. 367–369.

[24] Raed Romanov and Nikita Koval. 2022. *LPRQ Concurrent Queue Algorithm Evaluation.* https://doi.org/10.5281/zenodo.7337237

[25] 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 Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures* (Crete Island, Greece) *(SPAA '01)*. Association for Computing Machinery, New York, NY, USA, 134–143. https://doi.org/10.1145/378580.378611

[26] Dmitry Vyukov. 2021. Bounded MPMC queue. https://www.1024cores.net/home/lock-free-algorithms/queues/bounded-mpmc-queue

[27] Chaoran Yang and John Mellor-Crummey. 2016. A Wait-Free Queue as Fast as Fetch-and-Add. *SIGPLAN Not.* 51, 8, Article 16 (feb 2016), 13 pages. https://doi.org/10.1145/3016078.2851168

A Modified CRQ with no CAS2 in Dequeue()

```
// initially
Head = R; Tail = R

fun Enqueue(item: E): Bool = while (true) {
  t := Tail.FAA(1)
  if (Closed) return false
  cycle := t / R; i := t % R

  <safe, epoch> := A[i].SafeAndEpoch
  value := A[i].Value

  if (value == null && // the cell is empty
      // and enqueue is not overtaken
      epoch < cycle && (safe || Head <= t)) {
    if (A[i].CAS2(<<safe, epoch>, null>,
                  <<true, cycle>, item>))
      return true
  }
  // is the queue full?
  if (t - Head >= R) {
    Closed := true
    return false
  }
}
```

```
fun Dequeue(): E = while(true) {
  h := Head.FAA(1)
  cycle := h / R; i := h % R
  while(true) { // try update the cell state
    <safe, epoch> := A[i].SafeAndEpoch
    value := A[i].Value
    if (<safe, epoch> != A[i].SafeAndEpoch)
      continue // inconsistent view of the cell

    when { // transitions according to Figure 6
      epoch == cycle && value != null -> {
        // dequeue transition
        A[i].Value := null
        return value
      }
      epoch <= cycle && value == null -> {
        // empty transition
        if (A[i].SafeAndEpoch.CAS(<safe, epoch>,
                                  <safe, cycle>))
          break
      }
      epoch < cycle && value != null -> {
        // unsafe transition
        if (A[i].SafeAndEpoch.CAS(<safe, epoch>,
                                  <false, epoch>))
          break
      }
      // epoch > cycle
      else -> break // deq is overtaken
    }
  }
  // is the queue empty?
  if (Tail <= h + 1) return null
}
```

Listing 5. Modification of the CRQ algorithm [18] without CAS2 in Dequeue(). The key differences are highlighted in yellow.

B Additional Experiments

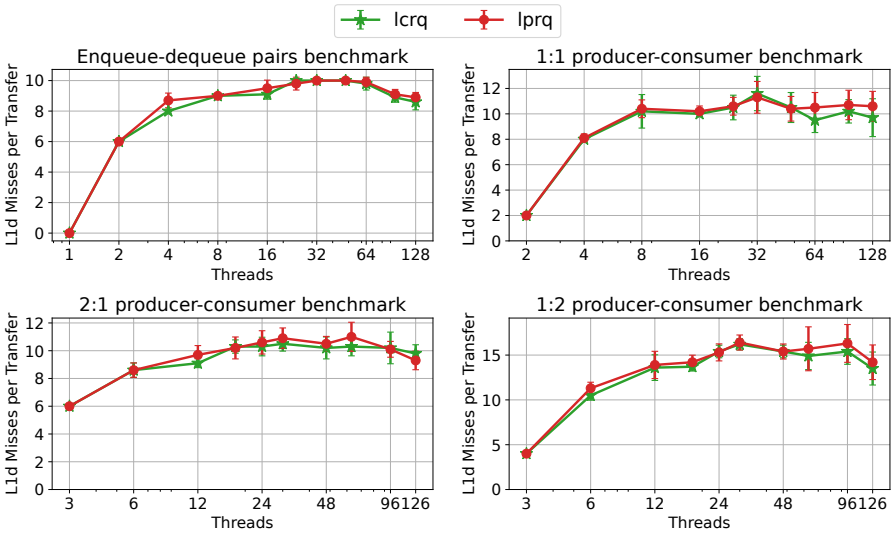


Figure 9. Numbers of L1d cache-misses per element transfer for LPRQ and LCRQ. **Lower is better.**