Brief Announcement: Fast Dual Ring Queues*

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

ABSTRACT

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

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

1. INTRODUCTION

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

Dual data structures, introduced by Scherer and Scott [6], extend the notion of nonblocking progress to *partial* methods—those that must wait for a precondition to hold. Informally, a partial method on a nonblocking dual structure is redefined to be a total method that either performs the original operation (if the precondition holds) or else modifies the data structure in a way that makes the caller's interest in the precondition (its *request*) visible to subsequent operations. This convention allows the code of the structure to control the order in which stalled methods will complete when preconditions are satisfied. It also makes it easy to ensure that stalled threads impose no burden on active threads—in particular, that they induce no memory contention.

The original dual structures [5], used for task dispatch in the Java standard library, were based on the well-known M&S queue [3] and Treiber stack [7]. In the intervening years, significantly faster concurrent queues have been devised—notably the linked concurrent ring queue (LCRQ) of Morrison and Afek [4]. 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 fixed-length buffer dubbed a concurrent ring queue (CRQ). Most operations on an LCRQ are satisfied by an individual ring queue, which uses a hardware fetch_and_increment (FAI) instruction to eliminate the contention normally associated with compare_and_swap (CAS).

Unfortunately, like most nonblocking queues, the LCRQ "totalizes" dequeue by returning an error code when the queue is empty. Threads that call dequeue in a loop, waiting for it to succeed, reintroduce contention, and their requests, once data is available, may be satisfied in an arbitrary (i.e., unfair) order. In this vein, we introduce two dual versions of the LCRQ (detailed treatment can be found in a technical report [2]). 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. Within a single multicore processor, throughput scales with the number of cores. Once threads are spread across processors, throughput remains 4–6× higher than that of the M&S-based structure.

Notation: "Dual" structures take their name from their ability to hold either data or antidata. If any datum can be used to satisfy any request, a quiescent dual structure will always be empty, populated only with data, or populated only with antidata. Dualism implies that a thread calling the public enqueue method may either enqueue data or dequeue antidata "under the hood." Likewise, a thread calling the public dequeue method may either dequeue data or enqueue antidata. When discussing dual algorithms, we will thus refer to the *polarity* of both threads and the structure, with a *positive* polarity referring to data, and a *negative* polarity referring to antidata.

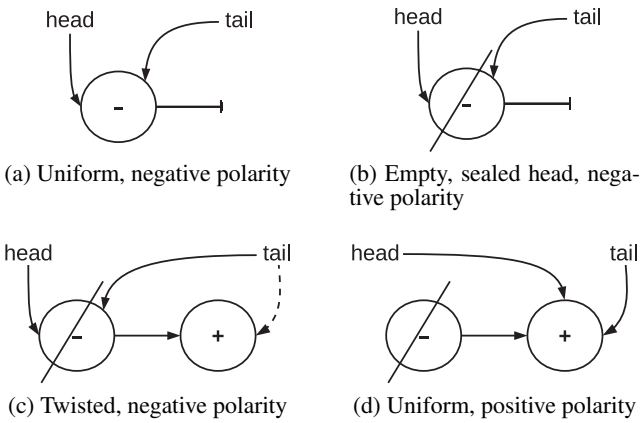
2. SINGLE POLARITY DUAL RING QUEUE

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

When the original LCRQ algorithm detects that a ring may be full, it *closes* the ring to prevent further insertions, and appends a new ring to the list. In the SPDQ, we introduce the notion of *sealing* an empty ring, to prevent further insertions, and we maintain the invariant that all non-sealed rings 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

*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.

SPAA'14, June 23–25, 2014, Prague, Czech Republic.
ACM 978-1-4503-2821-0/14/06.
http://dx.doi.org/10.1145/2612669.2612711.



(a) Uniform, negative polarity (b) Empty, sealed head, negative polarity

(c) Twisted, negative polarity (d) Uniform, positive polarity

Figure 1: Flipping the polarity of the SPDQ

head is sealed (closed and empty); **empty**—only one ring exists, and it is sealed.

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

Public enqueue (positive) operations inherit lock-free progress from the LCRQ algorithm [4]. In the worst case, a FAI-ing enqueuer may chase an unbounded series of FAI-ing dequeuers around a ring, 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 [3], the append can fail only if some other thread has appended a ring of its own, and the system will have made progress.

Because they may wait for data, public dequeue (negative) operations are more subtle. Scherer and Scott [6] model a partial operation on a dual structure in terms of a potentially unbounded sequence of nonblocking operations. The first operation linearizes the request for data of the calling thread, T. The last operation (the "successful follow-up") linearizes T's receipt of that data. In between, unsuccessful follow-up operations perform only local memory accesses, inducing no load on other threads. Finally, the total method (in our case, an enqueue) that satisfies T's pending request must ensure that no successful follow-up operation by another waiting thread can linearize in-between it (the satisfying operation) and an unsuccessful follow-up by T.

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. Without additional care, an antidata slot will be removed from consideration by other threads the moment the corresponding enqueuer performs its FAI. Mixing will happen afterward, leaving a "preemption window" in which the enqueuer, if it stalls, can leave the dequeuer waiting indefinitely. In practice, such occurrences can be expected to be extremely rare, and indeed the SPDQ performs quite well, achieving roughly 85% of the throughput of the original LCRQ while guaranteeing FIFO service to pending requests (Sec. 5). In Section 4 we will describe a modification to the SPDQ that closes the preemption window, providing fully lock-free behavior (in the dual data structure sense) at essentially no additional performance cost.

3. MULTI POLARITY DUAL RING QUEUE

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

In their original presentation of the CRQ [4], Morrison and Afek began by describing a hypothetical queue based on an infinite array. Similar intuition applies to the MPDQ. Since we are matching positive with negative operations, each thread, on arriving at a slot, must check if its partner has already arrived. If so, it mixes its data (antidata) with the antidata (data) of the corresponding operation. If not, it leaves its information in the slot (a negative thread also waits). As the program progresses, the data and antidata indices may repeatedly pass each other, flipping the polarity of the queue.

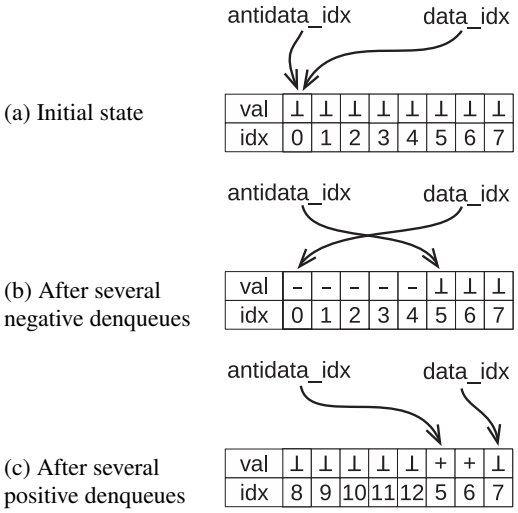


Figure 2: Flipping the polarity of the MPDQ

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

4. LOCK FREEDOM

The SPDQ and MPDQ, as presented so far, are eminently usable: they are significantly faster than the M&S-based dual queue of Scherer and Scott (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 reserved its matching operation, the negative thread is able to continue after a bounded number of steps by non-blocked threads.

For both algorithms we can close the preemption window by treating FAI as a mere suggestion to positive threads. Before they enqueue, they must verify that all smaller indices have already been satisfied by searching backwards around the ring buffer. The changes are smaller in the SPDQ case and (as we shall see in Sec. 5) have almost no impact on performance. The changes are larger in the MPDQ case, with a larger performance impact.

As established in the original CRQ algorithm, only dequeue operations can change the index of a given ring slot, allowing it to be reused for a later element in the logical sequence of data flowing

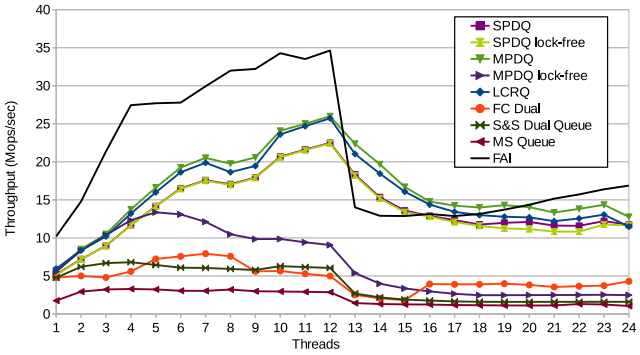


Figure 3: Performance on hot potato benchmark (second processor engaged at 13 cores)

through the queue. Thus, a discontinuity in indices (noticeable in Figure 2), indicates that a slot is ready and can be used to strictly order operations. In the SPDQ, since the preemption window only occurs when a positive thread dequeues from a negative ring, we can limit code changes to this single case. For the MPDQ, however, we must strictly order all positive operations. Since, at any point, the number of active threads is equal to the distance from the discontinuity to the head index, all threads will eventually succeed.

5. RESULTS

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

To obtain as random an access pattern as possible without admitting deadlock, we developed a *hot potato* test in which one thread, at the start of the test, enqueues a special value, called the *hot potato*. For the duration of the test, all threads randomly decide to enqueue or dequeue. If a thread ever dequeues the *hot potato*, however, it waits a small amount of time ($1\mu s$) and then re-enqueues it. During the wait, the queue has an opportunity to flip back and forth between data and antidata. We run the test for several seconds and report performance as throughput. For every queue, we ran five tests and took the maximum run. No large deviations among tests were noted for any of the queues.

In addition to the **SPDQ** and **MPDQ** of Sections 2 and 3, we consider: **SPDQ lock-free** and **MPDQ lock-free**—the nonblocking variants described in Section 4; **S&S Dual Queue**—the algorithm of Scherer & Scott [6]; **M&S Queue** and **LCRQ**—the non-dual algorithms of Michael & Scott [3] and of Morrison & Afek [4], with an outer loop in which negative threads retry until they succeed; and **FC Dual**—a best-effort implementation of a flat-combining dual queue, using the methodology of Hendler et al. [1].

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

As shown in Figure 3, our new algorithms have throughput significantly higher than any existing dual queue. Qualitatively, their scalability closely follows that of the LCRQ, across the full range of thread counts, while additionally providing fairness for dequeuing threads. The SPDQ is perhaps 20% slower than the LCRQ on average, presumably because of the overhead of "flipping." The MPDQ is about 5% *faster* than the LCRQ on average, presumably because it avoids the empty check and the contention caused by retries in dequeuing threads.

All three algorithms (LCRQ, SPDQ, MPDQ) peak at 12 threads, where there is maximum parallelism without incurring chip-crossing overheads. The raw FAI test similarly scales well within a single chip. Under some conditions, the new dual queues may even outperform the single integer FAI test: depending on the state of the queue, 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.

While the blocking versions of the SPDQ and MPDQ both outperform their lock-free variants, the performance hit is asymmetric. The lock-free SPDQ is almost imperceptibly slower than the blocking version. We expect this happens because the window closing code is only run rarely, when the queue's polarity is negative and many threads are waiting. The MPDQ takes a drastic hit in order to close the window: every positive thread must verify its order with respect to any other concurrent positive threads, adding several cache misses to the critical path of the hot potato test.

Overall, our algorithms outperform existing dual queues by a factor of 4–6× and scale much more aggressively. We encourage their consideration for thread pools and other applications that depend on fast inter-thread communication. We believe the basic, "almost nonblocking" versions should suffice in almost any "real world" application. Based on these tests, we recommend using the MPDQ in any application in which dequeuing threads need to wait for actual data. 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.

6. REFERENCES

[1] 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.

[2] J. Izraelevitz and M. L. Scott. Fast dual ring queues. Technical Report 990, Computer Science Dept., Univ. of Rochester, Jan. 2014.

[3] 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.

[4] 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.

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

[6] 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.

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