Brief Announcement: A Generic Construction for Nonblocking Dual Containers*

Joseph Izraelevitz and Michael L. Scott
Computer Science Department, University of Rochester
Rochester, NY 14627-0226, USA
{jhi1, scott}@cs.rochester.edu

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, or satisfy pending dequeues in order of thread priority.

1. INTRODUCTION

Dual data structures [10] extend the definition of nonblocking progress 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.

Scherer et al. [9] report that dual versions of the java.util.concurrent.SynchronousQueue improved the performance of task dispatch by as much as an order of magnitude. 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. [3] and the elimination-diffraction trees of Afek et al. [1]. Recently [4], we have developed two fast dual queues based on the LCRQ algorithm of Morrison and Afek [8].

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 reservations ("antidata"), depending on whether there have been more inserts or remove_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.

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 reservation it must both remove the reservation and unblock the waiting thread, all atomically. Second, since the same structure is used to hold either data or reservations, straightforward adaptations will apply the same ordering discipline to each.

We introduce a new construction that eliminates these drawbacks by joining a pair of subcontainers—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. Full details can be found in a companion technical report [5].

2. THE GENERIC DUAL CONSTRUCTION

When a thread calls the public insert method of our outer, "joined" 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*. Positive and negative operations are said to *correspond* when the former provides the datum for the latter.

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. 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. 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.

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, and remove returns the smallest.

*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.

PODC'14, July 15–18, 2014, Paris, France.
ACM 978-1-4503-2944-6/14/07
http://dx.doi.org/10.1145/2611462.2611510 .

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. The remove_conditional call removes the key's associated antidatum, so long as it its still the smallest antidatum under $<^-$. 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). At the time of the original peek, v must be the smallest antidatum present under $<^-$. As it turns out, many nonblocking container objects can be converted easily to support peek and remove_conditional.

To verify that one subcontainer is empty and insert into the other, atomically, 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 been "mixed" with antidata or data from its 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 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—implies that our construction is merely obstruction free.

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 [10]. 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." The description of our construction so far 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 any further delay.

3. CORRECTNESS

Detailed proofs of safety and liveness can be found in a companion technical report [5]. We assume an API, suggested by Scherer & Scott [10], that divides remove into separate remove_request and remove_followup methods. The first of these returns a special *ticket* value; the latter takes this ticket as argument and *succeeds* or *fails* depending on whether a matching insert has yet occurred in the object's history. We term the sequential object that exports this API a *two-order container* (TOC). The key difference between a conventional (total) concurrent container and a dual container is that spins on remove_followup can be entirely thread-local, with no impact on the contention observed by other threads.

Using the divided API, we establish the following:

THEOREM 1 (SAFETY). *Any realizable history of the generic dual container that comprises only completed operations is equivalent to a legal history of the TOC.*

THEOREM 2 (LIVENESS). *If both subcontainers of the generic dual container are correct and nonblocking, then the generic dual container itself is obstruction free.*

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 corresponding insert.

THEOREM 4 (CONTENTION FREEDOM). *An unsuccessful remove_followup() operation performs no remote memory accesses.*

4. EXPERIMENTAL RESULTS

We implemented the generic dual, all subcontainers, and comparison structures in C++ 11. All code was compiled using gcc 4.8.2 at the `-O3` optimization level, and run on Fedora Core 19 Linux. Our hardware was a machine with 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.

We measure throughput using a *hot potato* microbenchmark [4]. 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

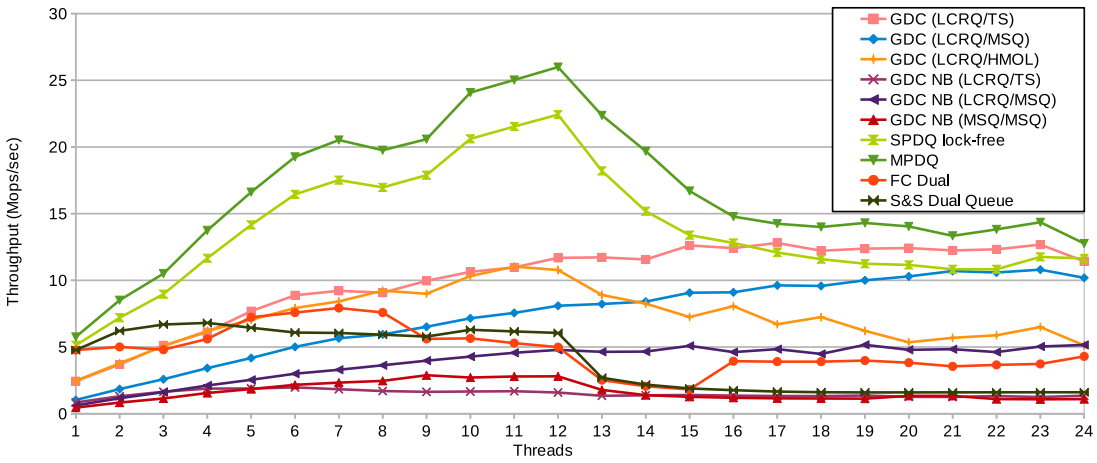

Figure 1: Performance on hot potato benchmark (second processor engaged at 13 cores)

the container. If any thread removes the hot potato, it waits a set amount of time ($1\mu s$) before reinserting the value then continues to randomly operate on the container. The reinsertion convention eliminates the possibility of deadlock. We ran each test for two seconds, and report the maximum throughput of five runs.

We test several combinations of subcontainers for our generic dual container (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 [8] for data and the Treiber stack [11] for antidata. **LCRQ(+), Treiber stack(−), nonblocking:** A comparison to demonstrate the impact of closing the preemption window. **LCRQ(+), M&S Queue** [7]**(−), 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, 6].

For comparison purposes, we also test existing nonblocking dual containers: **MPDQ:** a fast but blocking dual queue derived from the LCRQ [4]. **SPDQ lock-free:** an alternative, lock-free derivative of the LCRQ [4]. **S&S Dual Queue:** the dual queue of Scherer & Scott [10]. **FC Dual Queue:** a flat-combining dual queue inspired by the work of Hendler et al. [3, 4].

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). If all one wants in practice is a fast shared buffer, the "not quite nonblocking" version of our construction (with the preemption window intact) will deliver superior performance. Similarly, if mixed ordering disciplines are not required, the (non-generic) MPDQ and SPDQ will provide very fast FIFO handling of both data and antidata.

5. REFERENCES

[1] Y. Afek, G. Korland, M. Natanzon, and N. Shavit. Scalable producer-consumer pools based on elimination-diffraction trees. In *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 *15th Intl. Symp. on Distributed Computing (DISC)*, Lisbon, Portugal, Oct. 2001.

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

[4] J. Izraelevitz and M. L. Scott. Fast dual ring queues (brief announcement). In *26th ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, Prague, Czech Republic, June 2014. Expanded version available as TR 990, Dept. of Computer Science, Univ. of Rochester, Jan. 2014.

[5] J. Izraelevitz and M. L. Scott. A generic construction for nonblocking dual containers. Technical Report TR 992, Dept. of Computer Science, Univ. of Rochester, May 2014.

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

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

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

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

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

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