Generality and Speed in Nonblocking Dual Containers

JOSEPH IZRAELEVITZ and MICHAEL L. SCOTT, University of Rochester

Nonblocking dual data structures extend traditional notions of nonblocking progress to accommodate partial methods, both by bounding the number of steps that a thread can execute after its preconditions have been satisfied and by ensuring that a waiting thread performs no remote memory accesses that could interfere with the execution of other threads. A nonblocking dual container, in particular, is designed to hold either data or *requests*. An *insert* operation either adds data to the container or removes and *satisfies* a request; a *remove* operation either takes data out of the container or inserts a request.

We present the first *general-purpose* construction for nonblocking dual containers, allowing any nonblocking container for data to be paired with almost any nonblocking container for requests. We also present new custom algorithms, based on the LCRQ of Morrison and Afek, that outperform the fastest previously known dual containers by factors of four to six.

CCS Concepts: • **Theory of computation** → **Concurrent algorithms**; • **Computing methodologies** → **Concurrent algorithms**; • **General and reference** → *Design*; Performance;

Additional Key Words and Phrases: Dual containers, nonblocking data structures, synchronization

ACM Reference Format:
Joseph Izraelevitz and Michael L. Scott. 2017. Generality and speed in nonblocking dual containers. ACM Trans. Parallel Comput. 3, 4, Article 22 (March 2017), 37 pages.
DOI: http://dx.doi.org/10.1145/3040220

1. INTRODUCTION

A concurrent container object (e.g., a queue) that supports insert (enqueue) and remove (dequeue) methods must address this question: what happens if the element one wants to remove is not present? The two obvious answers are to wait for data within the remove method or to return an error code (or signal an exception). The former option appears—at least on the face of it—to make remove a blocking operation; the latter forces a thread that really needs to wait to spin *outside* the container and to perform a potentially unbounded number of fruitless remove operations that can lead to significant contention.

Dual data structures [Scherer and Scott 2004] extend the definition of nonblocking programs to accommodate *partial* methods—those that must wait for a precondition to hold. Informally, a partial method is replaced with a total *request* method that either performs the original operation (if the precondition holds) or modifies the data structure in a way that makes the caller's interest in the precondition (its *request*) visible to subsequent operations.

This work was supported in part by NSF grants CCF-0963759, CCF-1116055, CNS-1116109, CNS-1319417, CCF-1337224, and CCF-1422649, and by support from the IBM Canada Centres for Advanced Study.
Authors' addresses: J. Izraelevitz and M. L. Scott, Department of Computer Science, University of Rochester, Rochester, NY 14627-0226, USA; emails: {jhi1, scott}@cs.rochester.edu.

DOI: http://dx.doi.org/10.1145/3040220

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 data structure itself obtains explicit control over the order in which requests will complete when preconditions are satisfied (e.g., it might 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 threads with pending requests impose no burden on active threads: each waiting thread can spin, for example, on a separate local flag.

Related Work. In addition to the original dual queue and dual stack of Scherer and Scott [2004] and Scherer et al. [2005] describe a dual *exchanger* and both "fair" and "unfair" *synchronous queues* [Scherer et al. 2009] (the "unfair" version is actually a stack). Threads in an exchanger "pair up": each thread provides an object to a common exchange method and returns with the object provided by another thread. In a synchronous queue, producers and consumers pair up: enqueue and dequeue methods both wait for a matching operation of the opposite polarity. Both the exchanger and the synchronous queues are included in the java.util.concurrent standard library; Scherer et al. [2009] report that their use in the Executor runtime package improved the performance of task dispatch by as much as an order of magnitude.

Other synchronous queues include the flat combining algorithm of Hendler et al. [2010a] and the elimination-diffraction trees of Afek et al. [2010]. The authors of the former report it to be faster than the latter under most circumstances. Given the symmetry between producers and consumers, 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). We focus in this article on traditional "asymmetric" (thus, also nonsynchronous) queues, in which dequeue operations wait for a matching but not vice versa. As a basis for comparison in benchmarking tests (Section 5), we have constructed an asymmetric flat combining queue using the methodology of Hendler et al. [2010a].

A nonblocking, nonsynchronous queue can be of bounded size if producers fail when it is full, just as consumers fail when it is empty. Such queues are often used for message passing. Early (nondual) examples were designed to leverage the "combinability" of fetch-and-add (FAA) or fetch-and-increment (FAI) instructions [Freudenthal and Gottlieb 1991; Gottlieb et al. 1983], thereby avoiding serial bottlenecks. More recent dual examples have been optimized for modern processors; these include the PTLQueue of Dice [2014] and the BNPVB family of Pasetto et al. [2012]. None of these algorithms is nonblocking: each has a timing "window" in which a delay in one thread can obstruct the progress of others, despite the fact that progress should, in principle, be possible.

Past work has also explored *unbounded* FAA/FAI–based queues. An early (blocking) example can be found in the work of Wilson [1988]; more recent examples include the blocking HTQueue of Orozco et al. [2012] and the nonblocking (lock-free) LCRQ of Morrison and Afek [2013]. Further citations and commentary can be found in a recent article by Dice [2014]. None of these algorithms is dual: each requires a method with a false precondition to back out and retry. We will return to unbounded nonblocking FAA/FAI–based queues in Section 4, where we show how to make them dual.

After a brief review of dualism in Section 2, the remainder of this article discusses new implementations of dual containers. Section 3 (earlier versions of which appeared as a brief announcement [Izraelevitz and Scott 2014a] and a technical report [Izraelevitz and Scott 2014c]) introduces a generic construction for building dual containers out of other concurrent containers. Section 4 (also a previous brief announcement [Izraelevitz and Scott 2014b] and a technical report [Izraelevitz and Scott 2014d]) presents FIFO dual queues that significantly improve on the performance

of prior structures. We present performance results in Section 5 and conclude in Section 6. Source code is available at cs.rochester.edu/research/synchronization/.

2. BACKGROUND AND TERMINOLOGY

"Dual" data structures take their name from the ability of the structure to hold either data or, alternatively, *requests* for data (*antidata*).

In a queue, where any available datum is acceptable to a dequeue-er, a quiescent state will always find the structure empty, populated with data, or populated with antidata. A mix of data and antidata can occur only when the structure is in transition and some operation has yet to complete (i.e., to return or to wait for data).

Dualism implies that a thread calling the public insert method may either insert data or remove antidata "under the hood." Likewise, a thread calling the public remove method may either remove data or insert 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 insert method has a positive polarity; it will either insert its data into a positive container or remove antidata from a negative container. Conversely, a thread calling the public remove method has a negative polarity; it will either remove data from a positive container or insert antidata into a negative container. The only asymmetry—an important one—is that (in this article at least) only negative threads will ever wait: calls to the public insert are always total. Positive and negative operations are said to *correspond* when the former provides the datum for the latter; at the linearization point of whichever operation happens last, the two operations are said to *mix*.

In the framework of Scherer and Scott [2004], a nonblocking dual container object should export three public methods, all of which are total. The insert method places data into the container or, if antidata is available, satisfies and removes it. The remove_request method removes a datum from the container or, if data are not available, inserts a *request* (antidatum) instead. Either way, remove_request returns a unique *ticket* value that corresponds to the datum or antidatum. The remove_followup method takes a ticket as argument. If the ticket corresponds to an already removed datum, the method returns this datum and is said to be *successful*. If the ticket corresponds to an antidatum that has not yet been satisfied, the operation returns a distinguished NULL value and is said to be *unsuccessful*. If the ticket corresponds to an antidatum that *has* been satisfied, the operation returns the datum used to satisfy it and is again said to be *successful*.

In practice, it is generally desirable to provide a composite remove operation whose internal behavior comprises a remove_request followed by a potentially unbounded sequence of remove_followups, all but the last of which (if there is one) are unsuccessful. To be considered correct, the remove_followup sequence must satisfy two key properties. First, each unsuccessful remove_followup must refrain from performing any remote memory accesses—it must not read any location that is written concurrently by any other thread or write any location that is read or written concurrently by any other thread. Second, when a matching insert operation occurs, a waiting thread must wake up "right away." That is, suppose that insert operation I in thread t matches successful remove_followup operation S in thread u. No other operation (in particular, neither an unsuccessful remove_followup in u nor a successful remove_followup in any other thread) is permitted to linearize between I and S.

3. GENERIC CONSTRUCTION FOR NONBLOCKING DUAL CONTAINERS

To the best of our knowledge, all published nonblocking dual containers have shared a common design pattern: at any given time, the structure holds either data or antidata, depending on whether there have been more inserts or removes in the set of

operations completed to date (in some cases, the structure may also contain already *satisfied* antidata, whose space has yet to be reclaimed). As the balance of completed operations changes over time, the structure "flips" back and forth between the two kinds of contents.

While successful, this design strategy has two significant drawbacks. First, adapting an existing container to make it a dual structure is generally nontrivial: not only must an operation that flips the structure linearize with respect to all other operations, if it satisfies a request, it must also remove the antidatum and unblock the waiting thread as a single atomic operation (otherwise, there will be a window in which a delay in the positive thread can block the negative thread indefinitely). Second, since the same structure is used to hold either data or antidata, the same ordering discipline will generally be applied to both. Scherer [2005] designed, but did not publish, what he called "quacks" and "steues," with First In, First Out (FIFO) ordering for data and Last In, First Out (LIFO) ordering for antidata, or vice versa. These were implemented with a single structure—a linked list—and a change of convention on whether to insert at the head or the tail. Unfortunately, this approach does not generalize to other container structures.

In this section, we introduce a new construction that eliminates the drawbacks of previous approaches by joining a pair of containers—one for data and one for antidata. Any existing concurrent container can be used for the data side; on the antidata side, we require that the remove method be partitioned into a peek method and a separate remove_conditional. We introduce our construction in Section 3.1. It requires no hardware support beyond the usual load, store, and compare_and_swap (CAS). Section 3.2 presents safety and liveness proofs, demonstrating that the construction correctly merges the semantics of the constituent structures, preserves obstruction freedom, and avoids any memory contention caused by waiting threads. Section 5.1 presents microbenchmark results, showing reasonable performance for a variety of combinations of data and antidata structures.

3.1. The Generic Dual Construction

As suggested in the preceding paragraph, we build a nonblocking dual container using two underlying "subcontainers": one for data and one for antidata. We maintain the invariant that, at any given linearization point, at most one of the underlying subcontainers is nonempty. Thus, in a positive operation (i.e., a public insert), we may *satisfy* and remove an element from the antidata subcontainer, allowing the thread that is waiting on that element to return. Alternatively, we may verify that the antidata subcontainer is empty and instead insert into the data subcontainer. (The trick, of course, is to obtain a single linearization point for this logically two-part operation.) In a negative operation, we either remove and return an element from the data subcontainer or verify that the data subcontainer is empty and insert into the antidata subcontainer.

The outer container is said to have positive polarity when its data subcontainer is nonempty; it has negative polarity when its antidata subcontainer is nonempty. Its polarity is neutral when both subcontainers are empty.

3.1.1. Supported Subcontainers. We assume a conventional API for the data subcontainer. The insert method takes a datum (typically a pointer) as argument, and returns no useful value. The remove method takes no argument; it returns either a previously inserted datum or an EMPTY flag. More specifically, we assume that the data subcontainer maintains a total order $<^+$ on its elements, such that in any realizable linearization order in which all the arguments to insert are unique, (1) remove returns EMPTY whenever the number of previous remove operations equals or exceeds the number of previous insert operations; (2) if remove returns a, then there exists a

previous insert operation that provided a as argument, there is no previous remove operation that returned a, and there is no b such that b was provided by a previous insert operation, b was not removed by any previous remove operation, and $b <^+ a$. That is, remove always returns the smallest datum present under $<^+$.

For the antidata subcontainer, we assume a similar insert method, which takes an antidatum as argument, and a similar total order $<^-$ on elements. We require, however, that removal be partitioned into a pair of methods. The peek method takes no argument; it returns an antidatum and a special *key*. At the time of a peek call that returns (v, k), v must be the smallest antidatum present under $<^-$. The key can then be passed to a subsequent call to remove_conditional. That call will remove the value associated with its key argument so long as the value v associated with k is still the smallest under $<^-$; otherwise, remove_conditional has no effect. In our executable code, remove_conditional returns a Boolean indicating whether v was actually removed and thus can be garbage-collected; we ignore this value in the pseudocode.

Assuming that the operations of the subcontainers are linearizable and nonblocking, we will show in Section 3.2 that the outer container is obstruction free. As it turns out, many nonblocking container objects can be converted easily to support peek and remove_conditional. We experimented with converted versions of the Treiber stack [Treiber 1986], the M&S queue [Michael and Scott 1996], and the H&M sorted list [Harris 2001; Michael 2002]. Appendix A gives pseudocode for the converted Treiber stack.

3.1.2. Placeholders. As noted earlier, our construction requires that we be able to verify that one subcontainer is empty and insert into the other, atomically. To accomplish this task, we introduce the concept of *placeholders*. Instead of actually storing data or antidata in a subcontainer, we instead store a pointer to a placeholder object. Each placeholder contains a datum or an antidatum, together with a small amount of metadata. Specifically, a placeholder can be in one of four states: INVALID, ABORTED, VALID, or SATISFIED. An INVALID placeholder indicates an ongoing operation—the associated thread has begun to check for emptiness of the opposite subcontainer, but has not yet finished the check. An ABORTED placeholder indicates that the associated thread took too long in its emptiness check and any information it has regarding the status of the opposite subcontainer may be out of date. A VALID placeholder indicates that the associated thread has completed its emptiness check successfully and has inserted into the subcontainer of like polarity. Finally, a SATISFIED placeholder indicates that the associated data or antidata has been "mixed" with antidata or data from the corresponding operation.

On beginning a positive or negative operation on the outer container (Figure 1), we first store an INVALID placeholder in the subcontainer of like polarity. We then check for emptiness of the opposite subcontainer by repeatedly removing elements. If we find a VALID placeholder, we mix it with our own data or antidata, transition it from VALID to SATISFIED, and return, leaving our own INVALID placeholder behind. If we find an INVALID placeholder, we abort it, indicating that it has been logically removed from its subcontainer and that any information the owning thread may have had regarding the polarity of the outer container is now out of date. Finally, if we discover that the opposite subcontainer is empty, we can go back to our stored placeholder and attempt to validate it, completing our operation. If we find, however, that our placeholder has been aborted, then some thread of opposite polarity has removed us from our subcontainer. If that left our subcontainer empty, the other thread may have validated its own placeholder and returned successfully. We must therefore retry our operation from the beginning. The possibility that two threads, running more or less in tandem, may abort each other's placeholders—and both then need to retry—is why our construction is merely obstruction free, rather than lock free.

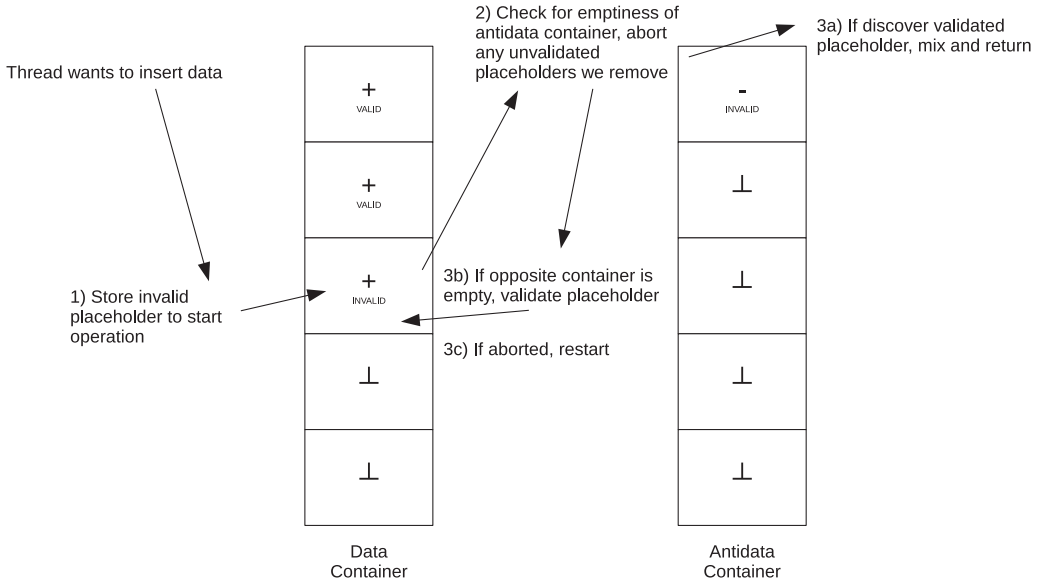

Fig. 1. Execution of the generic dual container.

3.1.3. Wakeup. One detail remains to be addressed. While a partial method of a dual data structure may block when a precondition is not met, the definitions of Section 2 place strict limits on this blocking. In particular, if a thread inserts a datum into a container, and another thread is waiting for that datum, the waiting thread must wake up "right away." (This requirement is formalized as Theorem 3 in Section 3.2.) The description of our construction presented earlier does not yet meet this requirement: it admits the possibility that a positive thread will remove a placeholder from the negative subcontainer and then wait an unbounded length of time (e.g., due to preemption by the operating system) before actually satisfying the placeholder and allowing its owner to return. In the meantime, an unbounded number of other operations (of either polarity) may complete.

We term this issue the *preemption window*. We close it with the peek and remove_conditional methods. A positive thread, instead of simply removing a placeholder from the negative subcontainer, first "peeks" at that placeholder (the one that is least under $<^-$) and publicly posts its intent—to satisfy the placeholder with a specific value—as the *active request*. After posting its request, the active thread attempts to fulfill it by satisfying the placeholder. Only then does it remove the placeholder from the subcontainer and take down the public request. Any other positive thread must check the *active request* field and help complete the request of the active thread before posting its own request. By updating placeholders while they are still in the negative subcontainer, we order all waiting threads, guaranteeing that they are able to return without any further delay.

As we shall see in Section 5.1, closing the preemption window has a nonnegligible negative impact on performance. If all one wants in practice is a fast shared buffer, a "not quite nonblocking" version of our construction, without the active request field, may actually be preferred.

3.1.4. Pseudocode. Pseudocode for the generic dual container appears in Figure 2. For convenience, we assume a sequentially consistent memory model; the ordering annotations for relaxed models are tedious but straightforward.

```
1   tuple Placeholder {
2     enum {INVALID, VALID, ABORTED, SATISFIED};
3     Object* val = NULL; // 32 bits
4     int state = INVALID; // 2 bits
5
6     // needed for nonblocking; otherwise ignored:
7     Request* req = NULL; // 30 bits
8           // (target aligned to 4 bytes)
9
10    Placeholder(Object* o) {val = o; req = NULL;}
11    bool satisfy(Object* v, Request* req) {
12      return CAS(this, ⟨NULL, VALID, NULL⟩,
13          ⟨v, SATISFIED, req⟩);
14    }
15  };
16
17  class GDual {
18    container* subcon[2];
19    bool nonblocking;
20    const DATA= 0, ANTI= 1;
21    Request* activeReq; // for nonblocking
22
23    GDual(container *dc, *ac) {
24      subcon[DATA] = dc;
25      subcon[ANTI] = ac;
26      activeReq = NULL;
27    }
28  };
29
30  Object* GDual:remove() {
31    return remsert(NULL, ANTI);
32  }
33
34  Object* GDual:insert(Object* val) {
35    return remsert(val, DATA);
36  }
37
38  Object* GDual:remsert(Object* o, bool polarity) {
39    Placeholder* ph;
40    bool nb = nonblocking && (polarity == DATA);
41    Object* rtn = EMPTY;
42
43    // allocate placeholder
44    ph = new Placeholder(o);
45
46    // actual transaction attempt
47    while (rtn == EMPTY) {
48      // begin transaction by emplacing placeholder
49      subcon[polarity]→insert(ph);
50
51      // do empty check on opposite
52      rtn = doOppositeCheck(ph, polarity, nb);
53      if (rtn ≠ EMPTY) {
54          // satisfied opposite, so finished
55        return rtn;
56      }
57
58      // empty check failed, so now we try
59      // to validate our placeholder
60      if (CAS(ph, ⟨o, INVALID, NULL⟩,
61          ⟨o, VALID, NULL⟩)) {
62        if (polarity == DATA) return OK;
63        // else spin waiting for data
64        while (ph→state ≠ SATISFIED) {}
65        return ph→val;
66      }
67      // else we couldn't validate our placeholder
68      // which means someone aborted us and
69      // we need to retry
70
71      ph = new Placeholder(o);
72    } // end while empty
73  }
74
75  Object* GDual:doOppositeCheck
76   (Placeholder* ph, bool polarity, bool nb) {
77    if (!nb)
78      return oppositeCheck(ph, polarity);
79    else
80      return oppositeCheckNB(ph);
81  }
82
83  Object* GDual:oppositeCheck
84   (Placeholder* ph, bool polarity) {
85    Placeholder* oph;
86
87    // loop to remove until empty or
88    // found a valid entry in opposite container
89    while (true) {
90      oph = subcon[!polarity]→remove();
91      if (oph == EMPTY) return EMPTY;
92
93      else { // attempt to abort oph
94        Object* oval = oph→val;
95        if (CAS(oph, ⟨oval, INVALID, NULL⟩,
96            ⟨oval, ABORTED, NULL⟩)) {
97          // abort succeeded;
98          continue;
99        } else { // abort failed
100         // placeholder guaranteed to be validated
101         if (polarity == DATA) {
102           *oph = ⟨ph→val, SATISFIED, NULL⟩;
103           return OK;
104         }
105         else return oph→val;
106       } // end abort failed
107     } // end attempt to abort
108   } // end empty check loop
109 }
110
111
112
```

Fig. 2. Pseudocode for the blocking generic dual construction.

A Placeholder tuple (a CAS-able word) contains both a value (datum or antidatum) and flags to indicate one of the four possible states: INVALID, ABORTED, VALID, and SATISFIED. Given the almost complete symmetry of positive and negative operations, we implement insert and remove as trivial wrappers around a single remsert method.

The polarity argument distinguishes positive and negative operations. The val argument provides data for positive operations; it is NULL for negative operations. The nb flag controls portions of the code that differ depending on whether we wish to close the preemption window and thus be fully nonblocking. Subcontainers are assumed to have been initialized before passing them to the GDual constructor.

On entering the main loop of remsert, we allocate memory for an INVALID (not yet validated) placeholder tuple, which we then store in the subcontainer whose polarity matches that of the current operation. On entering the oppositeCheck function, we attempt to remove (or peek at) an "opposite placeholder" (oph) from the other subcontainer. Assuming that such a placeholder exists, we guess that it has not yet been validated and we attempt to abort it. (As an optimization [not shown here], we could peek at its state before trying the CAS.) If the abort attempt fails in the nonblocking variant of the construction, oph could be in any of the other three states—VALID, SATISFIED, or ABORTED—since other threads may have access to it via peek. We first guess that it is VALID and attempt to satisfy it, remove it from its container, and return. If that attempt fails, oph is either SATISFIED or ABORTED; since it can no longer mix, we also attempt to remove it.

If our original attempt to abort oph fails in the blocking variant of the construction, oph's state can only be VALID since only the owning thread and we, the remover, have access to it. Consequently, we can "mix" with its contents (line 102) and return.

On discovering an empty opposite container, we break out of the empty check and attempt to validate our own placeholder, using CAS to resolve the race with any thread that removes or peeks at our placeholder. If the CAS succeeds, we have committed our operation and can either return (if we are a positive thread) or wait for our placeholder to be satisfied by a positive thread (if we are a negative thread). If the validation CAS fails, we have encountered a conflict with another thread and have been aborted. Since the opposite container may not be empty anymore, we loop back to the start of the outer loop.

In the nonblocking variant, we must do additional work to close the preemption window—work that manifests itself as a more complicated oppositeCheckNB function. As noted earlier, the preemption window occurs only when we are satisfying waiting threads—that is, when a positive thread operates on a negative structure.

At the beginning of its nonblocking opposite check, a positive thread preallocates a Request object, a small struct containing all information for mixing with antidata. On entering the main loop of the check, it first checks the activeReq field. A non–NULL value indicates an ongoing operation. If necessary, the thread assists the active request and clears the field.

If the thread discovers no active request, it calls peek to look at the antidata subcontainer. If peek returns EMPTY the check is complete – the subcontainer is indeed empty. However, if peek returns a placeholder, the positive thread will attempt to satisfy it. To do so, it writes the placeholder and key it received from peek into its Request object, then posts the now fully specified Request object to the outer container's activeReq field. Having posted the request, the thread helps itself in completing the request.

The helpRequestNB function takes a Request object and attempts to do the work it contains. After attempting to mix the desired data with antidata, the function takes down the active request and removes (if possible), the now irrelevant placeholder. It returns the final state of the placeholder, either ABORTED or SATISFIED. The former indicates that the request failed due to an INVALID placeholder;thus, the thread must try again. The latter indicates that the placeholder was indeed satisfied.

However, an ambiguity regarding the SATISFIED signal remains: it is possible that the placeholder was satisfied by a different thread earlier, but the placeholder was not removed due to a remove_conditional failure. To deal with this case, the satisfying thread stores a pointer to the satisfying Request object. Since Request objects are not

```
113 class Request {
114   Object* contents;
115   placeholder* ph;
116   Key key;
117 };
118
119 Object* GDual:oppositeCheckNB
120     (Placeholder* ph) {
121
122   // this method is called only
123   // by positive threads
124
125   Placeholder* oph;
126   Request* activeCopy;
127   Request* myReq = new Request();
128   myReq→contents = ph→contents;
129   Key k;
130   Object* rtn = EMPTY;
131
132   // loop to peek until empty
133   // or valid entry in opposite queue
134   while (true) {
135     // read active request, help if necessary
136     activeCopy = activeReq;
137     if (activeCopy ≠ NULL) {
138       helpRequestNB(activeCopy);
139       continue;
140     }
141     (k, oph) = subcon[ANTIDATA]→peek();
142     if (oph == EMPTY) {
143       // opposite is empty
144       // our check is complete
145       return EMPTY;
146     }
147     // set up my request
148     myReq→key = k;
149     myReq→ph = oph;
150
151     if (CAS(activeReq, NULL, myReq)) {
152       // posted my request as active
153       if (helpRequestNB(myReq) ≠ ABORTED) {
154         // my request was satisfied (by someone)
155         if (oph→req() == ph→req()) {
156           // verify my request ptr was used
157           return OK;
158         }
159       }
160       // my request was either aborted or oph
161       // was previously satisfied by another
162       // peeker, so reallocate and retry
163       myReq = new Request();
164       myReq→contents = ph→contents;
165     }
166   } // end peeking loop
167 }
168
169
170 int GDual:helpRequestNB(Request* req) {
171   int rtn;
172   Placeholder* oph = req→ph;
173   Object* oval = oph→contents;
174
175   // attempt to abort opposite operation
176   if (CAS(oph, ⟨NULL, INVALID, NULL⟩,
177   ⟨NULL, ABORTED, NULL⟩)) {
178     // abort succeeded
179     rtn = ABORTED;
180   }
181   else if (opp_ph→state == ABORTED) {
182     // someone else aborted the placeholder
183     rtn = ABORTED;
184   }
185   else {
186     // oph must be valid or satisfied
187     oph→satisfy(req→contents, req);
188     rtn = SATISFIED;
189   }
190   // take down posted request
191   CAS(activeReq, req, NULL);
192   // remove placeholder from opposite
193   subcon[ANTIDATA]→
194   remove_cond(req→key);
195   return rtn;
196 }
```

Fig. 3. Additional pseudocode for the nonblocking construction.

reclaimed until all references to them are lost, this pointer serves as a unique identifier for the active request. It allows the current thread to determine whether to return from oppositeCheckNB or to continue looking for placeholders.

Storage Management. As is usual in concurrent structures, we must coordinate across threads to determine when it is safe to free data. For the blocking variant, placeholders are managed by using a counterfield to determine when both the inserter and remover are done with the object. When a thread discovers that it is the second to complete, it frees the object. In the nonblocking variant of the construction, an arbitrary number of threads can gain access to a placeholder via peek. We therefore resort to hazard pointers [Michael 2004] (not shown in the pseudocode) for storage reclamation. For efficiency, we maintain thread-local pools of available placeholders.

3.2. Correctness

In this section, we provide safety and liveness proofs for nonblocking dual containers built using our generic construction. We assume that the underlying containers

are known to be linearizable and nonblocking and that they support the operations described in Section 3.1.1 with subcontainer-specific ordering disciplines $<^+$ and $<^-$. To simplify the presentation, we assume not only that all data values are unique (clearly they could be made so by including a thread id and serial number) but also that the values include any information (e.g., thread priority) needed to drive the $<^+$ and $<^-$ relations.

As described in Section 2, we require an interface whose remove operation is paired with explicit remove_request and remove_followup operations as opposed to a combined remove. We consider only *well-formed* concurrent histories, in which the calls to a given dual container in any given thread subhistory are a prefix of some string in $(i, r\, u^* s)^*$, where i is an insert operation, r is a remove_request, u is an unsuccessful remove_followup on the ticket returned by the previous r, and s is a successful remove_followup on that ticket. To cast our construction in this mold, we make four trivial modifications to the pseudocode of Figure 2: (1) rename remove to be remove_request; (2) modify the code at line 105 to return a ticket containing oph; (3) modify the code at line 65 to return a ticket containing ph; and (4) create a tiny remove_followup method that inspects the ticket and returns the val field of the placeholder therein.

3.2.1. Safety. To prove the safety of our construction, we need to identify desired sequential semantics, choose linearization points for our operations, and demonstrate that any realizable parallel execution has the same observable behavior as a sequential execution performed in linearization order.

The Two-Order Container. Since dualism is a parallel concept that maps inperfectly to sequential programming, we must invent a somewhat artificial object with which to demonstrate equivalence. We call this object a *two-order container* (TOC). Like our generic dual container, the TOC comprises positive and negative subcontainers, with respective ordering relations $<^+$ and $<^-$. In this sequential case, however, both subcontainers provide only the standard insert and remove methods.

The TOC exports insert, remove_request, and remove_followup methods. The TOC is said to be *empty* if its history to date includes an equal number of insert and remove_request operations. It is said to be *positive* or *negative* if its history includes an excess of insert or remove_request operations, respectively. The remove_request method creates an antidatum, which contains both a slot into which a data value can be written and whatever other information is required to drive the $<^-$ relation. If the TOC is positive, remove_request removes the smallest datum, according to $<^+$, from the positive subcontainer and writes it into the antidatum. If the TOC is negative or empty, remove_request inserts the antidatum into the negative subcontainer. In either case, it returns a *ticket* containing a reference to the antidatum. The insert method takes a datum as argument. If the TOC is positive or empty, insert adds its datum (and any other information needed to drive $<^+$) to the positive subcontainer; otherwise, it removes the smallest antidatum, according to $<^-$, from the negative subcontainer, and writes its datum into it. The remove_followup method takes a ticket (antidata reference) as argument; it returns the data value written in the antidatum, or NULL if there is none.

A *successful* remove_followup—one that returns non-NULL, is said to *match* the insert that provided its value. So, too, are the remove_request that returned the remove_followup's ticket, and any intervening unsuccessful remove_followups that were also passed that ticket. Similarly, the insert is said to match the successful remove_followup, its remove_request, and any unsuccessful remove_followups. We consider only *well-formed* sequential histories—those in which every ticket passed to remove_followup was returned by a previous remove_request, and no ticket is passed to remove_followup twice if the earlier call was successful.

Linearization Points. To prove the safety of our generic dual container (GDC), it suffices to show that in any realizable concurrent history it is possible to identify linearization points (each between the call and return of its operation) such that the history has the same observable behavior (i.e., return values) as a sequential execution, in linearization order, of the same operations on a TOC. To minimize confusion, we consider only the case in which the nonblocking flag is true; therefore, nb is true (line 40) if and only if the current outer-level operation is an insertand thus may need to satisfy a waiting thread.

Our linearization points are dynamic and chosen in retrospect. For simplicity, we consider only GDC histories in which all operations have completed. (The extensions needed for uncompleted operations are tedious but straightforward.) We assume that the history includes all instructions performed within operations of the underlying nonblocking containers and that the linearization points of these operations have already been identified. Working through the history in time order, we apply the following rules:

(1) If a call to satisfy succeeds at line 187, we linearize its operation at the linearization point of the active request's peek that was called at line 141.
(2) If an operation returns a validated data placeholder from line 105, we linearize the operation at the linearization point of the already-completed remove that was called at line 90.
(3) If a peek at line 141 or a remove at line 90 returns EMPTY, we consider the current operation and all other operations of the same polarity that have inserted a placeholder into their subcontainer but have not yet linearized. Among these, we select all those that successfully validate their placeholder somewhere later in the history. (Any that have already validated their placeholders will already have been linearized, by this same rule.) We then linearize all the selected operations at the subcontainer linearization point of the current peek or removein the order in which their respective line-49 inserts linearized.

A bit of study confirms that these three cases cover all possible paths through the code. The trivial remove_followup method, not shown in Figure 2, linearizes on its load of the val field of the placeholder referred to by its ticket.

The intuition behind the third, admittedly complicated, rule is that in-flight operations that will ultimately succeed at inserting a validated placeholder into a subcontainer should linearize *in insertion order* when the opposite-polarity subcontainer is known to be empty.

THEOREM 1 (LINEARIZABILITY). *Any realizable well-formed history of the GDC containing only completed operations is equivalent to a legal well-formed history of the TOC.*

PROOF. Inspection of the code in Figure 2 confirms that, as in the TOC, every insert or remove_request operation either inserts a subsequently validated placeholder into the like-polarity subcontainer or mixes with and removes (or verifies the removal of) a validated placeholder from the opposite polarity subcontainer (and never both). Let us refer to operations that insert subsequently validated placeholders as *leading* operations, and to operations that mix with an existing validated placeholder as *trailing* operations.

Whenever a subcontainer is empty, the history *of that subcontainer* must (by assumption of correctness of the subcontainers) include an equal number of like-polarity (leading) and opposite-polarity (trailing) operations. Whenever a subcontainer is nonempty, the history of that subcontainer must include an excess of like-polarity (leading) operations.

Our linearization procedure arranges, by construction, for every leading operation to linearize at a point at which the subcontainer of opposite polarity is empty, and for every trailing operation to linearize at a point at which the subcontainer of opposite polarity is nonempty. It is easy to show by induction that the GDC linearizes a leading operation if and only if the number of previously linearized like-polarity operations equals or exceeds the number of previously linearized opposite-polarity operations; it linearizes a trailing operation if and only if the number of previously linearized opposite-polarity operations exceeds the number of previously linearized like-polarity operations. Moreover—again by construction of the linearization procedure—operations that insert and remove (eventually) validated placeholders in subcontainers linearize in the order of the subcontainer operations, and the GDC duplicates the semantics of the TOC. □

3.2.2. Liveness and Contention Freedom. Theorem 2 asserts that the methods of the GDC are obstruction-free. Theorems 3 and 4 assert additional properties required of nonblocking dual data structures [Scherer and Scott 2004].

THEOREM 2 (OBSTRUCTION FREEDOM). *If variable* nonblocking *is true (line 40) and arguments* dc *and* ac *refer to correct nonblocking containers (line 23), then the generic dual container is indeed obstruction free.*

PROOF. Aside from the spin at line 65, which we eliminated in favor of repeated calls to a remove_followup method, the code of Figure 2 contains only three loops. The oppositeCheck and oppositeCheckNB loops (lines 89 and 134, respectively) remove elements repeatedly from a finite container and terminate when it is empty. The latter can also repeat due to contention with another thread on the activeRequest field; this manifests as failure of the CAS on line 151, indicating that the other thread is making progress. The remsert loop (line 47) repeats only when the CAS at line 60 fails due to contention with another thread. In the absence of such contention, all operations complete in bounded time. □

THEOREM 3 (IMMEDIATE WAKEUP). *If a thread A performs an unsuccessful remove_followup operation, u^A, and some other thread B performs a successful remove_followup operation, s^B, between A's remove_request, r^A, and u^A, then $r^B <^- r^A$ or i^B linearizes before r^A, where i^B is the insert operation that matches r^B.*

In other words, if r^A and r^B are in the antidata subcontainer at the same time and if $r^A <^- r^B$, then it is not possible for A to experience an unsuccessful remove_followup after B has experienced a successful remove_followup. Even more informally, a waiting thread is guaranteed to wake up immediately after the matching insert.

PROOF. By contradiction: Suppose that we have that $i^B \prec s^B \prec u^A$ (the premise), where $\prec$ indicates linearization order, but also $r^A \prec i^B$ and $r^A <^- r^B$ (the negation of the conclusion). The existence of u^A implies that the GDC is negative after r^A's linearization point and remains so at least through u^A. Thus, if there exists an insert i^A that matches r^A, we must have that $r^A \prec i^A$. Moreover, i^B must also be a trailing insert, meaning that $r^B \prec i^B$. Thus, r^A and r^B are both present as placeholders in the antidata subcontainer when i^B linearizes.

Clearly, if i^A exists, it must come before or after i^B. If $i^B \prec i^A$ or if i^A does not exist, then r^A and r^B are both present in the antidata subcontainer when i^B peeks at it and sees r^B, contradicting the assumption that $r^A <^- r^B$. If $i^A \prec i^B$, then i^A must perform its remove_conditional on the antidata subcontainer before i^B can see r^B, and it will satisfy r^B's placeholder in-between. This, in turn, implies that u^A cannot follow s^B, another contradiction. □

THEOREM 4 (CONTENTION FREEDOM). *Unsuccessful remove_followup() operations perform no remote memory accesses.*

PROOF. In the absence of false sharing, the cache line containing the caller's placeholder will remain in the local cache until it is written by the satisfying update. Waiting threads therefore cause no memory contention. □

4. FAST DUAL RING QUEUES

While the generic construction of the previous section allows any positive subcontainer to be paired with almost any negative subcontainer, intuition suggests (and the results in Section 5 confirm) that the pairing imposes nontrivial overheads. To maximize performance while preserving dual semantics and nonblocking progress, we have developed custom, combined structures based on modern FIFO queues.

The dual structures of Scherer and Scott [2004], found in the java.util.concurrent library, were based on the well-known M&S queue [Michael and Scott 1996] and (for the "unfair" non-FIFO version) the Treiber stack [Treiber 1986]. Since their development a decade ago, significantly faster concurrent queues have been devised. Notable among these is the linked concurrent ring queue (LCRQ) of Morrison and Afek [2013]. 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 within 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 FAI instruction, the hardware can (and indeed, does, on an x86 machine) arrange for all threads to succeed in linear time [Freudenthal and Gottlieb 1991]. 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 operations 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 section, 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 dual queues 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× higher than that of the M&S-based structure.

We review the operation of Morrison and Afek's LCRQ in Section 4.1. We introduce performance-oriented (but potentially blocking) versions of our new queues in Sections 4.2 (the single-polarity dual ring queue—SPDQ) and 4.3 (the multipolarity dual ring queue—MPDQ). Lock-free variants appear in Section 4.4. Proofs of linearizability appear in Section 4.5. As in Section 3, we assume the availability of (hardware-supported) CAS, though load-linked/store-conditional would also suffice. To resolve the ABA problem [Scott 2013, Sec. 2.3.1], 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 FAI instruction that always succeeds. Our pseudocode, as written, assumes sequential consistency for simplicity, and we omit the code required for storage management. Our C++ implementation uses `atomic` declarations for variables with races (forcing the compiler to insert store-load

fences when necessary for correctness) and delays reclamation using hazard pointers [Michael 2004]. Performance results (combined with those of the generic construction) appear in Section 5.2.

4.1. LCRQ Overview

We here provide a brief overview of the CRQ and LCRQ algorithms from which our SPDQ and MPDQ are derived. For a more complete treatment of the original algorithms, readers may wish to consult the paper by Morrison and Afek [2013]. Complete pseudocode for the LCRQ algorithm, adapted from that earlier paper, can be found in Appendix B.

The LCRQ is a major enhancement of the M&S linked-list queue [Michael and Scott 1996]. 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.

4.1.1. CRQ Algorithm. Each CRQ comprises a circular array of R elements and 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), the index (idx) of this data, and a safe bit that roughly indicates that the dequeuer of the data is known not to have arrived at the slot before the enqueuer. 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 mod R. The closed bit on the tail counter is used to throw a tantrum and inhibit further enqueues.

Ideal Operation. Ideally, an enqueue operation simply:

(1) performs an 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); and
(3) uses CASto insert the new data value into the chosen slot.

Conversely, the ideal dequeue operation:

(1) performs an FAI on the head counter to retrieve an index;
(2) performs a mod operation on the index to identify a slot; and
(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 the corresponding enqueue.

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 that we have wrapped all the way around the buffer.
>
> *Case 2e*. Evidence suggests (presented later) that what would have been the corresponding dequeue operation may already have run, implying that any data that we enqueue would never be dequeued.

In either of these cases, the enqueuer skips the index and counts on the dequeuer (if it has not run yet) to recover.

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 later).

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 head $-$ 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 head $\geq$tail. Prior to returning and letting the caller retry, 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.

4.1.2. 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; 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.

4.2. Single-Polarity Dual Ring Queue

In the original dual queue of Scherer and Scott [2004] (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.

In our 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 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.

4.2.1. 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*).

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

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" method (Figure 5), which handles both (similar to the remsert method of the generic construction).

Unless a denqueue invocation discovers otherwise, it assumes that 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. Note that line 241 of Figure 5 references the nonblocking variant of the SPDQ, for which we delay explanation until Section 4.4.

4.2.2. Modifications to the CRQ. Our SP_CRQ variant of the CRQ class incorporates three small changes. First, we add a Boolean field polarity at line 94 to indicate the type of ring. Second, when data "mixes" with existing antidata, we arrange to alert the waiting negative thread so that it can return. Specifically, we represent the antidata elements of a negative ring as pointers to Waiter objects (Figure 4). We then replace the element-removing CAS at line 158 with a call to ((Waiter*)old)→satisfy(arg), where arg is the datum being provided by the positive thread. If this CAS succeeds, we perform a "blind clean-up" CAS on the actual ring slot. If that second CAS fails, someone else has cleaned up for us; either way, we return. (This same idiom can be seen at line 413 in the lock-free variant of the SP_CRQ, which we will discuss in Section 4.4.)

Our third change adds another Boolean flag, sealed, to the CRQ at line 94 and a new method, seal (Figure 6), that attempts to close the queue atomically if it is currently empty. Once seal succeeds, the SP_CRQ is both closed, preventing additional enqueues, and empty, allowing it to be removed from the linked list. Our implementation of seal is based on the fixState method of the original CRQ.

```
198  class SPDQ {
199    SP_CRQ* head, tail;
200    bool nonblocking;
201  };
202
203  class SP_CRQ:CRQ {
204    bool sealed;
205    bool polarity;
206    bool seal();
207  };
```

```
208  class Waiter() {
209    Object* val;
210    Object* spin() {
211      while (val == NULL) {}
212      return val;
213    }
214    bool satisfy(Object* arg) {
215      return CAS(&val, NULL, arg);
216    }
217  };
```

Fig. 4. SPDQ data types.

```
218  Object* SPDQ:dequeue() {
219    Waiter* w = new Waiter();
220    // Waiter contains slot in which
221    // to place satisfying data
222    return denqueue(w, ANTIDATA);
223  }
224
225  void SPDQ:enqueue(Object* val) {
226    return denqueue(val, DATA);
227  }
228
229  Object* SPDQ:denqueue
230   (Object* val, bool polarity) {
231    SP_CRQ* h;
232    bool nb = (polarity == DATA && nonblocking);
233    while (true) {
234      h = head; // read polarity of queue
235      if (h→polarity == polarity) {
236        v = internal_enqueue(h, val, polarity);
237        if (v ≠ TWISTED) return OK;
238      }
239      else {
240        if (nb)
241          v = internal_dequeue_NB(val);
242        else
243          v = internal_dequeue(val, polarity);
244        if (v ≠ TOO_SLOW) return v;
245      }
246      // if internal operation failed,
247      // head has changed, so retry
248    }
249  }
```

Fig. 5. SPDQ denqueue.

```
250  bool SP_CRQ:seal() {
251    int h, t;
252
253    h = head;
254    ⟨closed,t⟩ = tail;
255    if(closed == 1 && h ≥ t){
256        sealed = true;
257        return true;
258      }
259    }
260
261    while (true) {
262      if (sealed) return true;
263
264      h = head;
265      ⟨closed,t⟩ = tail;
266      // check if not empty
267      if (h < t && sealed==false) {
268        return false; // if not, seal failed
269      }
270      // try to close while empty
271      // (if an enqueue occurs,
272      // tail moves, and CAS fails)
273      if (CAS(&tail, t, ⟨1, h⟩)) {
274        // CAS succeeded, so CRQ is
275        // closed and empty
276        sealed = true;
277        return true;
278      }
279    }
280
281  }
```

Fig. 6. SP_CRQ seal.

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

In the internal enqueue method (Figure 7), we first read tail and verify both that it is indeed the tail of the list [Michael and Scott 1996] 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.

4.2.4. Internal Dequeue. In the internal dequeue method (Figure 8), we again verify that 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

```
Object* SPDQ:internal_enqueue
 (SP_CRQ* h, Object* val, bool polarity) {
  SP_CRQ* t, next, newring;

  while (true) {
    t = tail;
    // verify tail is the actual tail
    if (t→next ≠ NULL) {
      next = t→next;
      (void)CAS(&tail, t, next);
      continue;
    }

    // verify correct polarity (detect twisting)
    if (t→polarity ≠ polarity) {
      (void)CAS(&head, h, h→next);
      return TWISTED;
    }

    // attempt enqueue on tail
    if (t→enqueue(val) == OK) {
      if (polarity == ANTIDATA)
        return ((Waiter*)val)→spin();
      else return OK;
    }

    // else the tail is closed
    newring = new SP_CRQ(polarity);
    newring→enqueue(val);

    // append ring
    if (CAS(&t→next, NULL, newring)) {
      (void)CAS(&tail, t, newring);
      if (polarity == ANTIDATA)
        return ((Waiter*)val)→spin();
      else return OK;
    }
  }
}
```

Fig. 7. SPDQ internal_enqueue.

```
Object* SPDQ:internal_dequeue
 (Object* val, bool polarity) {
  SP_CRQ* h, next, newring;

  while (true) {
    h = head;
    // verify queue polarity didn't change
    if (h→polarity == polarity) {
      // head polarity inverted,
      // so recheck queue state
      return TOO_SLOW;
    }
    // dequeue from head
    v = h→dequeue(val);
    if (v ≠ EMPTY) return v;

    // seal empty SP_CRQ so we can remove it
    else if (!h→seal()) continue;

    // at this point head SP_CRQ is sealed
    if (h→next ≠ NULL) {
      // swing the head
      (void)CAS(&head, h, h→next);
    } else {
      // add a new tail and swing head to it
      newring = new SP_CRQ*(polarity);
      newring→enqueue(val);

      // append our new ring to list,
      // which will cause twisting
      if (CAS(&h→next, NULL, newring)) {
        (void)CAS(&tail, h, newring);
        // swing head to fix twisting
        (void)CAS(&head, h, h→next);
        if (polarity == ANTIDATA)
          return ((Waiter*)val)→spin();
        else return v;
      }
    }
  }
}
```

Fig. 8. SPDQ internal_dequeue.

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 (Figure 9). 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.

4.2.5. Forward Progress. Public enqueue (positive) operations inherit lock-free progress from the LCRQ algorithm [Morrison and Afek 2013]. 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 its datum, closes the current ring, and appends the new ring to the list. As in the M&S queue [Michael and Scott 1996], the append can fail only

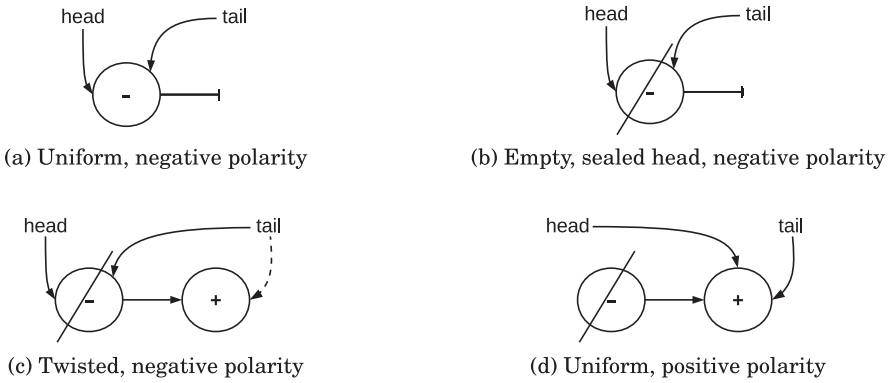

Fig. 9. Flipping the polarity of the SPDQ.

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. Recall from Section 2 that these are modeled as a potentially unbounded sequence of nonblocking operations. The initial remove_request linearizes the request for data of the calling thread, T. The last operation (the successful remove_followup) 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 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—manifesting the same preemption window as the generic dual construction. 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 (Section 5.2). In Section 4.4, 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.

4.3. Multi-Polarity Dual Ring Queue

In contrast to the SPDQ, the MPDQ incorporates the flipping functionality at the ring buffer level and leaves the linked list structure of the LCRQ mostly unchanged. The MPDQ takes advantage of the preallocated nature of ring slots, discussed at the beginning of Section 4.2. For a dual structure, it does not matter whether data or antidata is first placed in a slot; either can "mix" with the other.

4.3.1. Overview. In their original presentation of the CRQ, Morrison and Afek [2013] 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

```
364 tuple MP_Slot {
365   bool safe; // 1 bit
366   bool polarity; // 1 bit
367   int idx; // 30 bits
368   Object* val; // 32 bits (int or pointer)
369   // padded to cache line size
370 };
371
372 class MP_CRQ { // fields on distinct cache lines
373   ⟨bool closing, int idx⟩ data_idx; // 1, 31 bits
374   ⟨bool closing, int idx⟩ antidata_idx; // 1, 31 bits
375   ⟨bool closed, int idx⟩ closed_info; // 1, 31 bits
376   MP_CRQ* next;
377   MP_Slot ring[R]; // initially ring[i] = ⟨1, 1, i, NULL⟩ ∀i
378 };
```

```
379 class MPDQ { // fields on distinct cache lines
380   MP_CRQ* data_ptr;
381   MP_CRQ* antidata_ptr;
382   bool nonblocking;
383
384   Object* dequeue() {
385     Waiter* w = new Waiter();
386     return denqueue(w, ANTIDATA);
387   }
388
389   void enqueue(Object* val) {
390     (void)denqueue(val, DATA);
391   }
392 };
```

Fig. 10. MPDQ data types and simple methods.

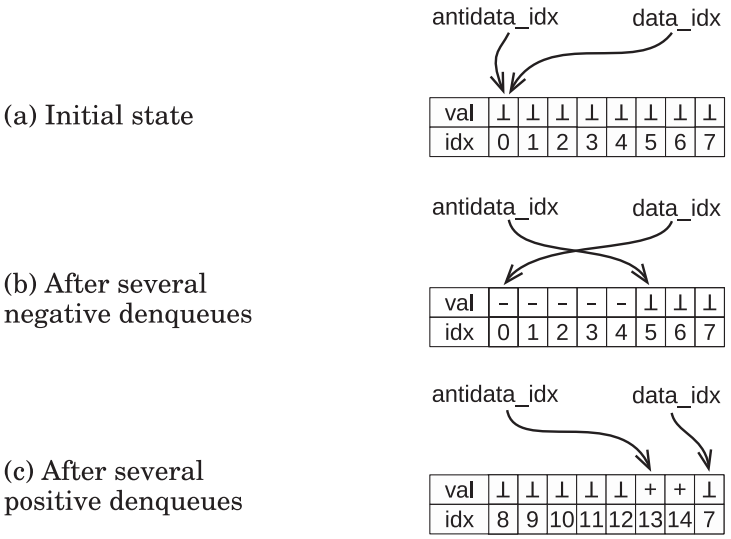

Fig. 11. Flipping the polarity of the MP_CRQ.

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 and
(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 (Figure 10). We also add a bit to each slot to identify its polarity.

Figure 11 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 4.3.3.

```
393  Object* MP_CRQ:denqueue
394    (Object* arg, bool polarity) {
395      ⟨bool closing, int idx⟩ p; // 1, 31 bits
396      ⟨bool closing, int idx⟩* my_cntr;
397      ⟨bool closing, int idx⟩* their_cntr;
398      MP_Slot* slot;
399      bool safe;
400      int idx;
401      Object* val;
402      bool slot_polarity;
403
404      int starvation_counter = 0;
405      int closeIdx = 0;
406
407      // determine which index to use
408      if (polarity == DATA) {
409        my_cntr = &data_idx;
410        their_cntr = &antidata_idx;
411      } else {
412        my_cntr = &antidata_idx;
413        their_cntr = &data_idx;
414      }
415
416      // do denqueue
417      while (true) {
418        ⟨p.closing, p.index⟩ = FAI(my_cntr);
419        // check for closing
420        if (p.closing == true) {
421          closeIdx =
422            discovered_closing(p.idx, polarity);
423          if (closeIdx ≤ p.idx) return CLOSED;
424        }
425        slot = &ring[p.idx % R];
426
427        while (true) {
428          ⟨safe, slot_polarity, idx, val⟩ = *slot;
429
430          // if slot nonempty
431          if (val ≠ NULL) {
432            // try to dequeue opposite
433            if (idx == p.idx
434                && slot_polarity ≠ polarity) {
435              if (CAS(&slot,
436                  ⟨safe, p.idx, val, slot_polarity⟩,
437                  ⟨safe, p.idx+R, NULL, slot_polarity⟩)) {
438                if (polarity == ANTIDATA) return val;
439                else {
440                  ((Waiter*)val)→satisfy(arg);
441                  return NULL;
442                }
443              } else continue;
444            }
445            // failed to dequeue; signal slot unsafe
446            // to prevent corresponding operation
447            else {
448              if (CAS(&slot,
449                  ⟨safe, idx, val, slot_polarity⟩,
450                  ⟨0, idx, val, slot_polarity⟩)) {
451                break;
452              } else continue;
453            }
454          }
455          // if slot empty, try to enqueue self
456          else {
457            if (safe == 1 || their_cntr→idx ≤ p.idx) {
458              if (CAS(&slot,
459                  ⟨safe, idx, NULL, slot_polarity⟩,
460                  ⟨1, idx, arg, polarity⟩)) {
461                return OK;
462              } else continue;
463            }
464            else break; // unsafe, try the next index
465          }
466        } // end inner while loop
467
468        starvation_counter++;
469
470        // if fail to make progress, close the ring
471        if ((p.idx−their_cntr→idx ≥ R)
472            || starvation_counter>STARVATION)) {
473          my_ptr→close();
474          closeIdx =
475            discovered_closing(p.idx, polarity);
476          if (closeIdx ≤ p.idx) return CLOSED;
477        }
478      } // end outer while loop
479  }
```

Fig. 12. MPDQ ring denqueue.

4.3.2. MP_CRQ *denqueue().* As the MPDQ is more symmetric than the SPDQ, we can combine all ring operations into a single denqueue method (Figure 12). 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.

Note that at no point will the MP_CRQ ever return EMPTY, as any thread that believes that 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.

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; and

(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.

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, wraparound 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; thus, 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 occurs when $|\,\text{data\_idx} - \text{antidata\_idx}\,| \geq R$. 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 4.2.5), 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 via tantrum semantics and moves on.

Counter overflow. On a 32b machine, we can allocate only 30b 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.

4.3.3. 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 (Figure 10). 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 (Figure 13), 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

```
480  int MP_CRQ:discovered_closing(bool polarity) {
481    ⟨bool closing, int idx⟩ d_idx;
482    ⟨bool closing, int idx⟩ a_idx;
483
484    // check if already closed
485    if (closed_info.closed == 1) return closed_info.idx;
486
487    // set closing
488    antidata_idx.atomic_set_closing(1);
489    data_idx.atomic_set_closing(1);
490
491    // next read both indices and try to close queue
492    d_idx = data_idx;
493    a_idx = antidata_idx;
494    int closed_idx = max(d_idx.idx, a_idx.idx);
495    (void)CAS(&closed_info, ⟨0, 0⟩, ⟨1, closed_idx⟩);
496    return closed_info.idx;
497  }
```

Fig. 13. MPDQ closing.

```
498  Object* MPDQ:denqueue(Object* arg,
499   bool polarity) {
500    MP_CRQ* m, next, newring;
501    int v;
502    MP_CRQ** my_ptr;
503    bool nb = (polarity == DATA && nonblocking);
504
505    if (polarity == DATA) my_ptr = &data_ptr;
506    else my_ptr = &antidata_ptr;
507
508    while (true) {
509      m = *my_ptr;
510      // denqueue
511      if (nb) v = m→denqueue_NB(arg);
512      else v = m→denqueue(arg, polarity);
513
514      // successful denqueue
515      if (v ≠ CLOSED) {
516        if (polarity == ANTIDATA && v == OK)
517          return ((Waiter*)val)→spin();
518        else if (polarity == DATA) return OK;
519        else return v;
520      }
521      // my_ptr is closed, move to next
522      if (m→next ≠ NULL)
523        (void)CAS(my_ptr, m, next);
524      else {
525        // if no next, add it
526        newring = new MP_CRQ();
527        v = newring→denqueue(arg);
528        if (CAS(&m→next, NULL, newring)) {
529          (void)CAS(my_ptr, m, newring);
530          if (polarity == ANTIDATA)
531            return ((Waiter*)val)→spin();
532          else return OK;
533        }
534      }
535    }
536  }
```

Fig. 14. MPDQ list denqueue.

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.

4.3.4. MPDQ List. At the backbone list 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 that it must move along to the next one. If no next ring exists, it creates one and appends it to the list in the style of the M&S queue. It first updates the next pointer of the previous final ring and then swings the main data_ptr and/or antidata_ptr, as appropriate. Once again, we are able to combine positive and negative operations into a denqueue method at the list level (Figure 14).

4.3.5. Forward Progress. Like the SPDQ and the generic dual construction, 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 in which no other thread can help it. The following section addresses this concern.

4.4. 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 nonblocked 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 Section 5.2) have almost no impact on performance. The changes are larger in the MPDQ case, with a larger performance impact. Pseudocode for both changes can be found in Appendix B.

4.4.1. SPDQ Wavefront. As can be observed in the original CRQ algorithm, only internal dequeue operations can change the value of a given slot's index. If all internal 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*, in which the difference between neighboring indices is greater than one.

The discontinuity in indices (the MPDQ analogue of which can be seen in Figure 11), indicates that a slot is ready and can be used to strictly order dequeue operations for the SPDQ. Since the preemption window occurs only when a positive thread dequeues from a negative ring, we can limit code changes to this single case.

Before a thread can complete an operation on an index (including returning a CLOSED signal from MP_CRQ:denqueue), 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 the number of active threads at any point is equal to the distance from the wavefront to the head, all threads will eventually succeed.

4.4.2. MPDQ Wavefront. In contrast to the SPDQ, the preemption window can also manifest itself in the MPDQ during an enqueue 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). The algorithm thus is similar to the SPDQ, involving a backward search after timeout, but cannot be isolated as easily—all enqueue operations must follow the wavefront protocol.

4.5. Correctness

This section provides proofs of correctness for the blocking versions of both the SPDQ and the MPDQ.

Due to the preemption window problem, neither algorithm adheres precisely to the definition of a nonblocking dual data structure. In particular, the preemption window appears when a positive thread has found its corresponding antidata but has not yet passed the data to the waiting negative thread. If we allow a negative thread to execute arbitrary code between calls to remove_followup, it may learn of a successful "subsequent" remove in some other thread and then go on to perform an unsuccessful remove_followup of its own. To avoid this case, we assume for purposes of our proof that remove_followup is blocking and always succeeds. In terms of sequential semantics, this means that remove_followup is partial: there are no legal sequential histories in which a remove_followup appears before its matching insert.

Like in Section 3.2, we assume that all data values are unique and we consider only well-formed concurrent histories, in which every thread subhistory for a given dual container is a prefix of some string in $(i, r\, s)^*$, where i is an insert operation, r is a remove_request, and s is a (successful, and conceivably blocking) remove_followup on the ticket returned by remove_request.

For purposes of the proof, we make trivial modifications to the pseudocode of both the SPDQ and the MPDQ: (1) for both, we rename remove to remove_request; (2) for both, instead of entering the Waiter::spin method, in which negative threads would wait for data (line 13), we return a ticket containing the Waiter object; (3) we modify the code at lines 337 and 438, where negative threads currently discover existing data, to return a ticket Waiter object whose val field contains this data; and (4) we rename the spin (Line 13) to remove_followup, exporting it via a public interface that takes a Waiter object as a parameter.

To prove safety, we first review the sequential semantics of a FIFO queue and its dual variant. For each of the SPDQ and the MPDQ, we then identify linearization points for each method and demonstrate that all realizable concurrent histories are equivalent to some valid sequential history.

FIFO Queue. A queue is a standard abstract container with insert and remove methods. When the queue is total, the remove method returns the oldest value previously inserted by an insert operation that has yet to be returned by remove. If no such value exists, remove returns an EMPTY signal.

A sequential dual queue exports three methods: insert, remove_request, and remove_followup. The insert method inserts a data value into the structure, while a remove_request inserts a request and returns a *ticket*. A remove_followup takes a ticket returned by a previous remove_request and returns a data value. The ith insert and ith remove_request are *matching*: a remove_followup called on the ticket returned by the ith remove_request always linearizes after, and returns the value inserted by, the ith insert. Separation of the (nonblocking) remove_request operation from the (blocking) remove_followup allows us to observe that remove_followup performs no remote memory operations prior to its linearization point. Reusing terminology from the proof of the generic dual container, in a matching insert/remove_request pair, we call the first to linearize the *leading* operation and the second the *trailing* operation.

4.5.1. SPDQ Safety. Our SPDQ data structure relies on the correctness of the underlying SP_CRQ "tantrum queue" implementation borrowed from Morrison and Afek [2013]. External insert and remove_request operations on the SPDQ have the following linearization points:

(1) When the tail is CLOSED, a leading operation (one whose match does not yet exist) linearizes upon appending a new tail SP_CRQ to the list, either at line 313 or at line 353.
(2) When the tail is not CLOSED, a leading operation linearizes upon enqueueing into the tail SP_CRQ at line 302.
(3) A trailing operation linearizes upon dequeuing from the underlying head SP_CRQ at line 336.

The trivial remove_followup method linearizes within the Waiter object when it discovers data (line 14).

THEOREM 5 (LINEARIZABILITY). *Any realizable well-formed history of the SPDQ containing only completed operations is equivalent to a legal well-formed history of the sequential dual queue.*

PROOF. We begin by noting a queue state invariant; the queue is always *uniform*, *twisted*, or *empty*. Inspection of the code reveals that this invariant is maintained by ensuring that every time an SP_CRQ is appended to the list, its predecessor is either of a like polarity or is sealed, and both properties are monotonic. By maintaining this invariant, we ensure that the queue also has a uniform polarity across all its contents and that it passes through a distinct empty state in which it contains neither data nor antidata. We now reason within a period between these empty states in which the queue stores a single polarity.

Leading operations of this polarity linearize in order within a single SP_CRQ and a given SP_CRQ is CLOSED before appending a successor. Thus, leading operations are ordered internally by FIFO order. Trailing operations linearize in order within a single SP_CRQ and a given SP_CRQ is sealed before dequeuing from its successor. Thus, trailing operations follow the same FIFO order as leading operations.

An operation that discovers that the tail is empty but of an opposite polarity will seal the tail, putting the queue into an empty state. This operation's match cannot have linearized by this point because, if it did, the tail could not be empty. Thus, the queue is in an empty state when the numbers of insert and remove_request operations are equal.

Since leading operations are stored in FIFO order according to the linearization points, matching operations are serviced in FIFO order according to the linearization points, and the transition between polarities must pass through an empty state first, the internal state of the SPDQ precisely matches a valid state of a sequential dual queue and has an equivalent sequential history. □

4.5.2. MPDQ Safety. The MPDQ's linearization points are similarly straightforward:

(1) If an operation is a leading operation and it successfully enqueues into an MP_CRQ at line 418, it linearizes at the most recent fetch-and-increment at line 418.
(2) If an operation is a leading operation and no room exists in the newest MP_CRQ, it linearizes upon appending a new tail MP_CRQ to the list at line 528.
(3) If an operation is a trailing operation and it successfully dequeues from an MP_CRQ at line 435, it linearizes at the most recent FAI at line 418.

As in the SPDQ, the trivial remove_followup method linearizes within the Waiter object when it discovers data (line 14).

THEOREM 6 (LINEARIZABILITY). *Any realizable well-formed history of the MPDQ containing only completed operations is equivalent to a legal well-formed history of the sequential dual queue.*

PROOF. We begin by noting that the MP_CRQ provides correct sequential dual queue semantics between its creation and its tantrum. By design, any enqueuer or dequeuer that fails at a given index ensures that no other operation can use that index. All successful operations linearize at their earlier FAI; consequently, their index corresponds with their order among like-polarity operations. Thus, all successful insert and remove_followup operations are FIFO ordered, respectively, by index, meaning that the ith insert shares a slot with the ith remove_followup.

The discovered_closing routine ensures that any MP_CRQ, upon throwing a tantrum, stores the same number of data and antidata entries and that an MP_CRQ can never overflow.

The linking protocol ensures that the previous MP_CRQ is closed at some index before appending a new MP_CRQ, and the enqueue protocol for a given polarity operation ensures that a prior MP_CRQ is full for said polarity before moving on to its successor.

Consequently, the internal state of the MPDQ always represents a valid list of data and antidata sorted in FIFO order. Across the history, matching operations share a slot such that the ith insert synchronizes with the ith remove_followup. □

4.5.3. Liveness. As noted in Section 4.4, the present ring buffer algorithms are vulnerable to a preemption window, possibly leading to blocking in remove_followup. However, insert and remove_request are lock free for both the SPDQ and the MPDQ. We here provide brief proofs of their liveness properties. Fully lock-free versions of the SPDQ and MPDQ appear in Appendix B.

SPDQ Liveness. The underlying SP_CRQ inherits its lock-free progress from the original CRQ. Outside of the SP_CRQ, three loops exist in the code, in denqueue, internal_enqueue, and internal_dequeue. If the denqueue loop retries, the polarity of the underlying queue did not match the head SP_CRQ, either because the polarity changed or the queue was twisted. If the polarity changed, this means that some thread made progress by appending a node. If the queue was twisted, subsequent calls to internal_enqueue or internal_dequeue will detect and remove the sealed head node. The loop in internal_enqueue only retries when failing to append a node or when helping the tail pointer reach the end of the list. In both cases, another thread has made progress. The loop in internal_dequeue retries only if it removes a sealed node from the head of the list, helping another thread to make progress. Thus, investigation of loops proves that the insert and remove_request methods of the SPDQ are lock-free.

MPDQ Liveness. The underlying MP_CRQ has two nested loops. The inner loop retries if one of several CASes on the underlying slot fails, either because the matching operation has already arrived or because the ring buffer has looped around. In both cases, some thread has made progress. The outer loop retries if the slot is deemed unusable. This loop is bounded by the starvation_counter, ensuring that it cannot continue indefinitely. The final loop is the list-level loop in denqueue, which retries only if a ring append fails due to another thread appending. Thus, investigation of loops proves that the insert and remove_request methods of the MPDQ are lock-free.

5. EXPERIMENTAL RESULTS

We evaluated our algorithms on a machine running Fedora Core 19 Linux with two 18-core, 2-way hyperthreaded Intel Xeon E5-2699 v3 processors at 3.6 GHz (i.e., with up to 72 hardware threads). Each core has private L1 and L2 caches; the last-level cache (45 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. Moving from 1 to 72 threads, we first placed a single thread on each core of one processor (1–18), then a single thread on each of the same core's hyperthreads (19–36). We subsequently filled

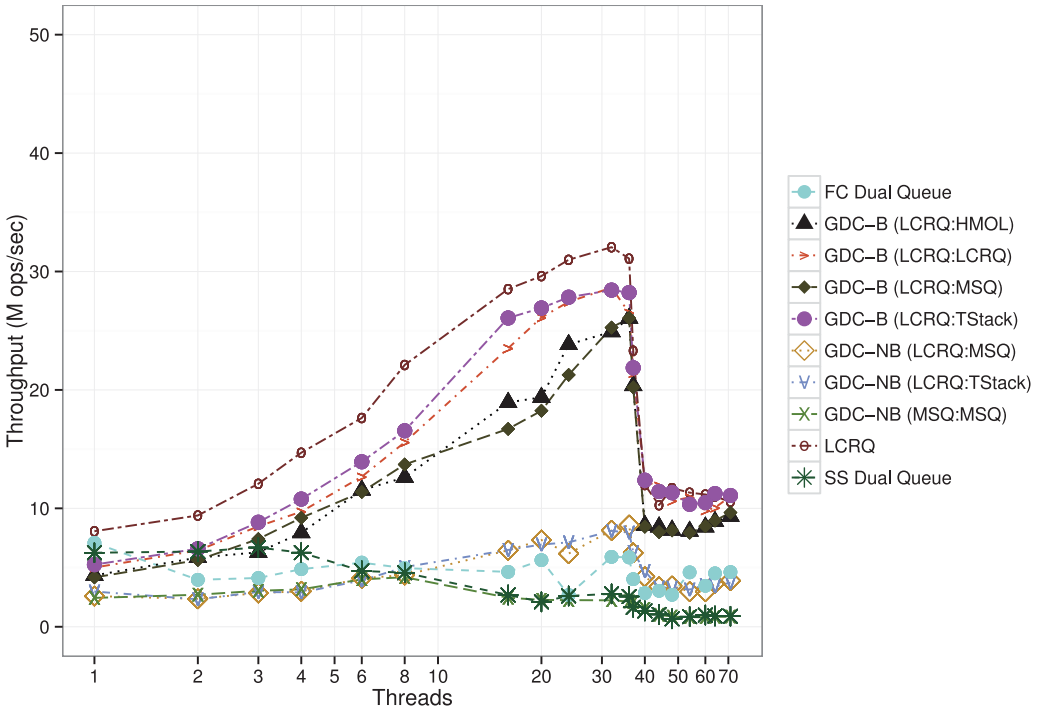

Fig. 15. Generic dual performance on the hot potato benchmark.

the second processor the same way, distributing first to physical cores (37–54) and then to their associated hyperthreads (55–72). Code was written in C++ and compiled using g++ 4.8.2 at the `-O3` optimization level. When a nonblocking memory allocator was appropriate, we used one adapted from the Rochester Software Transactional Memory (RSTM) package [Marathe et al. 2006].

5.0.4. Microbenchmark. To test the throughput of dual containers, we wanted to simulate as random an access pattern as possible. Unfortunately, purely random choice among insert and remove (enqueue and dequeue) allows for the possibility of deadlock when all threads call remove on a negative container.

To solve this problem, we used the *hot potato microbenchmark* [Izraelevitz and Scott 2014c]. This test, based on the children's game, allows each thread to access a dual structure randomly, choosing on each iteration whether to insert or remove an element. However, at the beginning of the test, one thread inserts the *hot potato*, a special data value, into the container. If any thread removes the hot potato, it waits a set amount of time (1 ms in our tests) before reinserting the value, then continuing to randomly operate on the container. The hot potato eliminates the possibility of deadlock and allows the test to continue with minimal interaction among threads outside of the data structure.

To test a given structure, we ran the hot potato benchmark for a fixed time interval of ten seconds. We ran each test five times and report the maximum throughput of these runs. No large deviations among tests were noted for any of the queues. Note that, while we divide the generic dual results from those of the dual ring queue for clarity of presentation, Figures 15 and 16 use the same scale on the Y axisthus, the curves are directly comparable. We use a logarithmic scale on the X axis to increase readability

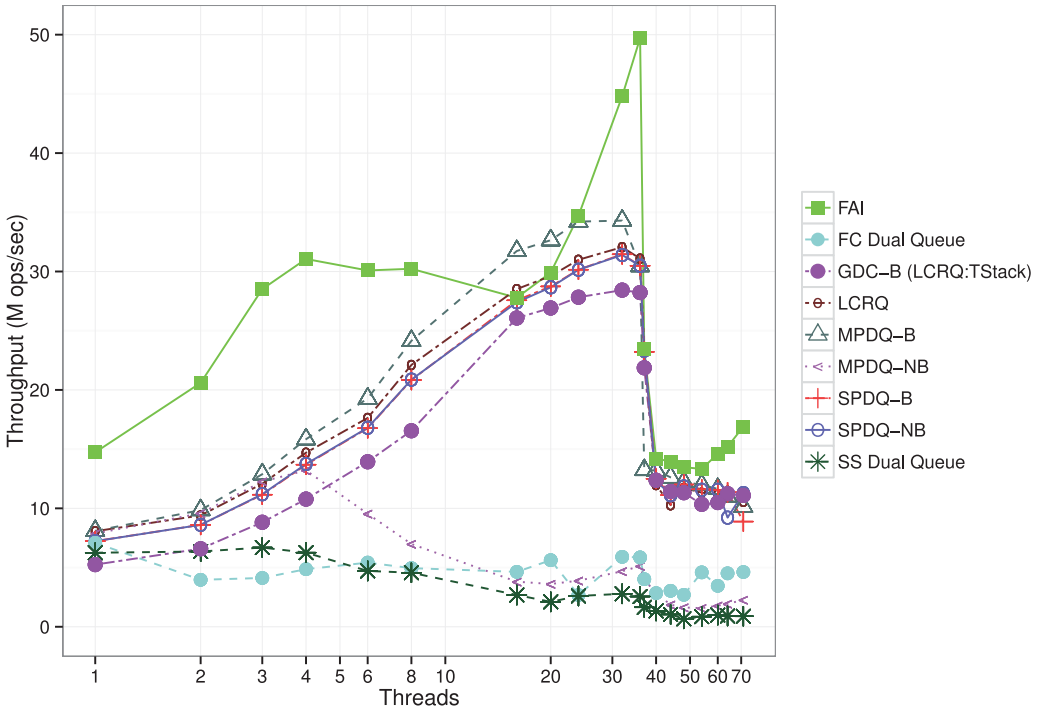

Fig. 16. Dual ring queue performance on hot potato benchmark.

at lower thread counts, where the most interesting effects occur; performance drops dramatically once threads span the boundary between sockets.

5.1. Generic Dual Results

Figure 15 shows the throughput of our generic dual container. We test several combinations of subcontainers, with and without the peek operation, to close the preemption window:

LCRQ(+), Treiber stack(–), blocking [GDC-B (LCRQ:TStack)]: The fastest combination. Uses the LCRQ of Morrison and Afek [2013] for data and the Treiber stack [Treiber 1986] for antidata.

LCRQ(+), Treiber stack(–), nonblocking [GDC-NB (LCRQ:TStack)]: A comparison to demonstrate the impact of closing the preemption window.

LCRQ(+), LCRQ(–), blocking [GDC-B (LCRQ:LCRQ)]: A FIFO dual queue, suitable for direct comparison to the MPDQ, SPDQ, or S&S dual queue.

LCRQ(+), M&S Queue(–), blocking [GDC-B (LCRQ:MSQ)]: Another FIFO dual queue, but with a less efficient antidata structure.

LCRQ(+), M&S Queue(–), nonblocking [GDC-NB (LCRQ:MSQ)]: Another comparison to demonstrate the impact of closing the preemption window, but for a FIFO structure.

M&S Queue(+), M&S Queue(–), nonblocking [GDC-NB (MSQ:MSQ)]: Demonstrates the baseline cost of our construction when compared directly to the S&S dual queue.

LCRQ(+), H&M Ordered List(–), blocking [GDC-B (LCRQ:HMOL)]: Orders waiting threads based on priority, using the lock-free ordered list of Harris [2001] and Michael [2002].

For comparison purposes, we also show other dual and total containers from prior work:

S&S Dual Queue: The algorithm of Scherer and Scott [2004].

FC Dual Queue: A best-effort implementation of a flat-combining dual queue, using the methodology of Hendler et al. [2010b].

LCRQ: The fast total ring queue of Morrison and Afek [2013].

All ring queues use a ring size of 2048 elements. In our realization of the 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 that their operations have completed simply return.

5.1.1. Optimizations. All GDC results employ optimizations not shown in Figure 2. First, we skip CAS operations when a precheck indicates that they will fail. Second, and more significantly, each operation prechecks the opposite subcontainer for empty (using the oppositeCheck functions), before inserting a placeholder. This precheck limits both the number of spurious memory allocations and the size of the subcontainers.

5.1.2. Performance. With the LCRQ as the data subcontainer and ignoring the preemption window, our generic dual container significantly outperforms traditional dual structures, including the S&S dual queue and the flat combining queue. Clearly, a fast base algorithm matters enormously. The antidata subcontainer also matters: using the Treiber stack over the M&S queue provides a roughly 25% speedup in the blocking case and even outperforms the LCRQ as an antidata subcontainer, presumably because LIFO ordering of waiting threads leads to better cache performance.

Closing the preemption window incurs a significant performance cost. With all threads competing to satisfy the same peeked-at placeholder, contention on that object becomes a bottleneck.

If mixed ordering disciplines are not required, the fastest overall performance comes from the MPDQ and SPDQ, as seen in the following section.

5.2. Custom Dual Results

Figure 16 demonstrates the performance of our custom fast dual ring queue algorithms. As in Section 5.1, we use a logarithmic X axis for the sake of visual clarity. We show throughput for the following:

SPDQ, MPDQ [SPDQ-B, MPDQ-B]: The blocking algorithms of Sections 4.2 and 4.3, respectively.

SPDQ nonblocking, MPDQ nonblocking [SPDQ-NB, MPDQ-NB]: The nonblocking variants described in Section 4.4.

M&S Queue, LCRQ [MSQ, LCRQ]: The nondual algorithms of Michael and Scott [1996] and of Morrison and Afek [2013], with an outer loop in which negative threads retry until they succeed in dequeuing data.

S&S Dual Queue, FC Dual Queue: Repeated from Section 5.1, the dual queue algorithm of Scherer and Scott [2004] and the flat-combining dual queue inspired by the work of Hendler et al. [2010b].

Generic Dual, LCRQ(+), Treiber stack(–), blocking [GDC-B (LCRQ:TStack)]: The fastest generic combination from Section 5.1, as a baseline against which to assess the performance improvement available from custom algorithms.

FAI: Another baseline, to obtain a sense of fundamental hardware limits. This curve simply plots the maximum rate at which threads can perform FAI operations on a shared counter.

As in Section 5.1, the various ring queues all use a ring size of 2048 elements.

5.2.1. Blocking Variants. As shown in Figure 16, our new algorithms have throughput significantly higher than that of 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 36 threads, when there is maximum parallelism without incurring chip-crossing overheads. The raw FAI test similarly scales well within a single chip, but it obtains a significant boost in throughput with the introduction of hyperthreads. With a tiny inner loop, this test benefits greatly from the shared L1 cache; the real algorithms, by contrast, perform enough work per iteration that the typical cache line has been "stolen" by another core before the sister hyperthread can use it. After moving to the second chip, all algorithms are limited by interconnect latency.

Interestingly, under some conditions, the new dual queues and LCRQ may even outperform the single integer FAI test. 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.

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

5.2.2. 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 suspect that this happens because the window closing code runs only 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 effectively "handshake" with the previous positive thread, adding several cache misses to the critical path of the hot potato test.

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 reenqueuing a hot potato slows everything down by causing queues to spend more time in a negative polarity.

6. CONCLUSION

In this work, we have presented improvements to both the usability and performance of dual data structures.

In Section 3, we introduced a generic construction for dual containers, allowing the programmer to choose orderings independently for data and pending requests, by combining any positive (data) container with almost any negative (antidata) container (specifically, any that supports the peek and remove_conditional operations). Our proofs demonstrate that the construction correctly combines the ordering semantics of the underlying containers, preserves obstruction freedom, and guarantees the immediate wakeup and contention freedom required of a dual data structure. Our experimental results suggest that while the mixing of ordering disciplines incurs nontrivial cost, the resulting containers are still fast enough to be quite useful in practice. In particular,

blocking ("not quite nonblocking") variants that use the LCRQ of Morrison and Afek [2013] for the data subcontainer outperform all dual containers published prior to this work.

In Section 4, we introduced two custom dual containers that extend the LCRQ 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 interthread communication.

In their basic form, our fast algorithms are again "almost nonblocking." We also introduced fully lock-free variants. We believe that 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.

APPENDIXES

A. GENERIC DUAL CONTAINERS PSEUDOCODE

Pseudocode for the generic dual container construction appeared in Figures 2 and 3. To be used as an antidata subcontainer in the fully nonblocking version of the construction, an existing container must be modified to support the peek and remove_conditional methods. We illustrate this modification using the Treiber stack [Treiber 1986].

In our implementation, the returned key from peek is the value of the top pointer. Any operation that modifies the stack must change this pointer and will force the paired remove_conditional operation to fail. Similar techniques can be used in most nonblocking data structures.

For presentation purposes, the following code assumes sequential consistency, allocates fresh nodes in push to avoid the ABA problem, and assumes the existence of automatic garbage collection. Our C++ code performs manual storage reclamation and uses counted pointers and a type-preserving allocator to avoid the ABA problem.

```
537 struct Node {
538   Object* val;
539   Node* down;
540   Node(Object* o){val=o;down=NULL;}
541 };
542
543 class TreiberStack {
544   Node* top = NULL;
545 };
546
547 bool TreiberStack:push(Object* obj) {
548   Node* newNode;
549   Node* topCopy;
550   newNode = new Node(obj);
551   while (true) {
552     topCopy = top; // read top pointer
553     newNode→down = topCopy;
554     // swing top; finished if success
555     if (CAS(&top, topCopy, newNode)) {
556       return true;
557     }
558   }
```

```
569     // swing top; finished if success
570     if (CAS(&top, topCopy, newTop)) {
571       return topCopy→val;
572     }
573   }
574 }
575
576 (Key, Object*) TreiberStack:peek() {
577   Node* topCopy;
578   Key key;
579   Object* obj;
580   do {
581     topCopy = top;
582     key = (Key)topCopy;
583     if (topCopy ≠ NULL) {
584       obj = topCopy→val;
585     }
586     else obj = NULL; // if empty
587   } while (key ≠ top);
588   return (key,obj);
589 }
590
```

```
559  }
560
561  Object* TreiberStack:pop() {
562    Node* topCopy;
563    Node* newTop;
564    while (true) {
565      topCopy = top; // read top pointer
566      // check if empty
567      if (topCopy == NULL) return EMPTY;
568      newTop = topCopy→down; // get new top
```

```
591  bool TreiberStack:remove_conditional(
592   Key key) {
593    Node* topCopy;
594    Node* newTop;
595    topCopy = (Node*)key;
596    newTop = topCopy→down;
597    // swing top; finished if success
598    if (CAS(&top, topCopy, newTop)) return true;
599    return false; // key wasn't top anymore
600  }
```

B. FAST DUAL QUEUES PSEUDOCODE

This appendix contains pseudocode for all algorithms relating to Section 4. For a detailed explanation of the LCRQ, see Morrison and Afek [2013].

B.1. Linked Concurrent Ring Queue (LCRQ)

```
602  tuple Slot {
603    bool safe; // 1 bit
604    int idx; // 31 bits
605    Object* val; // 32 bits (int or pointer)
606    // padded to cache line size
607  };
608
609  class CRQ { // fields are on distinct cache lines
610    int head; // 32 bits
611    ⟨bool closed, int idx⟩ tail; // 1, 31 bits
612    CRQ* next;
613    Slot ring[R]; // initially ring[i] = ⟨1, i, NULL⟩ ∀i
614  };
615
616  class LCRQ {
617    CRQ* head, tail;
618  };
619
620  void CRQ:enqueue(Object* arg) {
621    int h, t;
622    Slot* slot;
623    bool closed, safe;
624    Object* val; // 32 bit int or pointer
625    int idx;
626    int starvation_ctr = 0;
627
628    while (true) {
629      ⟨closed, t⟩ = FAI(&self.tail);
630      if (closed) return CLOSED;
631      slot = &ring[t % R];
632      ⟨safe, idx, val⟩ = *slot;
633
634      // verify slot is empty
635      if (val == NULL) {
636        // verify allowed index
```

```
662      Object* val;
663      int idx;
664
665      while (true) {
666        h = FAI(&head);
667        slot = &ring[h % R];
668        while (true) {
669          ⟨safe, idx, val⟩ = *slot;
670          // slot not empty
671          if (val ≠ NULL) {
672            // attempt ideal dequeue
673            if (idx == h) {
674              if (CAS(slot, ⟨safe, h, val⟩,
675                    ⟨safe, h + R, NULL⟩)) {
676                return val;
677              }
678            }
679            // failed to dequeue: signal unsafe
680            // to prevent corresponding enqueue
681            else if (CAS(slot, ⟨safe, idx, val⟩,
682                  ⟨0, idx, val⟩)) {
683              goto deq_fail;
684            }
685          }
686          // slot is empty: failed to dequeue;
687          // signal slot via index to prevent
688          // corresponding enqueue
689          else {
690            if (CAS(slot, ⟨safe, idx, NULL⟩,
691                  ⟨safe, h + R, NULL⟩))
692              goto deq_fail;
693            }
694        } // end of inner while loop
695
696    deq_fail:
```

```
637         if ((idx ≤ t)
638             // verify safe or fixable unsafe
639             && (safe == 1 || head ≤ t)
640             // then attempt enqueue
641             && CAS(slot, ⟨safe, idx, NULL⟩,
642               ⟨1, t, arg⟩)) return OK
643       }
644       // enqueue at this index failed, due to signal
645       // or occupied slot; check for full or starving
646       h = head;
647       starvation_ctr++;
648       if (t − h ≥ R
649           || starvation_ctr > STARVATION) {
650         TAS(&tail.closed); // close CRQ
651         return CLOSED;
652       }
653       // else we failed to enqueue and queue is not
654       // full, so we skip this index and try the next
655     }
656   }
657
658   void CRQ:dequeue() {
659     int h, t;
660     Slot* slot;
661     bool closed, safe;
722     }
723   }
724
725   void LCRQ:enqueue(Object* x) {
726     CRQ* crq, newcrq;
727
728     while (true) {
729       crq = tail;
730       // make sure tail is updated fully
731       if (crq→next ≠ NULL) {
732         (void)CAS(&tail, crq, crq→next)
733         continue;
734       }
735
736       // attempt enqueue
737       if (crq→enqueue(x) ≠ CLOSED) return;
738
739       // if queue is closed, append new CRQ
740       newcrq = new CRQ();
741       newcrq→enqueue(x);
742       if (CAS(&crq.next, NULL, newcrq)) {
743         (void)CAS(&tail, crq, newcrq);
```

```
697       // we failed to dequeue at this index,
698       // so check for empty
699       ⟨closed, t⟩ = tail;
700       if (t ≤ h + 1) {
701         fixState();
702         return EMPTY;
703       }
704       // else we failed to dequeue at this index
705       // and queue is not empty; try at next index
706
707     } // end of outer while loop
708   }
709
710
711
712   void CRQ:fixState() {
713     int h, t;
714
715     while (true) {
716       h = head; t = tail.idx;
717       if (h ≤ t)
718         // nothing to do as queue is consistent
719         return;
720       if (CAS(&tail, t, h)) // swing tail forward
721         return;
744       return;
745     }
746   }
747   }
748
749   Object* LCRQ:dequeue() {
750     CRQ* crq;
751     Object* v;
752
753     while (true) {
754       crq = head;
755       v = crq→dequeue()
756       if (v ≠ EMPTY) return v;
757       if (crq→next == NULL) return EMPTY;
758
759       // swing head if the current head has
760       // remained empty (and is thus closed)
761       v = crq→dequeue();
762       if (v ≠ EMPTY) return v;
763       (void)CAS(&head, crq, crq→next);
764     }
765   }
```

B.2. Lock-free SPDQ

```
766 Object* SP_CRQ:internal_dequeue_NB
767     (Object* arg) {
768
769   ⟨bool closed, int idx⟩ h, t; // 1, 31 bits
770   Slot* slot;
771   bool closed, safe;
772   int idx;
773   Object* val;
774
775   h = FAI(&head);
776
777   bool paused = false;
778
779   while (true) {
780     slot = &this→ring[h.idx % R];
781     ⟨safe, idx, val⟩ = *slot;
782
783     // find the wavefront
784
785     if (idx > h.idx) {
786       // behind the wavefront; move up
787       h.idx++;
788       continue;
789     }
790
791     if (idx < h.idx) {
792       // too far ahead — we've lapped
793       h.idx = h.idx − R;
794       continue;
795     }
796
797     // now we know our index matches the slot
798     // verify we are on the wavefront
799     if (h.idx ≠ 0
800         && ring[(h.idx − 1) % R].idx ==
801         h.idx − 1) {
802       // we aren't on the wavefront,

840         else {
841           h = FAI(&head);
842           continue;
843         }

803       // so wait and check again
804       if (!paused) {
805         usleep(1);
806         paused = true;
807         continue;
808       } else {
809         // we already timed out, so search
810         // backward for the wavefront
811         h.idx−−;
812         continue;
813       }
814     }
815     // now we know we're at the wavefront,
816     // so dequeue
817
818     if (val ≠ NULL) {
819       // slot is nonempty; try to dequeue
820       if (((Waiter*)val)→satisfy(arg)) {
821         (void)CAS(&slot, ⟨safe, h.idx, val⟩,
822           ⟨safe, h.idx + R, NULL⟩);
823         return OK;
824       } else {
825         // someone beat us to the wait structure
826         (void)CAS(&slot, ⟨safe, h.idx, val⟩,
827           ⟨safe, h.idx + R, NULL⟩);
828         h.idx++; // behind wavefront; move up
829         continue;
830       }
831     } else {
832       // if slot is empty, mark for counterpart
833       if (CAS(&slot, ⟨safe, h.idx, val⟩,
834           ⟨safe, h.idx + R, NULL⟩)) {
835         ⟨closed, t⟩ = tail;
836         if (t ≤ h + 1) {
837           fixState();
838           return EMPTY;
839         }

844       }
845     } // end else val is NULL
846   } // end of main while loop
847 }
```

B.3. Lock-free MPDQ

```
848 Object* MP_CRQ:denqueue_NB(Object* arg) {
849   ⟨bool closed, int idx⟩ p; // 1, 31 bits
850   ⟨bool closed, int idx⟩* my_cntr;
851   ⟨bool closed, int idx⟩* their_cntr;
852   MP_Slot* slot;
853   bool closed;
854   bool safe, safe_pr;
855   int idx, idx_pr;
856   Object* val, *val_pr;
857   bool slot_polarity, slot_polarity_pr;
858
859   MP_Slot* slot_prev;
860   int starvation_counter = 0;
861   int close_idx = 0;
862   bool paused = false;
863
864   // get heads
865   my_cntr = &data_idx;
866   their_cntr = &antidata_idx;
867

904     // and previous operation has finished
905     slot_prev = ring[(p.idx − 1) % R];
906     ⟨safe_pr, slot_polarity_pr, idx_pr, val_pr⟩ =
907         *slot_prev;
908     if (p.idx ≠ 0
909         && (idx_pr == (idx − 1))
910         && ((slot_polarity_pr == ANTIDATA
911               && val_pr ≠ NULL)
912             || val_pr == NULL)) {
913       // not ahead; wait and check again
914       if (!paused) {
915         usleep(1);
916         paused = true;
917         continue;
918       } else {
919         // already timed out; search backward
920         p.idx−−;
921         continue;
922       }
923     }
```

```
868   p = FAI(my_cntr);
869   while (true) {
870     // check for closed
871     if (p.closed == 1 || data_idx.closed == 1) {
872       close_idx =
873           discovered_closing(p.idx, DATA);
874       if (close_idx ≤ p.idx) return CLOSED;
875     }
876     // close queue if we fail to make progress or
877     // if the queue is getting full (to ensure the
878     // closed index is an actual slot)
879     if ((data_idx.idx − anti_data.idx
880         ≥ (R − 2 ∗ MAX_THREADS)
881       || starvation_counter > STARVATION)
882       && !p.closed) {
883       data_idx.close();
884       close_idx = discovered_closing(
885         data_idx.idx, DATA);
886       if (close_idx ≤ p.idx) return CLOSED;
887     }
888
889     slot = &ring[p.idx % R];
890     ⟨safe, slot_polarity, idx, val⟩ = ∗slot;
891     starvation_counter++;
892
893     // find wavefront
894     if (p.idx < idx) { // behind wavefront
895       p.idx++;
896       continue;
897     }
898     if (idx < p.idx) { // lapped ahead of wavefront
899       p.idx = p.idx − R;
900       continue;
901     }
902
903     // verify at wavefront
924
925     // on the wavefront, can operate
926     if (val ≠ NULL) {
927       // slot is nonempty; try to dequeue
928       if (idx == p.idx && slot_polarity ≠ DATA) {
929         if (((Waiter∗)val)→satisfy(arg)) {
930           (void)CAS(&slot, ⟨safe, p.idx, val⟩,
931             ⟨safe, p.idx + R, NULL⟩);
932           return OK;
933         } else {
934           // someone beat us to
935           // the wait structure
936           (void)CAS(&slot, ⟨safe, p.idx, val⟩,
937             ⟨safe, p.idx + R, NULL⟩);
938           p.idx++;
939           continue;
940         }
941       } else if (slot_polarity == DATA) {
942         // slot has wrong polarity;
943         // we lost race to enqueue
944         p.idx++;
945         continue;
946       }
947     } else {
948       // slot is empty; try to enqueue self
949       if (safe == 1 || antidata_idx.idx ≤ p.idx) {
950         // enqueue
951         if (CAS(&slot, ⟨safe, p.idx, NULL⟩,
952             ⟨1, p.idx, arg⟩))
953           return OK;
954       }
955       // else something got in the way —
956       // either counterpart or competition
957     }
958   } // end outer while loop
959 } // end dequeue
```

REFERENCES

Yehuda Afek, Guy Korland, Maria Natanzon, and Nir Shavit. 2010. Scalable producer-consumer pools based on elimination-diffraction trees. In *Proceedings of the 16th International Euro-Par Conference on Parallel Processing*. Springer, Ischia, Italy, 151–162.

Dave Dice. 2014. PTLQueue: A scalable bounded-capacity MPMC queue. Retrieved February 15, 2017 from https://blogs.oracle.com/dave/entry/ptlqueue_a_scalable_bounded_capacity. (2014).

Eric Freudenthal and Allan Gottlieb. 1991. Process coordination with fetch-and-increment. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'91)*. ACM, Santa Clara, CA, 260–268.

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* 5, 2, 164–189.

Timothy L. Harris. 2001. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing (DISC'01)*. Springer-Verlag, Lisbon, Portugal, 300–314.

Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. 2010b. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'10)*. ACM, Santorini, Greece, 355–364.

Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. 2010a. Scalable flat-combining based synchronous queues. In *Proceedings of the 24th International Conference on Distributed Computing (DISC'10)*. Springer, Cambridge, MA, 79–93.

Joseph Izraelevitz and Michael L. Scott. 2014b. Brief announcement: A generic construction for nonblocking dual containers. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC'14)*. ACM, Paris, France, 53–55.

Joseph Izraelevitz and Michael L. Scott. 2014a. Brief announcement: Fast dual ring queues. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'14)*. ACM, Prague, Czech Republic, 73–75.

Joseph Izraelevitz and Michael L. Scott. 2014c. *Fast Dual Ring Queues*. Technical Report TR 990. Department of Computer Science, University of Rochester.

Joseph Izraelevitz and Michael L. Scott. 2014d. *A Generic Construction for Nonblocking Dual Containers*. Technical Report TR 992. Department of Computer Science, University of Rochester, Rochester, NY.

Virendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer III, and Michael L. Scott. 2006. Lowering the overhead of nonblocking software transactional memory. In *1st ACM SIGPLAN Workshop on Transactional Computing (TRANSACT'06)*. Ottowa, ON, Canada.

Maged M. Michael. 2002. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'02)*. ACM, Winnipeg, MB, Canada, 73–82.

Maged M. Michael. 2004. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems* 15, 8, 491–504.

Maged M. Michael and Michael L. Scott. 1996. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC'96)*. ACM, Philadelphia, PA, 267–275.

Adam Morrison and Yehuda Afek. 2013. Fast concurrent queues for x86 processors. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'13)*. ACM, Shenzhen, China, 103–112.

Daniel Orozco, Elkin Garcia, Rishi Khan, Kelly Livingston, and Guang R. Gao. 2012. Toward high-throughput algorithms on many-core architectures. *ACM Transactions on Architecture and Code Optimization* 8, 4, Article 49, 21 pages.

Davide Pasetto, Massimiliano Meneghin, Hubertus Franke, Fabrizio Petrini, and Jimi Xenidis. 2012. Performance evaluation of interthread communication mechanisms on multicore/multithreaded architectures. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing (HPDC'12)*. ACM, Delft, The Netherlands, 131–132.

William N. Scherer, III. 2005. Personal communication.

William N. Scherer, III, Doug Lea, and Michael L. Scott. 2005. A scalable elimination-based exchange channel. In *Workshop on Synchronization and Concurrency in Object-Oriented Languages*. San Diego, CA. In conjunction with OOPSLA'05.

William N. Scherer, III, Doug Lea, and Michael L. Scott. 2009. Scalable synchronous queues. *Communications of the ACM* 52, 5, 100–108.

William N. Scherer, III and Michael L. Scott. 2004. Nonblocking concurrent data structures with condition synchronization. In *Proceedings of the 18th International Conference on Distributed Computing (DISC'04)*. Springer, Amsterdam, The Netherlands, 174–187.

Michael L. Scott. 2013. *Shared-Memory Synchronization*. Morgan & Claypool Publishers, San Francisco, CA.

R. Kent Treiber. 1986. *Systems Programming: Coping with Parallelism*. Technical Report RJ 5118. IBM Almaden Research Center, San Jose, CA.

James M. Wilson. 1988. *Operating System Data Structures for Shared Memory MIMD Machines with Fetch-and-add*. Ph.D. Dissertation. New York University, New York, NY.

Received March 2015; revised August 2016; accepted December 2016