Fast Dual Ring Queues*

Joseph Izraelevitz Michael L. Scott

Technical Report #990

Department of Computer Science, University of Rochester
{jhi1,scott}@cs.rochester.edu

January 2014

Abstract

In this paper, we present two new FIFO dual queues. Like all dual queues, they arrange for `dequeue` operations to block when the queue is empty, and to complete in the original order when data becomes available. Compared to alternatives in which `dequeue`s on an empty queue return an error code and force the caller to retry, dual queues provide a valuable guarantee of fairness.

Our algorithms, based on Morrison and Afek's LCRQ from PPoPP'13, outperform existing dual queues—notably the one in `java.util.concurrent`—by a factor of four to six. For both of our algorithms, we present extensions that guarantee lock freedom, albeit at some cost in performance.

1 Introduction

A container object (e.g., a queue) that supports `insert` (`enqueue`) and `remove` (`dequeue`) methods must address the question: what happens if the element one wants to remove is not present? The two obvious answers are to wait or to return an error code (or signal an exception). The latter option leads to spinning in applications that really need to wait (`repeat until (x = try_dequeue()) != EMPTY`). The former option is problematic in nonblocking algorithms: how can a method be nonblocking if it sometimes blocks?

Dual data structures, introduced by Scherer and Scott [10], extend the notion of nonblocking progress to *partial* methods—those that must wait for a precondition to hold. Informally, a partial method on a nonblocking dual structure is redefined to be a total method that either performs the original operation (if the precondition holds) or else modifies the data structure in a way that makes the caller's interest in the precondition (its *request*) visible to subsequent operations. In a busy-wait implementation, the request might be a pointer to a thread-local flag on which the stalled thread is spinning; in a scheduler-based implementation, it might be a reference to a semaphore on which the thread is suspended.

Compared to a traditional method that returns an error code, forcing the caller to retry the operation in a loop, dual data structures offer two important advantages: First, the code of the structure itself obtains explicit control over the order in which stalled methods will complete when preconditions are satisfied (it might, for example, give priority to the operation whose request was the first to linearize). Second, requests can (and, in the original formulation, must) be designed in such a way that stalled threads impose no burden

*This work was supported in part by NSF grants CCF-0963759, CCF-1116055, CNS-1116109, CNS-1319417, and CCF-1337224, and by support from the IBM Canada Centres for Advanced Study.

on active threads—in particular, that they induce no memory contention. Scherer et al. [9] report that dual versions of the `java.util.concurrent SynchronousQueue` class improved the performance of task dispatch by as much as an order of magnitude.

The Java dual structures were based on the well-known M&S queue [6] and (for the "unfair" non-FIFO version) Treiber stack [11]. Since their original development, significantly faster concurrent queues have been devised. Notable among these is the linked concurrent ring queue (LCRQ) of Morrison and Afek [7]. While the linked-list backbone of this queue is borrowed from the M&S queue, each list node is not an individual element but rather a clever, fixed-length buffer dubbed a concurrent ring queue (CRQ). Most operations on an LCRQ are satisfied by an individual ring queue, and are extremely fast. The secret to this speed is the observation that when multiple threads contend with compare-and-swap (CAS), only one thread will typically succeed, while the others must retry. By contrast, when multiple threads contend with a fetch-and-increment (FAI) instruction, the hardware can (and indeed, does, on an x86 machine) arrange for all threads to succeed, in linear time [2]. By arbitrating among threads mainly with FAI, the CRQ—and by extension the LCRQ—achieves a huge reduction in memory contention.

Unfortunately, like most nonblocking queues, the LCRQ "totalizes" `dequeue` by returning an error code when the queue is empty. Threads that call `dequeue` in a loop, waiting for it to succeed, reintroduce contention, and their requests, once data is finally available, may be satisfied in an arbitrary (i.e., unfair) order. In this paper, we describe two dual versions of the LCRQ. In one version, all elements in a given CRQ are guaranteed to have the same "polarity"—they will all be data or all be requests ("antidata"). In the other version, a given CRQ may contain elements of both polarities. In effect, these algorithms combine the fairness of Scherer et al.'s M&S-based dualqueues with the performance of the LCRQ. Within a single multicore processor, throughput scales with the number of cores (synchronization is not the bottleneck). Once threads are spread across processors, throughput remains 4–6$\times$ higher than that of the M&S-based structure.

We review the operation of Morrison and Afek's LCRQ in Section 2. We introduce performance-oriented (but potentially blocking) versions of our new queues in Sections 3 and 4, and lock-free variants in Section 5. Preliminary performance results appear in Section 6. Finally, in Section 7, we summarize our findings and make suggestions for future work.

1.1 Related Work

In addition to the original dual queue and dual stack of Scherer and Scott [10], extant nonblocking dual data structures include the `java.util.concurrent Exchanger` of Scherer et al. [8] and a variety of *synchronous queues*, in which enqueuers and dequeuers both wait for a matching partner. Synchronous queue examples include the flat combining version of Hendler et al. [4] and the elimination-diffraction trees of Afek et al. [1]. The authors of the former report it to be faster than the latter under most circumstances.

Given the symmetry between enqueuers and dequeuers, the size of a synchronous queue is bounded by the number of threads, and the flat combining synchronous queue in particular is optimized to exploit this symmetry for throughput (rather than fairness). For purposes of comparison, our results in Section 6 consider an asymmetric dual queue based on the general paradigm of flat combining [3].

1.2 Notation and Terminology

Dualism "Dual" data structures take their name from the ability of the structure to hold either data or antidata. In a queue, where any datum can be used to satisfy any request, any quiescent state will always

find the structure empty, populated with data, or populated with antidata—a mix can only occur when the structure is in transition and some operation has yet to complete (i.e., to return or wait for data).

Dualism implies that a thread calling the public `enqueue` method may either enqueue data or dequeue antidata "under the hood." Likewise, a thread calling the public `dequeue` method may either dequeue data or enqueue antidata. When discussing dual algorithms, we will thus refer to the *polarity* of both threads and the structure, with a *positive* polarity referring to data, and a *negative* polarity referring to antidata. Thus, a thread calling the public `enqueue` method attains a positive polarity; it will either enqueue its data into a positive queue or dequeue antidata from a negative queue. Conversely, a thread calling the public `dequeue` method attains a negative polarity; it will either dequeue data from a positive queue or enqueue antidata into a negative queue. The only asymmetry—an important one—is that only negative threads will ever wait: calls to the public `enqueue` are always total.

Atomic Operations and Memory Model We prototyped out algorithms in C++ using the g++ compiler and C++11 atomic operations. Our code assumes the availability of (hardware-supported) compare-and-swap (CAS), though load-linked / store-conditional would also suffice. To resolve the ABA problem, we assume that we can store pointers in half the CAS width, either by forcing a smaller addressing mode or by using a double-wide CAS. We also assume the availability of a hardware fetch-and-increment (FAI) instruction that always succeeds.

Our pseudocode, as written, assumes sequential consistency for simplicity. For brevity, we exclude the code for garbage collection. Our C++ prototype uses store-load fences where necessary for correctness, and delays storage reclamation using hazard pointers [5]. Complete pseudocode can be found in an appendix; full source code will be made available prior to publication of this paper.

2 LCRQ Overview

We here provide a brief overview of the CRQ and LCRQ algorithms from which our single and multi polarity dual ring queues are derived. For a more complete treatment of the original algorithms, readers may wish to consult the paper of Morrison and Afek [7].

The LCRQ is a major enhancement of the M&S linked-list queue [6]. Instead of an individual data element, each of its list nodes comprises a fixed-size FIFO buffer called a Concurrent Ring Queue (CRQ). To ensure forward progress and permit unbounded growth, the CRQ provides "tantrum" semantics: at any point it can "throw a tantrum," closing the buffer to any further enqueues. Tantrums can occur if, for instance, the circular buffer is full or if an enqueuer repeatedly loses a race with another thread, and wishes to avoid starvation. The linked list of the LCRQ handles the tantrum cases by appending additional CRQs as needed, allowing the queue to grow without bound.

2.1 CRQ Algorithm

The CRQ (Figure 1) consists of a circular array of R elements. The array is indexed by two counters, `head` and `tail`, whose remainders modulo R are used to index into the array. Each element of the array, called a `Slot`, contains a data value (`val`) and the index (`idx`) of this data. When empty, the slot's `val` member is set to `null`. In all cases, we maintain the invariant $\forall$ `i` $\ $ `ring[i].idx` $\equiv$ `i` $\bmod R$. The `closed` bit on the `tail` counter is used to throw a tantrum and inhibit further enqueues.

```
1  tuple Slot{
2     bool safe; // 1 bit
3     int idx; // 31 bits
4     Object val; // 32 bits (a pointer or int)
5     // padded to cache line size
6  };
7  class CRQ { // fields are on distinct cache lines
8     int head; // 32 bits
9     <bool closed, int idx> tail; // 1 bit, 31 bits
10    CRQ* next;
11    Slot ring[R]; // initialize ring[i]=<1,i,NULL>
12 };
```

Figure 1: CRQ data types

2.1.1 Ideal Operation

Ideally, an enqueue operation simply:

1. performs a FAI on the `tail` counter to retrieve an index;
2. performs a mod operation on the index to identify a slot in the buffer (`ring`);
3. uses compare-and-swap (CAS) to insert the new data value into the chosen slot.

Conversely, the ideal dequeue operation:

1. performs a FAI on the `head` counter to retrieve an index;
2. performs a mod operation on the index to identify a slot;
3. retrieves the current value in the slot and uses CAS to switch it to `null`.

Enqueue and dequeue operations that use the same index are said to *correspond*; each dequeue must retrieve the data stored by its corresponding enqueue.

2.1.2 Enqueue Exceptions to Ideal Operation

While CRQ operations tend to perform ideally most of the time, there are two cases in which an `enqueue` cannot do so:

Case 1e There is already data in the slot. Since the buffer is circular, this may be data that was stored with a smaller index and has yet to be dequeued, indicating we have wrapped all the way around the buffer.

Case 2e Evidence suggests (see below) that the corresponding dequeue operation may already have run, implying that any data we enqueue would never be dequeued. In this case we cannot use the slot, even if it is empty.

In either of these cases, the enqueuer skips the index, and counts on the dequeuer to recover.

2.1.3 Dequeue Exceptions to Ideal Operation

There are also two cases in which a `dequeue` cannot perform ideally:

Case 1d The corresponding enqueue has not yet run. In this case the dequeue operation must leave some signal for its corresponding enqueue to prevent it from completing. When the enqueue operation reaches this index, it will be in Case 2e.

Case 2d The corresponding enqueue already ran, but skipped this index due to either Case 1e or Case 2e (the latter may occur because of Case 1d at some previous index that mapped to the same slot).

The exception cases for `dequeue` are identified by finding either the wrong index in a slot or a `null` value; in both cases we need to leave a signal for the corresponding `enqueue`:

- If the slot is empty, we increase its index (`idx`) to the dequeuer's index plus R.
- Alternatively, if the slot holds data, we clear the slot's `safe` bit.

These signals constitute the evidence seen in Case 2e: an enqueuer must skip any slot that has an index larger than its own or that is not marked as safe. Note that an erroneous signal (sent when an enqueuer has already skipped a slot) does not compromise correctness: if the slot is empty, the dequeuer's index plus R will be the right index for the next possible enqueuer; if the slot is full, a cleared `safe` bit will be ignored by any delayed but logically earlier dequeuer. In the worst case, an unsafe slot may become unusable for an indefinite period of time (more on this below).

2.1.4 Final Notes on the CRQ

The CRQ algorithm also provides code for several additional cases:

Queue may be full: An `enqueue` must fail when the CRQ is full. This case can be detected by observing that $\texttt{head} - \texttt{tail} > R$. In this case we throw a tantrum and close the queue.

Enqueues are otherwise unable to complete: If slots have become unsafe, or if an enqueue chases a series of dequeues in lock-step, the enqueue may fail to make progress even when the queue is not full. In this case the enqueuer can close the CRQ arbitrarily, forcing execution to continue to the next one in the larger LCRQ list.

Queue is empty: This case can be detected by observing that $\texttt{head} \geq \texttt{tail}$. Prior to returning and letting the caller retry, the we check to see whether `head` has moved a long way ahead of `tail`. If so, the next enqueue operation would end up performing a very large number of FAI operations to bring `tail` forward to match. A special `fixState()` routine uses CAS to perform the catch-up in a single step.

Slot can be made safe: Once a slot has been made unsafe, it generally remains unsafe, forcing it to be skipped by future enqueues. However, if `head` is less than the current enqueuer's index, that enqueuer knows that its corresponding dequeuer has not completed, and, if the slot is empty, it can enqueue into the unsafe slot, transitioning it to safe in the process.

```
13 class SPDQ {
14    CRQ* head, tail;
15 };
16 class SP_CRQ : CRQ {
17    bool sealed;
18    bool polarity;
19    bool seal();
20 };
```

Figure 2: SPDQ data types

2.2 LCRQ Algorithm

The LCRQ is a nonblocking FIFO linked list of CRQs. Enqueuing into the LCRQ is equivalent to enqueuing into the tail CRQ of the linked list, and dequeuing from the LCRQ is equivalent to dequeuing from the head CRQ. Beyond these simple behaviors, additional checks detect when to add new CRQs to the tail of the LCRQ and when to delete CRQs from the head. As both of our dual queues significantly rework this section of the original algorithm, we omit the details here.

3 Single Polarity Dual Ring Queue

In the original dual queue of Scherer and Scott [10] (hereinafter the "S&S dual queue"), the linked list that represents the queue always contains either all data or all antidata. In effect, queue elements represent the operations (enqueues [data] or dequeues [antidata]) of which there is currently an excess in the history of the structure.

By contrast, slots in a ring queue algorithm are pre-allocated, and acquire their polarity from the operation (`enqueue` or `dequeue`) that encounters them first. A ring-queue–based dual queue may thus contain elements of both polarities. Suppose, for example, that an enqueue operation acquires an index corresponding to the last remaining (most recently enqueued) antidata in the queue. It can complete and return despite the fact that some of its predecessor `enqueue`s, with smaller indices, are running slowly and have yet to return. A newly arriving `enqueue` must then insert data despite the fact that not-yet-fully-retrieved antidata still sits in other slots.

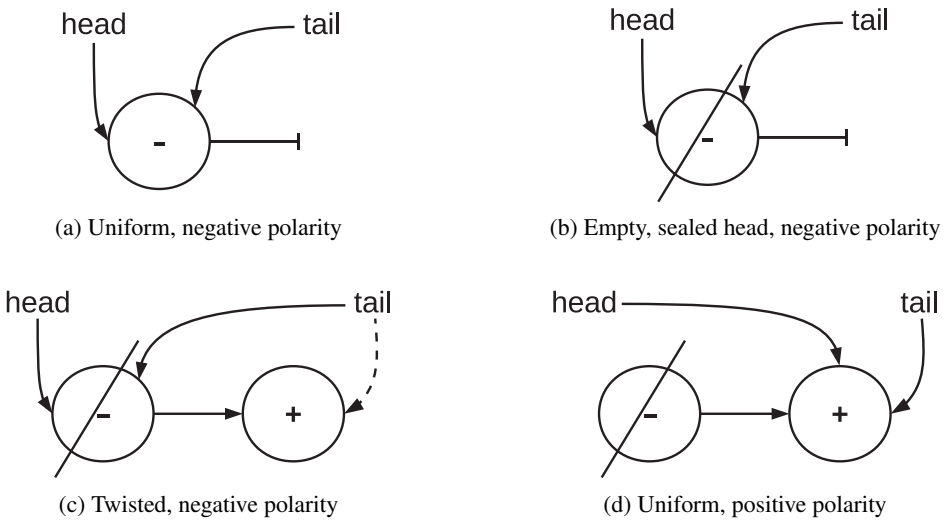
In our single polarity dual ring queue (SPDQ), each ring in the list has a single polarity—it can hold only data or only antidata. When the history of the queue moves from an excess of enqueues to an excess of dequeues, or vice versa, a new CRQ must be inserted in the list. This strategy has the advantage of requiring only modest changes to the the underlying CRQ. Its disadvantage is that performance may be poor when the queue is near empty and "flips" frequently from one polarity to the other.

3.1 Overview

To ensure correct operation, we maintain the invariant that all non-closed rings always have the same polarity. Specifically, we ensure that the queue as a whole is always in one of three valid states:

Uniform: All rings have the same polarity.

Twisted: All rings except the head have the same polarity, and the head is both empty and closed (*sealed*).



(a) Uniform, negative polarity

(b) Empty, sealed head, negative polarity

(c) Twisted, negative polarity

(d) Uniform, positive polarity

Figure 3: Flipping the polarity of the SPDQ

Empty: Only one ring exists, and it is both empty and closed.

Unless an operation discovers otherwise, it assumes the queue is in the uniform state. Upon beginning an operation, a thread will check the polarity of the head ring, and from there extrapolate the polarity of the queue. If it subsequently discovers that the queue is twisted, it attempts to remove the head and retries. If it discovers that the queue is empty, it creates a new ring, enqueues itself in that ring, and appends it to the list.

Since a public `enqueue` operation may end up dequeuing antidata internally, and a public `dequeue` method may end up enqueuing data, depending on the internal state of the queue, we combine these into a single "`denqueue`" operation, which handles both.

3.2 Modifications to the CRQ

Our `SP_CRQ` variant of the `CRQ` class incorporates three changes. The first is trivial: we add a boolean variable `polarity` that indicates the type of the ring.

Our second change ensures, when data "mixes" with existing antidata, that the waiting dequeuer (negative thread) is alerted ("satisfied"), and can return. This requires that the positive thread pass its data to its counterpart using the pointer found in the queue. This pointer dereference occurs within the code of the `SP_CRQ`.

Our third change adds a `seal` method and a boolean field `sealed`. The method attempts to atomically close the `SP_CRQ` if it is empty. Once `seal` succeeds, the `SP_CRQ` is closed, preventing additional enqueues, and the `SP_CRQ` is empty, allowing it to be removed from the linked list. Once the sealed method returns true, the ring may safely be removed from the queue. Our implementation of `seal` is based on the `fixState` method in the original CRQ.

3.3 Internal Enqueue

Once a thread has determined the polarity of the overall queue, it attempts the appropriate operation on the correct ring (either the head or tail).

```
21 Object SPDQ:dequeue() {
22    waiter* w = new waiter();
23    // waiter contains slot to place satisfying data
24    return denqueue(w, ANTIDATA);
25 }
26 void SPDQ:enqueue(Object val) {
27    return denqueue(val, DATA);
28 }
29 Object SPDQ:denqueue(Object val, bool polarity) {
30    SP_CRQ* h;
31    while (true) {
32       h = head; // read polarity of queue
33       if (h->polarity == polarity) {
34          v = internal_enqueue(h, val, polarity);
35          if (v == OK) return OK;
36       }
37       else {
38          v = internal_dequeue(val, polarity);
39          if (v != TOO_SLOW) return v;
40       }
41       // if internal operation failed,
42       // head has changed, so retry
43    }
44 }
```

Figure 4: SPDQ `denqueue`

In the internal enqueue operation (Fig. 5), we first read `tail` and verify both that it is indeed the tail of the list [6] and that the queue is not twisted. If one of these conditions does not hold, we correct it by moving `tail` or `head` accordingly. We then attempt to enqueue ourselves into the tail ring. If we succeed, we are done, and either return or wait, depending on our polarity. If we fail, indicating that the tail ring is closed, we create a new ring, enqueue into it, and append it to the list.

3.4 Internal Dequeue

In the internal dequeue operation (Fig. 6), we again verify we have the correct head. If so, we dequeue from it. If we succeed, we are finished. If not, the head ring is empty and should be removed. If the next ring exists, we simply swing the head pointer and continue there. If `head->next` is `null`, then the head ring is also the tail, and the entire queue is empty. If we can seal the head, we may flip the queue's polarity (Fig. 3). We flip the queue by adding a new ring of our own polarity, enqueuing ourselves into it, attaching to the head ring, and swinging the tail and head pointers. Prior to the final CAS, the queue is twisted. That is, the head ring is both closed and empty, but the remainder of the queue is of a different polarity. Any subsequent enqueue or dequeue will fix this state prior to continuing.

3.5 Forward Progress

Public `enqueue` (positive) operations inherit lock-free progress from the LCRQ algorithm [7]. In the worst case, an enqueuer may chase an unbounded series of dequeuers around a ring buffer, arriving at each slot too late to deposit its datum. Eventually, however, it "loses patience," creates a new ring buffer containing

```
45 Object SPDQ:internal_enqueue(SP_CRQ* h, Object val, bool polarity) {
46    SP_CRQ* t, next, newring;
47    while (true) {
48       t = tail;
49       // verify tail is the actual tail
50       if (t->next != NULL) {
51          next = t->next;
52          (void) CAS(&this->tail, t, next);
53          continue;
54       }
55       // verify correct polarity (detect twisting)
56       if (t->polarity != polarity) {
57          (void) CAS(&this->head, h, h->next);
58          return TWISTED;
59       }
60       // attempt enqueue on tail
61       if (t->enqueue(val) == OK) {
62          if (polarity == ANTIDATA) return spin((waiter*)val);
63          else return OK;
64       }
65       // else, the tail is closed
66       newring = new SP_CRQ(polarity);
67       newring->enqueue(val);
68
69       // append ring
70       if (CAS(&t->next,NULL,newring)) {
71          (void) CAS (&this->tail, t, newring);
72          if (polarity == ANTIDATA) return spin((waiter*)val);
73          else return OK;
74       }
75    }
76 }
```

Figure 5: SPDQ `internal_enqueue`

its datum, closes the current ring, and appends the new ring to the list. As in the M&S queue [6], the append can fail only if some other thread has appended a ring of its own, and the system will have made progress.

Because they may wait for data, public `dequeue` (negative) operations are more subtle. Scherer and Scott [10] model partial operations on dual data structures in terms of a potentially unbounded sequence of nonblocking operations. The first linearizes the request for data of the calling thread, T. The last operation (the "successful follow-up") linearizes T's receipt of that data. In between, unsuccessful follow-up operations perform only local memory accesses, inducing no load on other threads. Finally, the total method (in our case, an `enqueue`) that satisfies T's pending request must ensure that no successful follow-up operation by another waiting thread can linearize in-between it (the satisfying operation) and T's successful follow-up.

This final requirement is where the SPDQ as presented so far runs into trouble. A positive thread that encounters a negative queue must perform two key operations: remove the antidata from the queue and alert the waiting thread. In the S&S dual queue, it alerts the waiting thread first, by "mixing" its data into the antidata node. After this, *any* thread can remove the mixed node from the queue.

In the SPDQ as presented so far, an antidata slot is effectively removed from consideration by other

```
77  Object SPDQ:internal_dequeue(Object val, bool polarity) {
78     SP_CRQ* h, next, newring;
79     while (true) {
80        h = this->head;
81        // verify queue polarity didn't change
82        if (h->polarity == polarity) return TOO_SLOW;
83
84        // dequeue from head
85        v = h->dequeue(val);
86        if (v != EMPTY) return v;
87
88        // seal empty SP_CRQ so we can remove it
89        else if (!h->seal()) continue;
90
91        // at this point head SP_CRQ is sealed
92
93        // swing the head
94        if (h->next != NULL) {
95           (void) CAS(&this->head, h, h->next);
96        }
97        // or add a new tail and swing head to it
98        else {
99           newring = new SP_CRQ*(polarity);
100          newring->enqueue(val);
101          // append our new ring to list
102          // which will cause twisting
103          if (CAS(&h->next, NULL, newring)) {
104             (void) CAS(&this->tail, h, newring);
105             // swing head to fix twisting
106             (void) CAS(&this->head, h, h->next);
107             if (polarity==ANTIDATA) return ((waiter*)val)->spin();
108             else return v;
109          }
110       }
111    }
112 }
```

Figure 6: SPDQ `internal_dequeue`

threads the moment the corresponding enqueuer performs its FAI. Mixing happens afterward, leaving a window in which the enqueuer, if it stalls, can leave the dequeuer waiting indefinitely. In practice, such occurrences can be expected to be extremely rare, and indeed the SPDQ performs quite well, achieving roughly 85% of the throughput of the original LCRQ while guaranteeing FIFO service to pending requests (Sec. 6). In Section 5 we will describe a modification to the SPDQ that closes the preemption window, providing true lock-free behavior (in the dual data structure sense) at essentially no additional performance cost.

4 Multi Polarity Dual Ring Queue

In contrast to the SPDQ, the multi polarity dual ring queue (MPDQ) incorporates the flipping functionality at the ring buffer level, and leaves the linked list structure of the LCRQ mostly unchanged. The MPDQ

```
113 tuple MP_Slot {
114    bool safe; // 1 bit
115    bool polarity; // 1 bit
116    int idx; // 30 bits
117    int val; // 32 bits ( int or pointer )
118    // padded to cache line size
119 };
120 class MP_CRQ { // fields are on distinct cache lines
121    <bool closing, int idx> data_idx; // 1, 31 bits
122    <bool closing, int idx> antidata_idx; // 1, 31 bits
123    <bool closed, int idx> closed_info; // 1, 31 bits
124    MP_CRQ* next;
125    MP_Slot ring[R]; // initialize ring[i]=<1,i,NULL>
126 };
127 class MPDQ { // fields are on distinct cache lines
128    MP_CRQ* data_ptr;
129    MP_CRQ* antidata_ptr;
130 };
```

Figure 7: MPDQ data types

takes advantage of the pre-allocated nature of ring slots, discussed at the beginning of Section 3. For a dual structure, it does not matter whether data or antidata is first placed in a slot; either can "mix" with the other.

4.1 Overview

In their original presentation of the CRQ [7], Morrison and Afek began by describing a hypothetical queue based on an infinite array. Similar intuition applies to the MPDQ. Since we are matching positive with negative operations, each thread, on arriving at a slot, must check if its partner has already arrived. If so, it mixes its data (antidata) with the antidata (data) of the corresponding operation. If not, it leaves its information in the slot (a negative thread also waits).

To maintain the illusion of an infinite array, we must keep track of both data and antidata indices. These indices, as in the LCRQ, have two components:

1. which ring in the linked list to use
2. which ring slot to use

In contrast to the LCRQ case, we do not care which kind of operations are currently ahead of the other—only that a newly arriving operation can identify the correct ring and index to use. To accommodate these changes, we rename the indices and pointers (Fig. 7). We also add a bit to each slot to identify its polarity.

Figure 8 illustrates how `data_idx` and `antidata_idx` move past one another within the `MP_CRQ`. The biggest challenge in the new design is the need to stop both indices at a common slot when closing a ring; we discuss this challenge in Section 4.3.

4.2 MP_CRQ denqueue()

As the MPDQ is more symmetric than the SPDQ, we can combine all ring operations into a single `denqueue` method. Each thread, after choosing the ring on which to operate, obtains a slot from its respective index. It then attempts to dequeue its counterpart or, failing that, to enqueue itself.

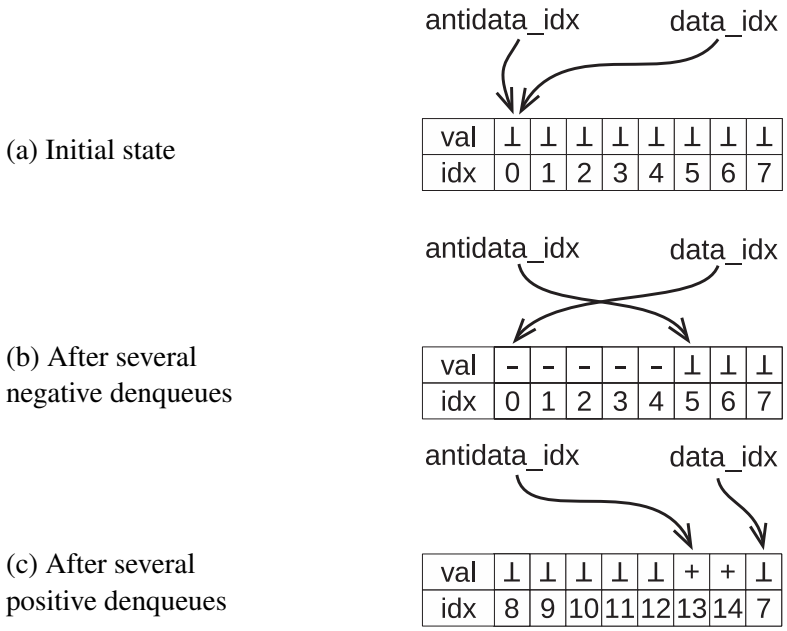


Figure 8: Flipping the polarity of the MP_CRQ

Note that at no point will the MP_CRQ ever return EMPTY, as any thread that believes the queue is empty would progress to the next slot and enqueue itself. For similar reasons, the `fixState` method of the original CRQ goes away, since one index passing another is considered normal operation.

4.2.1 Ideal Operation

Ideally, a denqueue operation:

1. performs a FAI on the `data_idx` or `antidata_idx` of the chosen ring, to obtain an index;
2. executes a mod operation on the index to identify a slot in the buffer (`ring`);
3. if arriving first, uses CAS to insert the new datum or antidatum into the array. If a negative thread, it waits for its corresponding operation;
4. if arriving second, retrieves the current value in the slot and uses CAS to switch it to `null`. If a positive thread, it satisfies the waiting negative thread.

4.2.2 MPDQ Exceptions to Ideal Operation

Like the LCRQ, the MPDQ tends to perform ideally most of the time. When it cannot do so, one of the following must have occurred:

Slot is occupied There is already data is the slot, but it is not from the corresponding thread. In this case, wrap-around has occurred. We handle it as in the LCRQ, by clearing the `safe` bit. Note that since both positive and negative threads attempt to enqueue, whichever sets the bit will successfully signal its counterpart.

Slot is unsafe In this case, our counterpart has possibly already arrived but was unable to use the slot, and cleared the `safe` bit. If we were to enqueue here, our counterpart might never receive it, so we simply skip the slot. In principle, if we verified that our counterpart has yet to begin, we could make the slot safe again, as in the LCRQ. Results of our testing suggest that this optimization is generally not worth the additional cache line miss to check the opposite polarity index. We therefore give up on unsafe slots permanently, falling back on the ability to add additional rings to the backbone list.

Ring is full This condition can be detected by observing that $|\texttt{data\_idx} - \texttt{antidata\_idx}| \geq R$. As in the LCRQ, we close the ring.

Livelock In a situation analogous to pathological cases in the LCRQ and SPDQ (as discussed in Section 3.5), it is possible in principle for the ring to be full or nearly full of non-`null` slots, each of which is pending mixing by some preempted thread. In this case, an active thread attempting to make progress may repeatedly mark slots unsafe. To preclude this possibility, each thread keeps a `starvation_counter`. If it fails to make progress after a set number of iterations, it closes the queue and moves on.

Counter overflow On a 32-bit machine, we can allocate only 30 bits to each index. This is a small enough number to make overflow a practical concern. Our implementation notices when the index nears its maximum, and closes the queue. Similar checks, not shown in the pseudocode, are used in all the tested algorithms.

4.3 Closing the MP_CRQ

As both indices in the MP_CRQ are used for enqueuing, they both must be closed simultaneously. Otherwise a thread that switches from dequeuing data to enqueuing antidata (or vice versa) might find itself in the wrong ring buffer. Our closing mechanism takes advantage of the observation that the actual index at which the queue is closed does not matter, so long as it is the same for both. Thus, we change the meaning of the spare bit for both the `data_idx` and `antidata_idx` from `closed` to `closing`. The new interpretation implies that the queue is *possibly* closed or *in the process of* closing. If a thread observes that the queue is closing, it cannot enqueue until it knows that the ring is closed for sure—and at what index. This determination is arbitrated by the `closed_info` tuple within the `MP_CRQ` (Fig. 7). The `closed` bit of that tuple indicates whether the queue is *actually* closed; `idx` indicates the index at which it closed.

To close the queue (Fig. 9), a thread first sets the `closing` bit on either `data_idx` or `antidata_idx`. It then calls `discovered_closing`, as does any thread that discovers a `closing` bit that has already been set. This method verifies that both indices are closing. Then, the method uses CAS to put the maximum of the two indices into `closed_info` and set its own `closed` bit. Some thread's CAS must succeed, at which point the queue is closed. Any thread with a larger index than that stored in `closed_info` must have been warned that its index may not be valid, and will not enqueue. Finally, some threads that find the queue to be closing may, in fact, still be able to enqueue as their indices are below that stored in `closed_info`. They return to their operations and continue.

4.4 MPDQ List

At the backbone level, the MPDQ must maintain pointers to the appropriate rings for both data and antidata. If any operation receives a closed signal from a ring, it knows it must move along to the next one. If no next ring exists, it creates one and append it to the list, in the style of the M&S queue. It first updates the

```
131 int MP_CRQ:discovered_closing(bool polarity) {
132    <bool closing, int idx> d_idx;
133    <bool closing, int idx> a_idx;
134
135    // check if already closed
136    if (closedInfo.closed==1)
137       return closedInfo.idx;
138
139    // set closing
140    antidata_idx.set_closing(1);
141    data_idx.set_closing(1);
142
143    // next read both indices and try to close queue
144    d_idx = data_idx;
145    a_idx = antidata_idx;
146    int closed_idx = max(d_idx.idx, a_idx.idx);
147    (void) CAS(&closedInfo, <0,0>, <1,closed_idx>);
148    return closedInfo.idx;
149 }
```

Figure 9: MPDQ closing

`next` pointer of the previous final ring, and then then swings the main `data_ptr` and/or `antidata_ptr` as appropriate.

4.5 Forward Progress

Like the SPDQ, the MPDQ as presented suffers from a "preemption window" in which a positive thread obtains an index, identifies its corresponding negative thread, but then stalls (e.g., due to preemption), leaving the negative thread inappropriately blocked and in a situation where no other thread can help it. The following section addresses this concern.

5 Lock freedom

The SPDQ and MPDQ, as presented so far, are eminently usable: they are significantly faster than the S&S dual queue (rivaling the speed of the LCRQ), and provide fair, FIFO service to waiting threads. To make them fully nonblocking, however, we must ensure that once a positive thread has identified a slot already in use by its corresponding thread, the negative thread is able to continue after a bounded number of steps by non-blocked threads.

For both algorithms we can close the "preemption window" by treating FAI as a mere suggestion to positive threads. Before they enqueue, they must verify that all smaller indices have already been satisfied. For purposes of exposition, we refer to the minimally indexed unsatisfied slot as the algorithm's *wavefront*. The changes are smaller in the SPDQ case and (as we shall see in Sec. 6) have almost no impact on performance. The changes are larger in the MPDQ case, with a larger performance impact.

5.1 SPDQ wavefront

As can be observed in the original CRQ algorithm, only dequeue operations can change the value of a given slot's index. If all dequeue operations must wait until the previous slot's index is changed, two simplifications occur:

1. No slot can become unsafe, since no dequeue operation can loop all the way around the ring before another dequeuer has a chance to finish its operation.

2. There is always exactly one indexing *discontinuity*, where the difference between neighboring indices is greater than one.

The discontinuity in indices (noticeable in Figure 8), indicates that a slot is ready and can be used to strictly order dequeue operations for the SPDQ. Since the preemption window only occurs when a positive thread dequeues from a negative ring, we can limit code changes to this single case.

5.2 MPDQ wavefront

In contrast to the SPDQ, the preemption window can manifest itself in the MPDQ during a denqueue operation. Since all positive operations must complete in order, and must wait (for nonblocking progress) until waiting partners of previous positive operations have been notified, enqueues also need to be considered when modifying the algorithm. With mixed ring polarities, a slot is ready either because it follows a discontinuity (the previous slot has been dequeued), or because the previous slot does not contain `null` (i.e., it has been enqueued).

Before a thread can complete an operation on an index (including returning the `CLOSED` signal), the index must be at the wavefront. If it is not, the thread, after waiting for a timeout, scans backward along the ring looking for the wavefront. Once the thread finds the wavefront, it attempts to operate on this slot, on the assumption that the thread to which the slot "belongs" is neglecting to do so. Any thread that finds its current slot already completed (by a successor that lost patience) must then continue forward. Since at any point, the number of active threads is equal to the difference from the wavefront to the head, all threads will eventually succeed.

6 Results

We evaluated our algorithms on a machine running Fedora Core 19 Linux on two six-core, two-way hyperthreaded Intel Xeon E5-2430 processors at 2.20GHz (i.e., with up to 24 hardware threads). Each core has private L1 and L2 caches; the last-level cache (15 MB) is shared by all cores of a given processor. Tests were performed while we were the sole users of the machine. Threads were pinned to cores for all tests. As we increased the number of threads, we used all cores on a given processor first, and then all hyperthreads on that processor, before moving to the second processor. Code was written in C++ and compiled using g++ 4.8.2 at the `-O3` optimization level.

6.1 Microbenchmark

Our goal in exercising these dual queues was to have as random access as possible to the queue without letting the system go into deadlock when all threads dequeue at once on an empty queue. To achieve this goal we developed a *hot potato* test inspired by the children's game. One thread, at the start of the test,

enqueues a special value, called the *hot potato*. For the duration of the test, all threads randomly decide to enqueue or dequeue. However, if they ever dequeue the *hot potato*, they must wait a set amount of time (generally a microsecond), then enqueue it back into the queue. Using this mechanism, we randomize access by each thread and allow the queue to flip back and forth between data and antidata, but avoid the deadlock case. We run the test for a fixed period of time (several seconds) and report performance as throughput.

For every queue, we ran five tests and took the maximum run. No large deviations among tests were noted for any of the queues.

6.2 Tested Algorithms

Our tests cover eight different queue algorithms:

SPDQ, MPDQ: The algorithms of Sections 3 and 4, respectively.

SPDQ lock-free, MPDQ lock-free: The nonblocking variants described in Section 5.

S&S Dual Queue: The algorithm of Scherer & Scott [10].

M&S Queue, LCRQ: The non-dual algorithms of Michael & Scott [6] and of Morrison & Afek [7], with an outer loop in which negative threads retry until they succeed in dequeuing data.

FC Dual: A best-effort implementation of a flat-combining dualqueue, using the methodology of Hendler et al. [3].

The various ring queues all use a ring size of 2048 elements. In our flat combining queue, each thread, instead of actually performing its desired operation on the structure, instead registers a request for the operation in a preassigned slot of a queue-specific array. Threads waiting for their operations to complete periodically compete for a global spin lock. Any thread that acquires the lock becomes a *combiner*. It makes a full pass through the request array, pairing up positive and negative operations, and storing excess data in a list for the next combiner. Other threads that see their operations have completed simply return.

To obtain a sense of fundamental hardware limits, we also ran a test in which all threads contend on a FAI counter, performing updates as fast as they can.

6.3 Blocking Variants

As shown in Figure 10, our new algorithms have throughput significantly higher than any existing dual queue. Qualitatively, their scalability closely follows that of the LCRQ, across the full range of thread counts, while additionally providing fairness for dequeuing threads. The SPDQ is perhaps 20% slower than the LCRQ on average, presumably because of the overhead of "flipping." The MPDQ is about 5% *faster* than the LCRQ on average, presumably because it avoids the empty check and the contention caused by retries in dequeuing threads.

All three algorithms (LCRQ, SPDQ, MPDQ) peak at twelve threads, where there is maximum parallelism without incurring chip-crossing overheads. The raw FAI test similarly scales well within a single chip. After moving to the second chip, all algorithms become increasingly constrained by the bus speed.

Interestingly, under some conditions, the new dual queues may even outperform the single integer FAI test. However, depending on the state of the queue, the active threads may spread their FAI operations over as many as three different integers (`head`,`tail`, and the `head` or `tail` of the next ring), distributing the bottleneck.

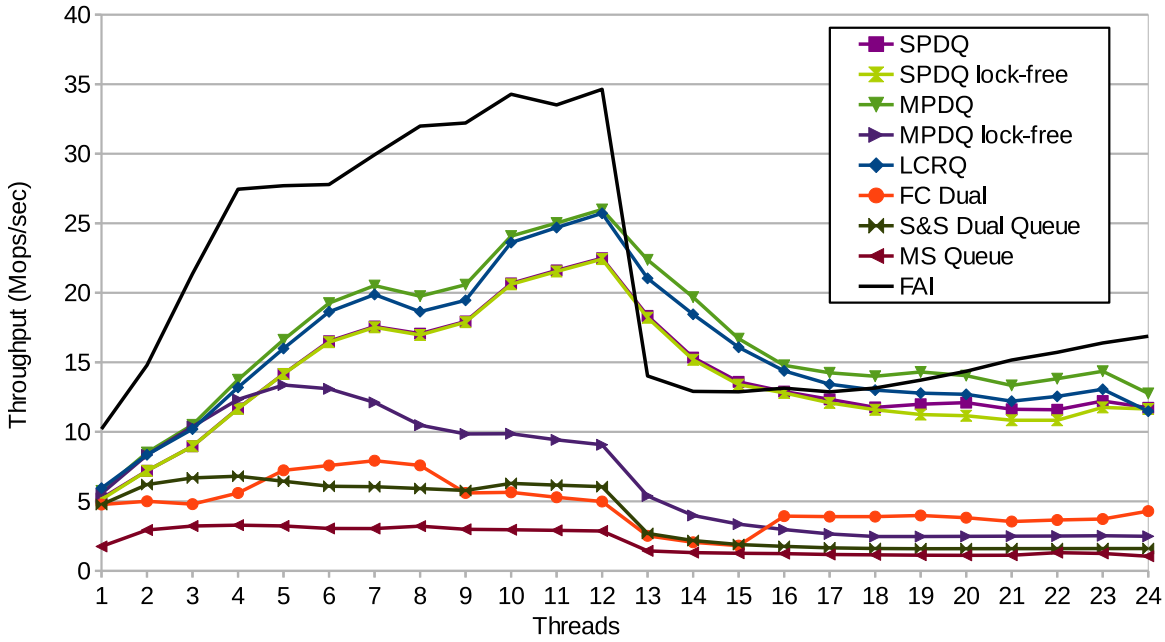


Figure 10: Performance on hot potato benchmark

Based on these tests, we recommend using the MPDQ in any application in which dequeuing threads need to wait for actual data.

6.4 Lock-free Variants

While the blocking versions of the SPDQ and MPDQ both outperform their lock-free variants, the performance hit is asymmetric. The lock-free SPDQ is almost imperceptibly slower than the blocking version. We expect this happens because the window closing code is only run rarely, when the queue's polarity is negative and many threads are waiting. While the window closing penalty per preemption incident is linear in the number of threads, the performance hit is isolated.

The MPDQ takes a drastic hit in order to close the window. Since we cannot isolate data from antidata within the MPDQ, every positive thread must "handshake," via flags in slots, with the previous positive thread, adding several cache misses to the critical path of the hot potato test.

6.4.1 Sensitivity of Results

We experimented with changes to a large number of timing parameters in the hot potato test. Several of these change the overall throughput, but none affects the relative performance of the tested algorithms. In general, larger ring sizes improve speed across the range of threads. Increasing the delay before re-enqueuing a hot potato slows everything down by causing queues to spend more time in a negative polarity.

7 Conclusion

In this paper, we have presented two fast algorithms that extend the LCRQ of Morrison and Afek [7] to provide fair, FIFO service to threads that are waiting to dequeue data. Our algorithms outperform existing dual queues by a factor of 4–6× and scale much more aggressively. We hope that these algorithms will be considered for use in thread pools and other applications that depend on fast inter-thread communication.

In their basic form, our algorithms are "almost nonblocking." We also presented fully lock-free variants. We believe the basic versions should suffice in almost any "real world" application. The MPDQ algorithm in particular is substantially simpler than the original LCRQ, and even outperforms it by a little bit. If one is unwilling to accept the possibility that a dequeuing thread may wait longer than necessary if its corresponding enqueuer is preempted at just the wrong point in time, the lock-free version of the SPDQ still provides dramatically better performance than the S&S dual queue.

References

[1] Y. Afek, G. Korland, M. Natanzon, and N. Shavit. Scalable producer-consumer pools based on elimination-diffraction trees. In *Proc. of the 16th Intl. Euro-Par Conf. on Parallel Processing*, pages 151–162, Ischia, Italy, Aug.–Sep. 2010.

[2] E. Freudenthal and A. Gottlieb. Process coordination with fetch-and-increment. In *Proc. of the 4th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 260–268, Santa Clara, CA, Apr. 1991.

[3] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proc. of the 22nd ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, pages 355–364, Santorini, Greece, June 2010.

[4] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Scalable flat-combining based synchronous queues. In *Proc. of the 24th Intl. Conf. on Distributed Computing (DISC)*, pages 79–93, Sept. 2010.

[5] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. on Parallel and Distributed Systems*, 15(8):491–504, Aug. 2004.

[6] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. of the 15th ACM Symp. on Principles of Distributed Computing (PODC)*, pages 267–275, Philadelphia, PA, May 1996.

[7] A. Morrison and Y. Afek. Fast concurrent queues for x86 processors. In *Proc. of the 18th ACM Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 103–112, Shenzhen, China, Feb. 2013.

[8] W. N. Scherer III, D. Lea, and M. L. Scott. A scalable elimination-based exchange channel. In *Wkshp. on Synchronization and Concurrency in Object-Oriented Languages*, San Diego, CA, Oct. 2005. In conjunction with OOPSLA '05.

[9] W. N. Scherer III, D. Lea, and M. L. Scott. Scalable synchronous queues. *Communications of the ACM*, 52(5):100–108, May 2009.

[10] W. N. Scherer III and M. L. Scott. Nonblocking concurrent data structures with condition synchronization. In *Proc. of the 18th Intl. Symp. on Distributed Computing (DISC)*, pages 174–187, Amsterdam, The Netherlands, Oct. 2004.

[11] R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, Apr. 1986.

A Pseudocode

This appendix contains pseudocode for all algorithms in the paper.

A.1 Original Linked Concurrent Ring Queue (LCRQ) [7]

```
tuple Slot {
   bool safe; // 1 bit
   int idx; // 31 bits
   Object val; // 32 bits ( int or pointer )
   // padded to cache line size
};
class CRQ { // fields are on distinct cache lines
   int head; // 32 bits
   <bool closed, int idx> tail; // 1, 31 bits
   CRQ* next;
   Slot ring[R]; // initialize ring[i]=<1,i,NULL>
};
class LCRQ {
   CRQ* head, tail;
};
```

```
CRQ:enqueue(int arg) {
   int h, t;
   Slot* slot;
   bool closed;
   bool safe;
   Object val; // 32 bit int or pointer
   int idx;
   int starvation_counter = 0;

   while (true) {
      <closed, t> := FAI(&self.tail);
      if (closed) return CLOSED;
      slot = &ring[t mod R];
      <safe,idx,val> = *slot;

      // verify slot is empty
      if (val == NULL) {
         // verify allowed index
         if ((idx <= t) &&
         // verify safe or fixable unsafe
         (safe==1 || head<=t)
         &&
         // then attempt enqueue
         CAS(slot, <safe,idx,NULL>, <1,t,arg>)) {
            return OK
         }
      }
      // enqueue on this index failed,
      // due to signal or occupied slot
      // check for full or starving
      h = head;
      starvation_counter++;
```

```
197         if (t-h >= R || starvation_counter>STARVATION) {
198            TAS(&tail.closed); // atomically close CRQ
199            return CLOSED;
200         }
201         // else we failed to enqueue and queue is not full
202         // so we skip this index and try the next
203      }
204   }
```

```
205   CRQ:dequeue() {
206      int h, t;
207      Slot* slot;
208      bool closed;
209      bool safe;
210      Object val;
211      int idx;
212
213      while (true) {
214         h = FAI(&head);
215         slot = &ring[h%R];
216         while (true) {
217            <safe,idx,val> = *slot;
218            // slot not empty
219            if (val != NULL) {
220               // attempt ideal dequeue
221               if (idx == h) {
222                  if (CAS(slot, <safe,h,val>, <safe,h+R,NULL>)) return val;
223               }
224               // failed to dequeue: signal unsafe
225               // to prevent corresponding enqueue
226               else if (CAS(slot, <safe, idx, val>,  <0, idx, val>))
227                  goto deq_fail;
228            }
229            // slot empty : failed to dequeue;
230            // signal slot via index to prevent corresponding enqueue
231            else if (CAS(slot, <safe,idx,NULL>, <safe,h+R,NULL>)) goto deq_fail;
232         } // end of inner while loop
233
234   deq_fail:
235         // we failed to dequeue at this index, so check for empty
236         <closed,t> = tail;
237         if (t <= h+1) {
238            fixState();
239            return EMPTY;
240         }
241         // else we failed to dequeue at this index
242         // and queue is not empty; try at next index
243
244      } // end of outer while loop
245   }
```

```
246   void CRQ:fixState() {
247      int h, t;
248
249      while (true) {
```

```
250         h = head; t = tail.idx;
251         if (h <= t)
252            // nothing to do as queue is consistent
253            return;
254         if (CAS(&tail, t, h)) // swing tail forward
255            return;
256      }
257   }
```

```
258   LCRQ:enqueue(Object x) {
259      CRQ* crq, newcrq;
260
261      while (true) {
262         crq = tail;
263         // make sure tail is updated fully
264         if (crq->next != NULL) {
265            (void) CAS(&tail, crq, crq->next)
266            continue;
267         }
268         // attempt enqueue
269         if (crq->enqueue(x) != CLOSED) return OK;
270
271         // if queue is closed, append new CRQ
272         newcrq = new CRQ();
273         newcrq->enqueue(x);
274         if (CAS(&crq.next, NULL, newcrq)) {
275            (void) CAS(&tail, crq, newcrq);
276            return OK;
277         }
278      }
279   }
```

```
280   LCRQ:dequeue() {
281      CRQ* crq;
282      Object v;
283
284      while (true) {
285         crq = head;
286         v = crq->dequeue()
287         if (v != EMPTY) return v;
288         if (crq->next == NULL) return EMPTY;
289
290         // swing head if the current head has remained empty
291         // (and is thus closed)
292         v = crq->dequeue();
293         if (v != EMPTY) return v;
294
295         (void) CAS(&head, crq, crq->next);
296      }
297   }
```

A.2 Single Polarity Dual Ring Queue (SPDQ)

```
class SPDQ{
   CRQ* head,tail;
   bool lock_free;
};
class SP_CRQ : CRQ {
   bool sealed;
   bool polarity;
};
class waiter() {
   int val;
   int spin() {
      while (val == NULL) {}
      return val;
   }
   bool satisfy(Object arg) {
      if (val != NULL) {
         val = arg;
         return true;
      }
      return false;
   }
};
```

```
SPDQ:dequeue() {
   waiter* w = new waiter();
   return denqueue(w, ANTIDATA);
}
SPDQ:enqueue(Object val) {
   return denqueue(val, DATA);
}
SPDQ:denqueue(Object val, bool polarity) {
   SP_CRQ* h;
   while (true) {
      h = head; // read polarity of queue
      if (h->polarity == polarity) {
         v = internal_enqueue(h, val, polarity);
         if (v == OK) return OK;
      } else {
         v = internal_dequeue(val, polarity);
         if (v != TOO_SLOW) return v;
      }
      // if internal operation failed, head has changed, so retry
   }
}
```

```
bool SP_CRQ:seal() {
   int h, t;

   while (true) {
      if (sealed) return true;

      h = head;
      t = tail;
      if (h<t) { // check if not empty
         return false; // if not, seal failed
```

```
351         }
352         // try to close while empty;
353         // if an enqueue occurs, tail moves, and CAS fails
354         if (CAS(&tail, t, <1,h>)) {
355            // CAS succeeded, so CRQ is
356            // closed and empty
357            sealed = true;
358            return true;
359         }
360      }
361   }
```

```
362   Object SPDQ:internal_enqueue(SP_CRQ* h, Object val, bool polarity) {
363      SP_CRQ* t, next, newring;
364      while (true) {
365         t = tail;
366         // verify tail is the actual tail
367         if (t->next != NULL) {
368            next = t->next;
369            (void) CAS(&this->tail, t, next);
370            continue;
371         }
372         // verify correct polarity (detect twisting)
373         if (t->polarity != polarity) {
374            (void) CAS(&this->head, h, h->next);
375            return TWISTED;
376         }
377         // attempt enqueue on tail
378         if (t->enqueue(val) == OK) {
379            if (polarity == ANTIDATA) return spin((waiter*)val);
380            else return OK;
381         }
382         // else, the tail is closed
383         newring = new SP_CRQ(polarity);
384         newring->enqueue(val);
385
386         // append ring
387         if (CAS(&t->next, NULL, newring)) {
388            (void) CAS(&this->tail, t, newring);
389            if (polarity == ANTIDATA) return spin((waiter*)val);
390            else return OK;
391         }
392      }
393   }
```

```
394   Object SPDQ:internal_dequeue(Object val, bool polarity) {
395      SP_CRQ* h, next, newring;
396      while (true) {
397         h = this->head;
398         // verify queue polarity didn't change
399         if (h->polarity == polarity) return TOO_SLOW;
400
401         // dequeue from head
402         if (polarity == DATA && this->lock_free)
403            v = h->dequeue_lock_free(val);
```

```
404         else
405            v = h->dequeue(val);
406         if (v != EMPTY) return v;
407
408         // seal empty SP_CRQ so we can remove it
409         else if (!h->seal()) continue;
410
411         // at this point head SP_CRQ is sealed
412
413         // swing the head
414         if (h->next != NULL)
415            (void) CAS(&this->head, h, h->next);
416         // or add a new tail and swing head to it
417         else {
418            newring = new SP_CRQ*(polarity);
419            newring->enqueue(val);
420            // append our new ring to list, which will cause twisting
421            if (CAS(&h->next, NULL, newring)) {
422               (void) CAS(&this->tail, h, newring);
423               // swing head to fix twisting
424               (void) CAS(&this->head, h, h->next);
425               if (polarity == ANTIDATA)
426                  return ((waiter*)val)->spin();
427               else return v;
428            }
429         }
430      }
431   }
```

A.3 Lock-free SPDQ

```
432   Object SP_CRQ:dequeue_lock_free(bool polarity, Object arg) {
433      <bool closed, int idx> h, t; // 1, 31 bits
434      Slot* slot;
435      bool closed;
436      bool safe;
437      int idx;
438      Object val;
439
440      h = FAI(&head);
441
442      bool paused = false;
443
444      while (true) {
445         slot = &this->ring[h.idx%R];
446         <safe,idx,val> = *slot;
447
448         // find the wave front
449
450         // move up because we're behind the wavefront
451         if (idx > h.idx) {
452            h.idx++;
453            continue;
454         }
```

```
455
456         // we're too far ahead and have lapped
457         if (idx < h.idx) {
458            h.idx = h.idx-R;
459            continue;
460         }
461
462         // now we know our index matches the slot
463
464         // verify we are on the wave front
465         if (h.idx!=0 && ring[(h.idx-1)%R].idx == h.idx-1) {
466            // we aren't, so wait a second and check again
467            if (!paused) {
468               usleep(1);
469               paused = true;
470               continue;
471            }
472            // we already timed out, so search backwards for the front
473            else {
474               h.idx--;
475               continue;
476            }
477         }
478         // now we know we're at the wavefront, so dequeue
479
480         // if slot is nonempty, dequeue
481         if (val != NULL) {
482            if (((waiter*)val)->satisfy(arg)) {
483               (void) CAS(&slot, <safe,h.idx,val>, <safe,h.idx+R,NULL>);
484               return OK;
485            } else { // someone beat us to the wait structure
486               (void) CAS(&slot, <safe,h.idx,val>, <safe,h.idx+R,NULL>);
487               h.idx++; // behind wavefront; move up
488               continue;
489            }
490         } else {
491            // if slot is empty, mark for counterpart
492            if (CAS(&slot, <safe,h.idx,val>, <safe,h.idx+R,NULL>)) {
493               <closed, t> = tail;
494               if (t <= h+1) {
495                  fixState();
496                  return EMPTY;
497               } else {
498                  h = FAI(&head);
499                  continue;
500               }
501            }
502         }
503      }
504   }
```

A.4 Mixed Polarity Dual Ring Queue (MPDQ)

```
505   tuple MP_Slot {
```

```
506     bool safe; // 1 bit
507     bool polarity; // 1 bit
508     int idx; // 30 bits
509     int val; // 32 bits ( int or pointer )
510     // padded to cache line size
511  };
512  class MP_CRQ { // fields are on distinct cache lines
513     <bool closing, int idx> data_idx;// 1, 31 bits
514     <bool closing, int idx> antidata_idx;// 1, 31 bits
515     <bool closed, int idx> closed_info;// 1, 31 bits
516     MP_CRQ* next;
517     MP_Slot ring[R]; // initialize ring[i] = <1,i,NULL>
518  };
519  class MPDQ { // fields are on distinct cache lines
520     MP_CRQ* data_ptr;
521     MP_CRQ* antidata_ptr;
522     bool lock_free;
523  };
```

```
524  Object MP_CRQ:denqueue(Object arg, bool polarity) {
525     <bool closed, int idx> p; // 1, 31 bits
526     <bool closed, int idx>* my_cntr;
527     <bool closed, int idx>* their_cntr;
528     MP_Slot* slot;
529     bool closed;
530     bool safe;
531     int idx;
532     int val;
533     bool slot_polarity;
534
535     int starvation_counter = 0;
536     int closeIdx = 0;
537
538     // determine which index to use
539     if (polarity == DATA) {
540        my_cntr = &data_idx;
541        their_cntr = &antidata_idx;
542     } else {
543        my_cntr = &antidata_idx;
544        their_cntr = &data_idx;
545     }
546
547     // do denqueue
548     while (true) {
549        <p.closing, p.index> = FAI(my_cntr);
550        // check for closing
551        if (p.closing == true) {
552           closeIdx = discovered_closing(p.idx, polarity);
553           if (closeIdx <= p.idx) return CLOSED;
554        }
555
556        slot = &ring[p.idx%R];
557
558        while (true) {
559           <safe,slot_polarity,idx,val> = *slot;
560
```

```
561           // if slot nonempty
562           if (val != NULL) {
563              // try to dequeue opposite
564              if (idx==p.idx && slot_polarity!=polarity) {
565                 if (CAS(&slot, <safe,p.idx,val,slot_polarity>,
566                       <safe,p.idx+R,NULL,slot_polarity>)) {
567                    if (polarity == ANTIDATA) return val;
568                    else ((waiter*)val)->satisfy(arg);
569                 } else continue;
570              }
571              // failed to dequeue:
572              // signal slot unsafe to prevent
573              // corresponding operation
574              else {
575                 if (CAS(&slot, <safe,idx,val,slot_polarity>,
576                       <0,idx,val,slot_polarity>)) {
577                    break;
578                 } else continue;
579              }
580           }
581           // if slot empty, try to enqueue self
582           else {
583              if (safe==1 || their_cntr->idx<=p.idx) {
584                 if (CAS(&slot, <safe,idx,NULL,slot_polarity>,
585                       <1,idx,arg,polarity>)) {
586                    return OK;
587                 } else continue;
588              }
589              else break; // unsafe, try the next index
590           }
591        } // end inner while loop
592
593        starvation_counter++;
594
595        // if fail to make progress, close the ring
596        if ((p.idx-their_cntr->idx >= R || starvation_counter > STARVATION)
597              && !p.closed) {
598           my_ptr->close();
599           closeIdx = discovered_closing(p.idx, polarity);
600           if (closeIdx <= p.idx) return CLOSED;
601        }
602     } // end outer while loop
603  }
```

```
604  Object MP_CRQ:denqueue(Object arg, bool polarity) {
605     CRQ* m, next, newring;
606     int v;
607     CRQ** my_ptr;
608
609     if (polarity == DATA) my_ptr = &data_ptr;
610     else my_ptr = &antidata_ptr;
611
612     while (true) {
613        m = *my_ptr;
614        // denqueue
615        if (polarity==DATA && lock_free)
```

```
616            v = m->denqueue_lock_free(arg, polarity);
617         else v = m->denqueue(arg, polarity);
618
619         // successful denqueue
620         if (v != CLOSED) {
621            if (polarity==ANTIDATA && v==OK) return spin((waiter*)val);
622            else if (polarity == DATA) return OK;
623            else return v;
624         }
625
626         // my_ptr is closed, move to next
627         if (m->next != NULL) (void) CAS(my_ptr, m, next);
628         else {
629            // if no next, add it
630            newring = new MP_CRQ();
631            v = newring->denqueue(arg);
632            if (CAS(&m->next, NULL, newring)) {
633               (void) CAS(my_ptr, m, newring);
634               if (polarity == ANTIDATA) return spin((waiter*)val);
635               else return OK;
636            }
637         }
638      }
639   }
```

```
640   int MP_CRQ:discovered_closing(bool polarity) {
641      <bool closing, int idx> d_idx;
642      <bool closing, int idx> a_idx;
643
644      // check if already closed
645      if (closedInfo.closed == 1) return closedInfo.idx;
646
647      // set closing
648      antidata_idx.set_closing(1);
649      data_idx.set_closing(1);
650
651      // next read both indices and try to close queue
652      d_idx = data_idx;
653      a_idx = antidata_idx;
654      int closed_idx = max(d_idx.idx, a_idx.idx);
655      (void) CAS(&closedInfo, <0,0>, <1,closed_idx>);
656      return closedInfo.idx;
657   }
```

A.5 Lock-free MPDQ

```
658   Object MP_CRQ:denqueue_lock_free(Object arg, bool polarity) {
659      <bool closed,int idx> p; // 1, 31 bits
660      <bool closed,int idx>* my_cntr;
661      <bool closed,int idx>* their_cntr;
662      MP_Slot* slot;
663      bool closed;
664      bool safe, safe_pr;
665      int idx, idx_pr;
```

```
666     int val, val_pr;
667     bool slot_polarity, slot_polarity_pr;
668
669     MP_Slot* slot_prev;   //slot : pointer to Slot
670     int starvation_counter = 0;
671     int closeIdx = 0;
672     bool paused=false;
673
674     // get heads
675     my_cntr = &data_idx;
676     their_cntr = &antidata_idx;
677
678     p = FAI(my_cntr);
679     while (true) {
680        // check for closed
681        if (p.closed==1 || data_idx.closed==1) {
682           closeIdx = discovered_closing(p.idx, polarity);
683           if (closeIdx <= p.idx) return CLOSED;
684        }
685        // if fail to make progress
686        // close queue as nearing full to ensure the
687        // closed index is represented by an actual slot
688        if ((data_idx.idx-anti_data.idx >= (R-2*MAX_THREADS)
689              || starvation_counter > STARVATION)
690              && !p.closed) {
691           data_idx.close();
692           closeIdx = discovered_closing(data_idx.idx, polarity);
693           if (closeIdx <= p.idx) return CLOSED;
694        }
695
696        slot = &ring[p.idx%R];
697        <safe,slot_polarity,idx,val> = *slot;
698        starvation_counter++;
699
700        // find wavefront
701        if (p.idx < idx) { // behind wavefront
702           p.idx++;
703           continue;
704        }
705        if (idx < p.idx) { // lapped ahead of wavefront
706           p.idx = p.idx-R;
707           continue;
708        }
709
710        // verify at wavefront
711        slot_prev = ring[(p.idx-1)%R];
712        <safe_pr,slot_polarity_pr,idx_pr,val_pr> = *slot_prev;
713        if (p.idx != 0 && idx_pr == (idx-1) &&
714              ((slot_polarity_pr == ANTIDATA && val_pr != NULL)
715              || val_pr == NULL)) {
716           // not ahead; wait and check again
717           if (!paused) {
718              usleep(1);
719              paused = true;
720              continue;
721           } else {
```

```
722             // already timed out; search backwards
723             p.idx--;
724             continue;
725          }
726       }
727
728       // on the wavefront, can operate
729
730       // if slot nonempty
731       if (val != NULL) {
732          // try to dequeue
733          if (idx==p.idx && slot_polarity!=polarity) {
734             if (((waiter*)val)->satisfy(arg)) {
735                (void) CAS(&slot, <safe,p.idx,val>, <safe,p.idx+R,NULL>);
736                return OK;
737             } else { // someone beat us to the wait structure
738                (void) CAS(&slot, <safe,p.idx,val>, <safe,p.idx+R,NULL>);
739                p.idx++;
740                continue;
741             }
742          }
743          // if slot wrong polarity, we lost race to enqueue
744          else if (slot_polarity == polarity) {
745             p.idx++;
746             continue;
747          }
748       }
749
750       // if slot empty, try to enqueue self
751       else {
752          if (safe==1 || antidata_idx.idx<=p.idx) {
753             // enqueue
754             if (CAS(&slot, <safe,p.idx,val>, <1,p.idx,arg>)) return OK;
755          }
756          // else something got in my way -
757          // either my counterpart or competition
758       }
759    }// end outer while loop
760 }// end dequeue
```