A Generic Construction for Nonblocking Dual Containers*

Joseph Izraelevitz Michael L. Scott

Technical Report #992

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

May 2014

Abstract

A dual container has the property that when it is empty, the remove method will insert an explicit reservation ("antidata") into the container, rather than returning an error flag. This convention gives the container explicit control over the order in which pending requests will be satisfied once data becomes available. The dual pattern also allows the method's caller to spin on a thread-local flag, avoiding memory contention. In this paper we introduce a new nonblocking construction that allows any nonblocking container for data to be paired with almost any nonblocking container for antidata. This construction provides a composite ordering discipline—e.g., it can satisfy pending pops from a stack in FIFO order (for fairness) satisfy pending remove_mins in LIFO order (to maximize retention of thread cache footprint), or satisfy pending dequeues in order of thread priority.

1 Introduction

As originally codified by Herlihy and Wing [6], nonblocking concurrent data structures support only *total* methods, "because it is unclear how to interpret the nonblocking condition for partial operations" [5, p. 128]. Dual data structures [14] extend the definition to *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 else modifies the data structure in a way that makes the caller's interest in the precondition (its *reservation*) visible to subsequent operations. The data structure can then assume full control over the order in which reservations should be satisfied when data becomes available, and each waiting thread can spin on a separate local flag, avoiding memory contention.

Nonblocking dual containers in particular have proven very useful. Scherer et al. [13] report that dual versions of the java.util.concurrent.SynchronousQueue improved the performance of task dispatch by as much as an order of magnitude. (A *synchronous queue* is one in which both enqueue and dequeue methods wait for a matching operation. The Java versions are based on the Treiber stack [15] and the M&S queue [10].) The Java library also includes a dual Exchanger class, in which operations of a single type "match up." Other synchronous queues include the flat combining version of Hendler et al. [4] and the elimination-diffraction trees of Afek et al. [1]. Recently [7], we have developed two fast dual queues based on the LCRQ algorithm of Morrison and Afek [11].

To the best of our knowledge, all previously published dual containers have used a single structure to hold either data or reservations ("antidata"), depending on whether there have been more inserts or remove

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

requests in the set of operations completed to date (in some cases the structure may also contain already-*satisfied* reservations, 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 pattern has two significant drawbacks. First, converting 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 reservation it must remove the reservation *and* unblock the waiting thread as a single atomic operation. Second, since the same structure is used to hold either data or reservations, straightforward conversion will apply the same ordering discipline to each.

There are times when it may be highly advantageous to use different orders for data and antidata. In a work queue, for example, the data order might be FIFO (for fairness) or sorted by request priority. At the same time, the antidata order (used to order worker threads when data is not available) might be FIFO to balance workload, or LIFO to maximize cache locality. Scherer et al. report that LIFO antidata order dramatically improves the performance of work queues in the java.util.concurrent standard library [13]. In the current version of the library, however, that order is only available when data is also LIFO. On a heterogeneous machine, antidata might be sorted by core type to maximize throughput (by keeping the fastest cores busy) or to minimize energy consumption (by keeping the coolest cores busy). More complex thread scheduling schemes—e.g., to respect machine topology—might optimize for other metrics.

In unpublished work (alluded to in Sec. 3.7 of his dissertation [12]), Scherer developed "quack" and "steue" algorithms that mixed FIFO and LIFO disciplines—one for data and the other for antidata—but each such construction is a unique, nontrivial contribution. Our contribution in this paper is a *generic* construction to join separate containers for data and antidata. Any existing concurrent container can be used for the data side; to preserve nonblocking (specifically, obstruction-free) progress on the antidata side, we require that the remove method be partitioned into a peek method and a separate remove_conditional (this partition is typically straightforward).

We introduce our construction in Section 2. It requires no hardware support beyond the usual load, store, and compare_and_swap (CAS). Section 3 outlines 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. Both safety and liveness are nontrivial, involving linearization points that are identified by reasoning on the history. Section 4 presents microbenchmark results, showing reasonable performance for a variety of combinations of data and antidata structures.

2 The Generic Dual Construction

As suggested in the introduction, we build a nonblocking dual container using two underlying "subcontainers": one for data and one for antidata. When a thread calls the public insert method of the outer container, we refer to its operation (and sometimes the thread itself) as having *positive polarity*. When a thread calls the public remove method, we refer to its operation (and sometimes the thread itself) as having *negative polarity*.

We maintain the invariant that at any given linearization point, at most one of the underlying subcontainers is nonempty. Thus, in a positive operation, 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 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. 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*. The only asymmetry—and it is a crucial one—is that the public insert method is total, while remove is partial: negative threads must wait, spinning on a local variable, until a datum is available.

2.1 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. 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*. The key can then be passed to a subsequent call to remove_conditional. Between a peek that returns (v, k) and the first remove_conditional that takes k as argument, we require that any intervening peek also return (v, k): subcontainer implementations can simply cache the "current" (v, k) pair (see the Appendix for an example). At the linearization point of the original peek, v must be the smallest antidatum present under $<^-$. On invocation, remove_conditional removes v if it has remained the smallest element under $<^-$; otherwise it does nothing. 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 that the outer container is linearizable and obstruction free. As it turns out, many nonblocking container objects can be converted easily to support peek and remove_conditional. The experiments reported in Section 4 employ converted versions of the Treiber stack [15] and M&S queue [10]. A similar conversion could be applied to the H&M sorted list [2, 8], a nonblocking skip list, or any of various other structures.

2.2 Placeholders

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: unvalidated, aborted, validated, and satisfied. An unvalidated 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 validated 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 "mixed" with the corresponding operation.

On beginning a positive or negative operation on the outer container, we first store an unvalidated placeholder in the subcontainer of like polarity. We then check for emptiness of the opposite subcontainer by repeatedly removing elements. If we find a validated placeholder, we mix it with our own data or antidata, transition it from validated to satisfied, and return, leaving our own unvalidated placeholder behind. If we find an unvalidated placeholder, we abort it, indicating that it has been removed from its subcontainer and that any information the owning thread may have had regarding the polarity of the outer container is now

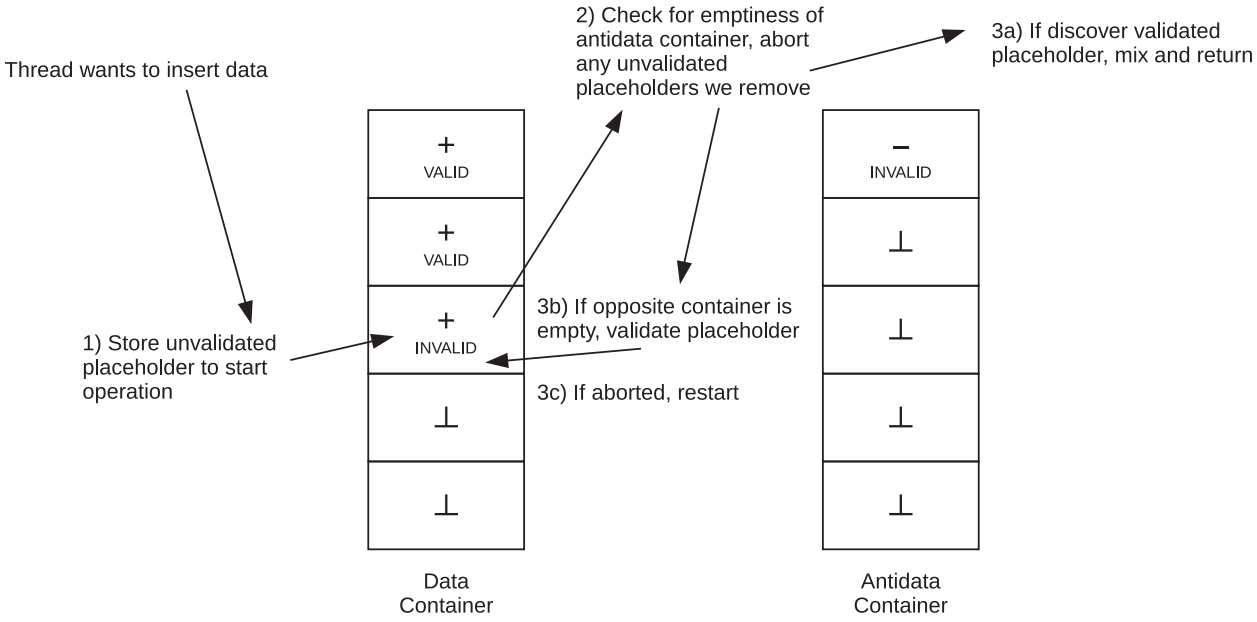

Figure 1: Execution of generic dual container (blocking variant)

out of date. Finally, if we discover that the opposite subcontainer is empty, we 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.

2.3 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 Scherer and Scott place strict limits on this blocking [14]. 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.) The description of our construction above 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 T, instead of simply removing a placeholder from the negative subcontainer, first peeks at the head placeholder and satisfies or aborts it. Only then does it remove that placeholder from the subcontainer. Any other thread that discovers a satisfied or aborted placeholder can help T's operation by removing the placeholder for it. By updating placeholders while they are still in the negative subcontainer, we order all waiting threads, guaranteeing that they are able to return without further delay.

As we shall see in Section 4, closing the preemption window incurs a nonnegligible performance penalty. If all one wants in practice is a fast shared buffer, the "not quite nonblocking" version of our construction may actually be preferred.

```
1  tuple placeholder {
2    Object contents = NULL;
3    bool val, abt, sat = ⟨false, false, false⟩;
4    const VAL= ABT= SAT= true;
5    bool satisfy(Object v) {
6      return CAS(this, ⟨NULL, VAL, !ABT, !SAT⟩,
7        ⟨v, VAL, !ABT, SAT⟩);
8    }
9  };
10 class generic_dual {
11   container* subcon[2];
12   bool obstruction_free;
13   const DATA= 0, ANTI= 1;
14   generic_dual(container *dc, *ac) {
15     subcon[DATA] = dc;
16     subcon[ANTI] = ac;
17   }
18 };
19
20 Object generic_dual:remove() {
21   return remsert(NULL, ANTI);
22 }
23 Object generic_dual:insert(Object val) {
24   return remsert(val, DATA);
25 }
26
27 Object generic_dual:remsert(Object val, bool polarity) {
28   placeholder *ph, *oph;
29   key k;
30   bool nb = (polarity==DATA && obstruction_free);
31     // use peek + remove_cond only if positive
32     // polarity and running nonblocking variant
33   while (true) {
34     ph = new placeholder; ph→contents = val;
35     // start operation
36     subcon[polarity]→insert(ph);
37     while (true) { // start empty check loop
38       if (nb) (oph, k) = subcon[!polarity]→peek();
39       else oph = subcon[!polarity]→remove();
40       if (oph == EMPTY) break;
41     else { // attempt to abort oph
42       Object oval = oph→val;
43       if (CAS(oph, ⟨oval, !VAL, !ABT, !SAT⟩,
44           ⟨oval, !VAL, ABT, !SAT⟩)) {
45         // abort succeeded; if nonblocking, remove
46         // placeholder before continuing
47         if (nb) (void) subcon[!polarity]→remove_cond(k);
48       } else { // abort failed
49         if (nb) {
50           // might be validated, aborted, or satisfied
51           if (oph→satisfy(ph→val)) {
52             // we satisfied the placeholder, so return
53             (void) subcon[!polarity]→remove_cond(k);
54             return OK;
55           }
56           // else already satisfied or aborted
57           (void) subcon[!polarity]→remove_cond(k);
58         } else { // blocking variant;
59           // placeholder guaranteed to be validated
60           if (polarity==DATA) {
61             *oph = ⟨ph→val, VAL, !ABT, SAT⟩;
62             return OK;
63           }
64           else return oph→val;
65         }
66       } // end abort failed
67     } // end attempt to abort
68   } // end empty check loop
69
70   // now opposite container is empty;
71   // try to validate our placeholder
72   if (CAS(ph, ⟨val, !VAL, !ABT, !SAT⟩,
73       ⟨val, VAL, !ABT, !SAT⟩)) {
74     if (polarity==DATA) return OK;
75     // else spin waiting for data
76     while (!ph→sat) {}
77     return ph→val;
78   }
79   // else we were aborted; retry entire operation
80 } // end outer loop
81 } // end remsert
```

Figure 2: Pseudocode for the generic dual construction

2.4 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.

A placeholder tuple (a CAS-able word) contains both a value (datum or antidatum) and flags to indicate its state. To simplify exposition, we use three bits, organized as follows, to encode the four possible states: unvalidated = ⟨!VAL, !ABT, !SAT⟩; aborted = ⟨!VAL, ABT, !SAT⟩; validated = ⟨VAL, !ABT, !SAT⟩; satisfied = ⟨VAL, !ABT, SAT⟩. All other combinations of flags are unreachable. A binary polarity flag is used to choose between data and antidata subcontainers. These containers are assumed to be initialized before passing them to the generic_dual constructor.

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.

On entering the outer loop of remsert, we allocate memory for an unvalidated placeholder, which we then store in the subcontainer whose polarity matches that of the current operation. On entering the inner loop, we attempt to remove (or peek at) an "opposite placeholder" (oph) from the other subcontainer. Assuming 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—validated, satisfied, or aborted—since other threads may have access to it via peek. We first guess that it is validated, 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 60) and return.

On discovering an empty opposite container, we break out of the inner loop 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're a positive thread) or wait for our placeholder to be satisfied by a positive thread. If the validation CAS fails, we have encountered a conflict with another thread and been aborted. Since the opposite container may not be empty anymore, we loop back to the start of the outer loop.

As is usual in concurrent structures, we must coordinate across threads to determine when it is safe to free data. For the blocking variant, our garbage collection is handled by verifying both the inserter and remover of a given placeholder are finished with it using a counter. When the second thread marks the placeholder as abandoned it frees the tuple. 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 [9] (not shown in the pseudocode) for storage reclamation. For efficiency, we maintain thread-local pools of available placeholders.

3 Correctness

In this section we provide informal 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 2.1, 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 or insertion time) needed to drive the $<^+$ and $<^-$ relations.

In the framework of Scherer and Scott [14], 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 is not available, inserts a *reservation* (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*.

We consider only *well-formed* parallel 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 64 to return a ticket containing oph; (3) modify the code at line 76 to return a ticket containing ph; (4) create a tiny remove_followup method that inspects the ticket and returns the val field of the placeholder therein.

3.1 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.

3.1.1 The Two-order Container

Since a sequential object never has partial methods, 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.

3.1.2 Linearization points

To prove the safety of our generic dual container (GDC), it suffices to show that in any realizable parallel 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 obstruction_free flag is true, so nb is true (line 30) if and only if the current outer-level operation is an insert, and thus may need to satisfy a waiting thread.

Our linearization points are dynamic, and identified by reasoning on execution histories. Again to minimize confusion, we consider only histories in which all GDC operations have completed. (The extensions needed for uncompleted operations are tedious but straightforward.) We assume that each history includes all instructions performed within operations of the underlying nonblocking containers, and that the lineariza-

tion 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 51, we linearize its operation at the linearization point of the already-completed peek that was called at line 38.

2. If an operation returns a validated data placeholder at line 64, we linearize the operation at the linearization point of the already-completed remove that was called at line 39. (The return at line 62 is unreachable in the obstruction-free variant of the code.)

3. If a peek at line 38 or a remove at line 39 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 linearization point of the current peek or remove, in the order in which their respective line-36 inserts linearized.

A bit of study confirms that these three cases cover all possible paths through the code. 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. 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.

Theorem 1. *Any realizable history of the GDC containing only completed operations is equivalent to a legal 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 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 where the subcontainer of opposite polarity is empty, and for every trailing operation to linearize at a point where 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. Given that GDC subcontainers are assumed to be correct implementations of the TOC subcontainers, the GDC duplicates the semantics of the TOC. □

3.2 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 [14].

Theorem 2. *If variable obstruction_free is true (line 30) and arguments dc and ac refer to correct nonblocking containers (line 14), then the generic dual container is indeed obstruction free.*

Proof. Aside from the spin at line 76, which we eliminated in favor of repeated calls to a remove_followup method, the code of Figure 2 contains only two loops. The inner loop (line 37) removes elements repeatedly from a finite container, and terminates when it is empty. The outer loop (line 33) repeats only when the CAS at line 72 fails due to contention with another thread. In the absence of such contention, all operations complete in bounded time. □

Theorem 3. *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 we have $i^B \prec s^B \prec u^A$ (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 $r^A \prec i^A$. Moreover i^B must also be a trailing insert, meaning $r^B \prec i^B$. So 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. *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 Experimental Results

We implemented the generic dual, all subcontainers, and comparison structures in C++ 11 using its supported atomic operations. All code was compiled using gcc 4.8.2 at the `-O3` optimization level. We evaluated our algorithm on a Fedora Core 19 Linux machine. This machine has two six-core, two-way hyperthreaded Intel Xeon E5-2430 processors at 2.20 GHz, supporting up to 24 hardware threads. The L3 cache (15 MB) is shared by all cores of a given processor; L1 and L2 caches are per-core private. To maximize cache locality, we pinned each thread to its core, filling a processor first using each core and then each hyperthread before moving to the second processor.

4.1 Benchmark

To test the throughput of the generic dual, we wanted to simulate as random an access pattern as possible. Unfortunately, purely random choice among insert and remove 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* [7]. 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)

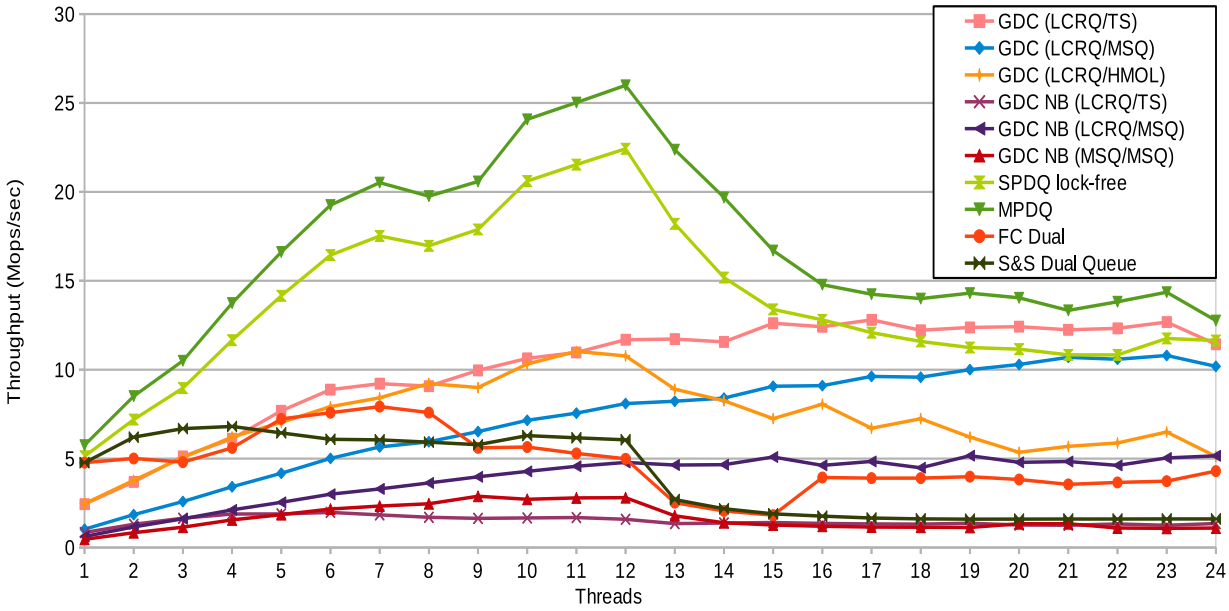

Figure 3: Throughput on the hot potato benchmark

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 two seconds. We ran each test five times. As is conventional, we report the maximum throughput across these runs; there was, however, little variation.

4.2 Tested Algorithms

We tested several combinations of subcontainers in the GDC, with and without the peek operation to close the preemption window:

LCRQ(+), Treiber stack(−), blocking: The fastest combination. Uses Morrison and Afek's LCRQ [11] for data and the Treiber stack [15] for antidata.

LCRQ(+), Treiber stack(−), nonblocking: A comparison to demonstrate the impact of closing the preemption window.

LCRQ(+), M&S Queue(−), blocking: A FIFO dual queue, suitable for direct comparison to the MPDQ, SPDQ, or S&S dual queue.

LCRQ(+), M&S Queue(−), nonblocking: Another comparison to demonstrate the impact of closing the preemption window, but with a more efficient peek than in the Treiber stack.

M&S Queue(+), M&S Queue(−), nonblocking: Demonstrates the baseline cost of our construction when compared directly to the S&S dual queue.

LCRQ(+), H&M Ordered List(−), blocking: Orders waiting threads based on priority, using the lock-free ordered list of Harris and Michael [2, 8].

For comparison purposes, we also tested several existing dual containers:

MPDQ: A fast but blocking dual queue derived from the LCRQ [7].

SPDQ lock-free: An alternative, lock-free derivative of the LCRQ [7].

S&S Dual Queue: The lock-free dual queue algorithm of Scherer & Scott [14].

FC Dual Queue: A flat-combining blocking dual queue inspired by the work of Hendler et al. [3, 7].

4.3 Optimizations

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

4.4 Performance

With the LCRQ as the data subcontainer, and ignoring the preemption window, our generic dual 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 consistent 25% speedup in the blocking case.

Closing the preemption window incurs a significant performance cost, especially when crossing the boundary between chips. With all threads competing to satisfy the same peeked-at placeholder, the cache line tends to bounce between processors. Additional contention arises when the peek modification requires internal caching of values (as in the Treiber stack—Appendix A).

If mixed ordering disciplines are not required, the fastest overall performance clearly comes from the MPDQ and SPDQ.

5 Conclusion

We have presented the generic dual container, a construction that supports arbitrary ordering on both data and pending requests. 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 variants that use the LCRQ for the data subcontainer outperform all dual containers published prior to this year.

In future work, we hope to implement additional optimizations and extend our construction to structures—e.g., sets and maps—that have other partial methods.

References

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

[2] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proc. of the 15th Intl. Symp. on Distributed Computing (DISC)*, Lisbon, Portugal, Oct. 2001.

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

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

[5] M. P. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, Jan. 1991.

[6] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[7] J. Izraelevitz and M. L. Scott. Brief announcement: Fast dual ring queues. In *Proc. of the 26th ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, Prague, Czech Republic, June 2014. Extended version published as TR 990, Computer Science Dept., Univ. Rochester, Jan. 2014.

[8] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proc. of the 14th ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, Winnipeg, MB, Canada, Aug. 2002.

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

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

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

[12] W. N. Scherer III. *Synchronization and Concurrency in User-level Software Systems*. PhD thesis, Computer Science Dept., Univ. of Rochester, Jan. 2006.

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

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

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

A Antidata Treiber Stack

When used as an antidata subcontainer in the fully nonblocking version of the GDC, an existing container object must be modified to support the peek and remove_conditional methods. For some containers—FIFO queues in particular—the modifications are trivial: a newly inserted element never takes precedence over an existing element, so peek continues to return the same element until remove_conditional is called. For other containers, we can implement a *caching* strategy that remembers a peeked value until it is removed. This appendix illustrates the caching technique in the context of the Treiber stack [15].

Note that peek and remove_conditional are implemented in two steps. The outer (public) methods keep track of the last returned value; they can be reused for other containers (e.g., priority queues). The inner (private) methods __peek and __remove_conditional are container specific; they use the unique top pointer as a key.

For presentation purposes, the code below assumes sequential consistency, allocates fresh nodes in push to avoid the ABA problem, and assumes the existence of automatic garbage collection. Our C++ code uses counted pointers.

```
struct Node {
  Object val;
  Node* down;
};
struct KeyVal {
  int key,
  Object val;
};
class TreiberStack{ // fields are on distinct cache lines
  Node* top = NULL;
  KeyVal* peekInfo = NULL;
};

bool TreiberStack::push(Object e) {
  Node* newNode, topCopy;
  newNode = new Node(e, NULL);
  while (true) {
    topCopy = top; // read top pointer
    newNode→down = topCopy;
    // swing top; finished if success
    if (CAS(&top, topCopy, newNode)) return true;
  }
}

Object TreiberStack::pop() {
  Node* topCopy;
  Node* newTop;
  while (true) {
    topCopy = top; // read top pointer
    // check if empty
    if (topCopy==NULL) return EMPTY;
    newTop = topCopy→down; // get new top
    // swing top; finished if success
    if (CAS(&top, topCopy, newTop)) return topCopy→val;
  }
}
```

```
KeyVal TreiberStack::__peek() {
  Node* topCopy;
  KeyVal kv;
  do {
    topCopy = top;
    kv.key = topCopy;
    if (topCopy!=NULL) kv.val = topCopy→val;
    else kv.val = 0; // if empty
  } while (kv.key!=top.all);
  return kv;
}

KeyVal TreiberStack::peek() {
  KeyVal* p;
  KeyVal kv;
  while (true) {
    p = peekInfo;
    if (p!=NULL) {
      kv = *p;
      if (p==peekInfo) return kv;
    } else {
      p = new KeyVal();
      kv = __peek();
      *p = kv;
      if (kv.val==0 || CAS(&peekInfo, NULL, p)) return kv;
    }
  }
}

bool TreiberStack::__remove_conditional(int key) {
  Node* topCopy;
  Node* newTop;
  topCopy = (Node*)key;
  newTop = topCopy→down;
  // swing top; finished if success
  if (CAS(&top, topCopy, newTop)) return true;
  return false; // key wasn't top anymore
}

Object TreiberStack::remove_conditional(int key) {
  KeyVal* p;
  p = peekInfo;
  if (p!=NULL && p→key==key) {
    (void) CAS(&peekInfo, p, NULL); // reset peekInfo
  }
  return __remove_conditional(key);
}
```