

# Scalable Synchronous Queues

By William N. Scherer III, Doug Lea, and Michael L. Scott

## Abstract

In a thread-safe *concurrent queue*, consumers typically wait for producers to make data available. In a *synchronous queue*, producers similarly wait for consumers to take the data. We present two new nonblocking, contention-free synchronous queues that achieve high performance through a form of *dualism*: The underlying data structure may hold both data and, symmetrically, *requests*.

We present performance results on 16-processor SPARC and 4-processor Opteron machines. We compare our algorithms to commonly used alternatives from the literature and from the Java SE 5.0 class `java.util.concurrent.SynchronousQueue` both directly in synthetic microbenchmarks and indirectly as the core of Java's `ThreadPoolExecutor` mechanism. Our new algorithms consistently outperform the Java SE 5.0 `SynchronousQueue` by factors of three in unfair mode and 14 in fair mode; this translates to factors of two and ten for the `ThreadPoolExecutor`. Our synchronous queues have been adopted for inclusion in Java 6.

## 1. INTRODUCTION

Mechanisms to transfer data between threads are among the most fundamental building blocks of concurrent systems. Shared memory transfers are typically effected via a concurrent data structure that may be known variously as a *buffer*, a *channel*, or a *concurrent queue*. This structure serves to “pair up” producers and consumers. It can also serve to smooth out fluctuations in their relative rates of progress by buffering unconsumed data. This buffering, in systems that provide it, is naturally asymmetric: A consumer that tries to take data from an empty concurrent queue will wait for a producer to perform a matching `put` operation; however, a producer need not wait to perform a `put` unless space has run out. That is, producers can “run ahead” of consumers, but consumers cannot “run ahead” of producers.

A *synchronous queue* provides the “pairing up” function without the buffering; it is entirely symmetric: Producers and consumers wait for one another, “shake hands,” and leave in pairs. For decades, synchronous queues have played a prominent role in both the theory and practice of concurrent programming. They constitute the central synchronization primitive of Hoare's CSP<sup>8</sup> and of languages derived from it, and are closely related to the *rendezvous* of Ada. They are also widely used in message-passing software and in stream-style “hand-off” algorithms.<sup>2, Chap. 8</sup> (In this paper we focus on synchronous queues within a multithreaded program, not across address spaces or distributed nodes.)

Unfortunately, design-level tractability of synchronous queues has often come at the price of poor performance. “Textbook” algorithms for `put` and `take` may repeatedly suffer from *contention* (slowdown due to conflicts

with other threads for access to a cache line) and/or *blocking* (loops or scheduling operations that wait for activity in another thread). Listing 1, for example, shows one of the most commonly used implementations, due to Hanson.<sup>3</sup> It employs three separate *semaphores*, each of which is a potential source of contention and (in `acquire` operations) blocking.<sup>a</sup>

The synchronization burden of algorithms like Hanson's is especially significant on modern multicore and multiprocessor machines, where the OS scheduler may take thousands of cycles to block or unblock threads. Even an uncontended semaphore operation usually requires special read-modify-write or memory barrier (fence) instructions, each of which can take tens of cycles.<sup>b</sup>

**Listing 1: Hanson's synchronous queue. Semaphore `sync` indicates whether `item` is valid (initially, `no`); `send` holds 1 minus the number of pending `puts`; `recv` holds 0 minus the number of pending `takes`.**

```

00 public class HansonSQ<E> {
01     E item = null;
02     Semaphore sync = new Semaphore(0);
03     Semaphore send = new Semaphore(1);
04     Semaphore recv = new Semaphore(0);
05
06     public E take() {
07         recv.acquire();
08         E x = item;
09         sync.release();
10         send.release();
11         return x;
12     }
13
14     public void put(E x) {
15         send.acquire();
16         item = x;
17         recv.release();
18         sync.acquire();
19     }
20 }

```

<sup>a</sup> Semaphores are the original mechanism for scheduler-based synchronization (they date from the mid-1960s). Each semaphore contains a counter and a list of waiting threads. An `acquire` operation decrements the counter and then waits for it to be nonnegative. A `release` operation increments the counter and unblocks a waiting thread if the result is nonpositive. In effect, a semaphore functions as a non-synchronous concurrent queue in which the transferred data is null.

<sup>b</sup> Read-modify-write instructions (e.g., `compare_and_swap` [CAS]) facilitate constructing concurrent algorithms via atomic memory updates. Fences enforce ordering constraints on memory operations.

A previous version of this paper was published in *Proceedings of the 11th ACM Symposium on Principles and Practice of Parallel Programming*, Mar. 2006.

It is also difficult to extend Listing 1 and other “classic” synchronous queue algorithms to provide additional functionality. Many applications require `poll` and `offer` operations, which take an item only if a producer is already present, or `put` an item only if a consumer is already waiting (otherwise, these operations return an error). Similarly, many applications require the ability to time out if producers or consumers do not appear within a certain *patience* interval or if the waiting thread is asynchronously interrupted. In the `java.util.concurrent` library, one of the `ThreadPoolExecutor` implementations uses all of these capabilities: Producers deliver tasks to waiting worker threads if immediately available, but otherwise create new worker threads. Conversely, worker threads terminate themselves if no work appears within a given keep-alive period (or if the pool is shut down via an interrupt).

Additionally, applications using synchronous queues vary in their need for *fairness*: Given multiple waiting producers, it may or may not be important to an application whether the one waiting the longest (or shortest) will be the next to pair up with the next arriving consumer (and vice versa). Since these choices amount to application-level policy decisions, algorithms should minimize imposed constraints. For example, while fairness is often considered a virtue, a thread pool normally runs faster if the most-recently-used waiting worker thread usually receives incoming work, due to the footprint retained in the cache and the translation lookaside buffer.

In this paper we present synchronous queue algorithms that combine a rich programming interface with very low intrinsic overhead. Our algorithms avoid all blocking other than that intrinsic to the notion of synchronous handoff: A producer thread must wait until a consumer appears (and vice versa); there is no other way for one thread’s delay to impede another’s progress. We describe two algorithmic variants: a *fair* algorithm that ensures strict FIFO ordering and an *unfair* algorithm that makes no guarantees about ordering (but is actually based on a LIFO stack). Section 2 of this paper presents the background for our approach. Section 3 describes the algorithms and Section 4 presents empirical performance data. We conclude and discuss potential extensions to this work in Section 5.

## 2. BACKGROUND

### 2.1. Nonblocking synchronization

Concurrent data structures are commonly protected with *locks*, which enforce *mutual exclusion* on *critical sections* executed by different threads. A naive synchronous queue might be protected by a single lock, forcing all `put` and `take` operations to execute serially. (A thread that blocked waiting for a peer would of course release the lock, allowing the peer to execute the matching operation.) With a bit of care and a second lock, we might allow one producer and one consumer to execute concurrently in many cases.

Unfortunately, locks suffer from several serious problems. Among other things, they introduce blocking beyond that required by data structure semantics: If thread A holds a lock that thread B needs, then B must wait, even if A has been

preempted and will not run again for quite a while. A multi-programmed system with thread priorities or asynchronous events may suffer spurious deadlocks due to *priority inversion*: B needs the lock A holds, but A cannot run, because B is a handler or has higher priority.

Nonblocking concurrent objects address these problems by avoiding mutual exclusion. Loosely speaking, their methods ensure that the object’s invariants hold after every single instruction, and that its state can safely be seen—and manipulated—by other concurrent threads. Unsurprisingly, devising such methods can be a tricky business, and indeed the number of data structures for which correct nonblocking implementations are known is fairly small.

Linearizability<sup>7</sup> is the standard technique for demonstrating that a nonblocking implementation of an object is *correct* (i.e., that it continuously maintains object invariants). Informally, linearizability “provides the illusion that each operation... takes effect instantaneously at some point between its invocation and its response.”<sup>7, abstract</sup> Orthogonally, nonblocking implementations may provide guarantees of various strength regarding the *progress* of method calls. In a *wait-free* implementation, every contending thread is guaranteed to complete its method call within a bounded number of its own execution steps.<sup>5</sup> Wait-free algorithms tend to have unacceptably high overheads in practice, due to the need to finish operations on other threads’ behalf. In a *lock-free* implementation, *some* contending thread is guaranteed to complete its method call within a bounded number of any thread’s steps.<sup>5</sup> The algorithms we present in this paper are all lock-free. Some algorithms provide a weaker guarantee known as *obstruction freedom*; it ensures that a thread can complete its method call within a bounded number of steps in the absence of contention, i.e., if no other threads execute competing methods concurrently.<sup>6</sup>

### 2.2. Dual data structures

In traditional nonblocking implementations of concurrent objects, every method is *total*: It has no preconditions that must be satisfied before it can complete. Operations that might normally block before completing, such as dequeuing from an empty queue, are generally *totalized* to simply return a failure code when their preconditions are not met. By calling the totalized method in a loop until it succeeds, one can simulate the partial operation. This simulation, however, does not necessarily respect our intuition for object semantics. For example, consider the following sequence of events for threads A, B, C, and D:

```
A calls dequeue
B calls dequeue
C enqueues a 1
D enqueues a 2
B’s call returns the 1
A’s call returns the 2
```

If thread A’s call to dequeue is known to have started before thread B’s call, then intuitively, we would think that A should get the first result out of the queue. Yet, with the call-in-a-loop idiom, ordering is simply a function of which

thread happens to retry its dequeue operation first once data becomes available. Further, each invocation of the totalized method introduces performance-degrading contention for memory–interconnect bandwidth.

As an alternative, suppose we could register a request for a hand-off partner. Inserting this *reservation* could be done in a nonblocking manner, and checking to see whether a partner has arrived to *fulfill* our reservation could consist of reading a Boolean flag in the request data structure. A *dual data structure*<sup>16, 19</sup> takes precisely this approach: Objects may contain both data and reservations. We divide partial methods into separate, first-class *request* and *follow-up* operations, each of which has its own invocation and response. A total queue, for example, would provide `dequeue_request` and `dequeue_followup` methods (Listing 2). By analogy with Lamport’s bakery algorithm,<sup>10</sup> the request operation returns a unique *ticket* that represents the reservation and is then passed as an argument to the follow-up method. The follow-up, for its part, returns either the desired result (if one is *matched* to the ticket) or, if the method’s precondition has not yet been satisfied, an error indication.

The key difference between a dual data structure and a “totalized” partial method is that linearization of the `p_request` call allows the dual data structure to determine the fulfillment order for pending requests. In addition, unsuccessful follow-ups, unlike unsuccessful calls to totalized methods, are readily designed to avoid bus or memory contention. For programmer convenience, we provide demand methods, which wait until they can return successfully. Our implementations use both busy-wait spinning and scheduler-based suspension to effect waiting in threads whose preconditions are not met.

When reasoning about progress, we must deal with the fact that a partial method may wait for an arbitrary amount of time (perform an arbitrary number of unsuccessful follow-ups) before its precondition is satisfied. Clearly it is desirable that requests and follow-ups be nonblocking. In practice, good system performance will also typically require that unsuccessful follow-ups not interfere with other threads’ progress. We define a data structure as *contention-free* if none of its follow-up operations, in any execution, performs more than a constant number of remote memory accesses across all unsuccessful invocations with the same request ticket. On a machine with an invalidation-based cache coherence protocol, a read of

**Listing 2: Combined operations: dequeue pseudocode (enqueue is symmetric).**

```
datum dequeue(SynchronousQueue Q) {
  reservation r = Q.dequeue_reserve();
  do {
    datum d = Q.dequeue_followup(r);
    if (failed != d) return d;
    /* else delay -- spinning and/or scheduler-based */
    while (!timed_out());
    if (Q.dequeue_abort(r)) return failed;
    return Q.dequeue_followup(r);
  }
}
```

location *o* by thread *t* is said to be *remote* if *o* has been written by some thread other than *t* since *t* last accessed it; a write by *t* is remote if *o* has been accessed by some thread other than *t* since *t* last wrote it. On a machine that cannot cache remote locations, an access is remote if it refers to memory allocated on another node. Compared to the *local-spin property*,<sup>13</sup> contention freedom allows operations to block in ways other than busy-wait spinning; in particular, it allows other actions to be performed while waiting for a request to be satisfied.

### 3. ALGORITHM DESCRIPTIONS

In this section we discuss various implementations of synchronous queues. We start with classic algorithms used extensively in production software, then we review newer implementations that improve upon them. Finally, we describe our new algorithms.

#### 3.1. Classic synchronous queues

Perhaps the simplest implementation of synchronous queues is the naive monitor-based algorithm that appears in Listing 3. In this implementation, a single monitor serializes access to a single item and to a `putting` flag that indicates whether a producer has currently supplied data. Producers wait for the flag to be clear (lines 15–16), set the flag (17), insert an item (18), and then wait until a consumer takes the data (20–21). Consumers await the presence of an item (05–06), take it (07), and mark it as taken (08) before returning. At each point where their actions might potentially unblock another thread, producer and consumer threads awaken all possible candidates (09, 20, 24). Unfortunately, this approach results in a number of wake-ups quadratic in the number of waiting producer and consumer threads; coupled with the high cost of blocking or

**Listing 3: Naive synchronous queue.**

```
00 public class NaiveSQ<E> {
01   boolean putting = false;
02   E item = null;
03
04   public synchronized E take() {
05     while (item == null)
06       wait();
07     E e = item;
08     item = null;
09     notifyAll();
10     return e;
11   }
12
13   public synchronized void put (E e) {
14     if (e == null) return;
15     while (putting)
16       wait();
17     putting = true;
18     item = e;
19     notifyAll();
20     while (item != null)
21       wait();
22     putting = false;
23     notifyAll();
24   }
25 }
```

unblocking a thread, this results in poor performance.

Hanson's synchronous queue (Listing 1) improves upon the naive approach by using semaphores to target wake-ups to only the single producer or consumer thread that an operation has unblocked. However, as noted in Section 1, it still incurs the overhead of three separate synchronization events per transfer for each of the producer and consumer; further, it normally blocks at least once per operation. It is possible to streamline some of these synchronization points in common execution scenarios by using a fast-path acquire sequence;<sup>11</sup> this was done in early releases of the *dl.util.concurrent* package which evolved into *java.util.concurrent*.

### 3.2. The Java SE 5.0 synchronous queue

The Java SE 5.0 synchronous queue (Listing 4) uses a pair of queues (in fair mode; stacks for unfair mode) to separately hold waiting producers and consumers. This approach echoes the scheduler data structures of Anderson et al;<sup>1</sup> it improves considerably on semaphore-based approaches. When a producer or consumer finds its counterpart already waiting, the new arrival needs to perform only one synchronization operation: acquiring a lock that protects both queues (line 18 or 33). Even if no counterpart is waiting, the only additional synchronization required is to await one (25 or 40). A transfer thus requires only three synchronization operations, compared to the six incurred by Hanson's algorithm. In particular, using a queue instead of a semaphore allows producers to publish data items as they arrive (line 36) instead of having to first awaken after blocking on a semaphore; consumers need not wait.

### 3.3. Combining dual data structures with synchronous queues

A key limitation of the Java SE 5.0 *SynchronousQueue* class is its reliance on a single lock to protect both queues. Coarse-grained synchronization of this form is well known for introducing serialization bottlenecks; by creating nonblocking implementations, we eliminate a major impediment to scalability.

Our new algorithms add support for time-out and for bidirectional synchronous waiting to our previous nonblocking dual queue and dual stack algorithms<sup>19</sup> (those in turn were derived from the classic Treiber stack<sup>21</sup> and the M&S queue<sup>14</sup>). The nonsynchronous dual data structures already block when a consumer arrives before a producer; our challenge is to arrange for producers to block until a consumer arrives as well. In the queue, waiting is accomplished by spinning until a pointer changes from null to non-null, or vice versa; in the stack, it is accomplished by pushing a "fulfilling" node and arranging for adjacent matching nodes to "annihilate" one another.

We describe basic versions of the synchronous dual queue and stack in the sections "The synchronous dual queue" and "The synchronous dual stack," respectively. The section "Time-out" then sketches the manner in which we add time-out support. The section "Pragmatics" discusses additional pragmatic issues. Throughout the discussion, we present fragments of code to illustrate particular features; full source is available online at <http://gee.cs.oswego.edu/cgi-bin/viewcvs.cgi/jsr166/src/main/java/util/concurrent/SynchronousQueue.java>.

**Listing 4: The Java SE 5.0 *SynchronousQueue* class, fair (queue-based) version. The unfair version uses stacks instead of queues, but is otherwise identical. (For clarity, we have omitted details of the way in which *AbstractQueuedSynchronizers* are used, and code to generalize *waitingProducers* and *waitingConsumers* to either stacks or queues.)**

```
00 public class Java5SQ<E> {
01     ReentrantLock qlock = new ReentrantLock();
02     Queue waitingProducers = new Queue();
03     Queue waitingConsumers = new Queue();
04
05     static class Node
06         extends AbstractQueuedSynchronizer {
07         E item;
08         Node next;
09
10         Node(Object x) { item = x; }
11         void waitForTake() { /* (uses AQS) */ }
12         E waitForPut() { /* (uses AQS) */ }
13     }
14
15     public E take() {
16         Node node;
17         boolean mustWait;
18         qlock.lock();
19         node = waitingProducers.pop();
20         if (mustWait = (node == null))
21             node = waitingConsumers.push(null);
22         qlock.unlock();
23
24         if (mustWait)
25             return node.waitForPut();
26         else
27             return node.item;
28     }
29
30     public void put(E e) {
31         Node node;
32         boolean mustWait;
33         qlock.lock();
34         node = waitingConsumers.pop();
35         if (mustWait = (node == null))
36             node = waitingProducers.push(e);
37         qlock.unlock();
38
39         if (mustWait)
40             node.waitForTake();
41         else
42             node.item = e;
43     }
44 }
```

**The Synchronous Dual Queue:** We represent the synchronous dual queue as a singly linked list with head and tail pointers. The list may contain *data nodes* or *request nodes* (reservations), but never both at once. Listing 5 shows the enqueue method. (Except for the direction of data transfer, dequeue is symmetric.) To enqueue, we first read the head and tail pointers (lines 06–07). From here, there are two main cases. The first occurs when the queue is empty ( $h == t$ ) or contains data (line 08). We read the next pointer for the tail-most node in the queue (09). If all values read are mutually consistent (10) and the queue's tail pointer is current (11), we attempt to insert our offering at the tail of the queue (13–14). If successful, we wait until a consumer signals that it has

**Listing 5: Synchronous dual queue: Spin-based enqueue;** dequeue is symmetric except for the direction of data transfer. The various `cas field (old, new)` operations attempt to change `field` from `old` to `new`, and return a success/failure indication. On modern processors they can be implemented with a single atomic `compare_and_swap` instruction, or its equivalent.

```

00 class Node { E data; Node next;...}
01
02 void enqueue(E e) {
03     Node offer = new Node(e, Data);
04
05     while (true) {
06         Node t = tail;
07         Node h = head;
08         if (h == t || !t.isRequest()) {
09             Node n = t.next;
10             if (t == tail) {
11                 if (null != n) {
12                     casTail(t, n);
13                 } else if (t.casNext(n, offer)) {
14                     casTail(t, offer);
15                     while (offer.data == e)
16                         /* spin */;
17                     h = head;
18                     if (offer == h.next)
19                         casHead(h, offer);
20                     return;
21                 }
22             }
23         } else {
24             Node n = h.next;
25             if (t != tail || h != head || n == null)
26                 continue; // inconsistent snapshot
27             boolean success = n.casData(null, e);
28             casHead(h, n);
29             if (success)
30                 return;
31         }
32     }
33 }

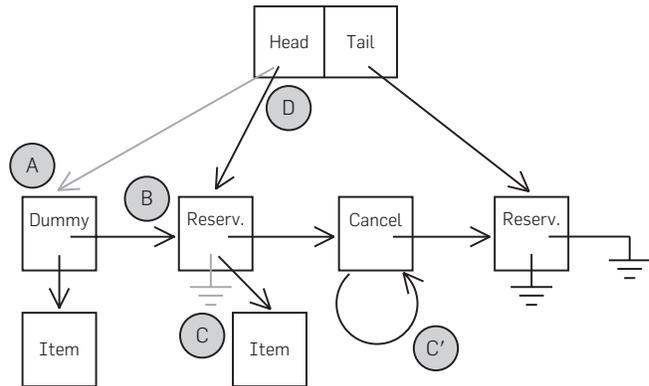
```

claimed our data (15–16), which it does by updating our node’s data pointer to null. Then we help remove our node from the head of the queue and return (18–20). The request linearizes in this code path at line 13 when we successfully insert our offering into the queue; a successful follow-up linearizes when we notice at line 15 that our data has been taken.

The other case occurs when the queue consists of reservations, and is depicted in Figure 1. After originally reading the head node (step A), we read its successor (line 24/step B) and verify consistency (25). Then, we attempt to supply our data to the headmost reservation (27/C). If this succeeds, we dequeue the former dummy node (28/D) and return (30). If it fails, we need to go to the next reservation, so we dequeue the old dummy node anyway (28) and retry the entire operation (32, 05). The request linearizes in this code path when we successfully supply data to a waiting consumer at line 27; the follow-up linearization point occurs immediately thereafter.

**The Synchronous Dual Stack:** We represent the synchronous dual stack as a singly linked list with head pointer. Like the dual queue, the stack may contain either data or

**Figure 1: Synchronous dual queue: Enqueuing when reservations are present.**



**Listing 6: Synchronous dual stack: Spin-based annihilating push;** pop is symmetric except for the direction of data transfer. (For clarity, code for time-out is omitted.)

```

00 class Node { E data; Node next, match; ... }
01
02 void push (E e) {
03     Node f, d = new Node(e, Data);
04
05     while (true) {
06         Node h = head;
07         if (null == h || h.isData()) {
08             d.next = h;
09             if (!casHead(h, d))
10                 continue;
11             while (d.match == null)
12                 /* spin */;
13             h = head;
14             if (null != h && d == h.next)
15                 casHead(h, d.next);
16             return;
17         } else if (h.isRequest()) {
18             f = new Node(e, Data | Fulfilling, h);
19             if (!casHead(h, f))
20                 continue;
21             h = f.next;
22             Node n = h.next;
23             h.casMatch(null, f);
24             casHead(f, n);
25             return;
26         } else { // h is fulfilling
27             Node n = h.next;
28             Node nn = n.next;
29             n.casMatch(null, h);
30             casHead(h, nn);
31         }
32     }
33 }

```

reservations, except that in this case there may, temporarily, be a single node of the opposite type at the head.

Code for the push operation appears in Listing 6. (Except for the direction of data transfer, `pop` is symmetric.) We begin by reading the node at the top of the stack (line 06).

The three main conditional branches (beginning at lines 07, 17, and 26) correspond to the type of node we find.

The first case occurs when the stack is empty or contains only data (line 07). We attempt to insert a new datum (09), and wait for a consumer to claim that datum (11–12) before returning. The reservation linearizes in this code path when we push our datum at line 09; a successful follow-up linearizes when we notice that our data has been taken at line 11.

The second case occurs when the stack contains (only) reservations (17). We attempt to place a fulfilling datum on the top of the stack (19); if we succeed, any other thread that wishes to perform an operation must now help us fulfill the request before proceeding to its own work. We then read our way down the stack to find the successor node to the reservation we are fulfilling (21–22) and mark the reservation fulfilled (23). Note that our CAS could fail if another thread helps us and performs it first. Finally, we pop both the reservation and our fulfilling node from the stack (24) and return. The reservation linearizes in this code path at line 19, when we push our fulfilling datum above a reservation; the follow-up linearization point occurs immediately thereafter.

The remaining case occurs when we find another thread’s fulfilling datum or reservation (26) at the top of the stack. We must complete the pairing and annihilation of the top two stack nodes before we can continue our own work. We first read our way down the stack to find the data or reservation for which the fulfilling node is present (27–28) and then we mark the underlying node as fulfilled (29) and pop the paired nodes from the stack (30).

Referring to Figure 2, when a consumer wishes to retrieve data from an empty stack, it first must insert a reservation (step A). It then waits until its data pointer (branching to the right) is non-null. Meanwhile, if a producer appears, it satisfies the consumer in a two-step process. First (step B), it pushes a fulfilling data node at the top of the stack. Then, it swings the reservation’s data pointer to its fulfilling node (step C). Finally, it updates the top-of-stack pointer to match the reservation node’s next pointer (step D, not shown). After the producer has completed step B, other threads can help update the reservation’s data pointer (step C); and the consumer thread can additionally help remove itself from the stack (step D).

**Time-Out:** Although the algorithms presented in the sections “The Synchronous Dual Queue” and “The

Synchronous Dual Stack” are complete implementations of synchronous queues, real systems require the ability to specify limited patience so that a producer (or consumer) can time out if no consumer (producer) arrives soon enough to pair up. As noted earlier, Hanson’s synchronous queue offers no simple way to do this. Space limitations preclude discussion of the relatively straightforward manner in which we add time-out support to our synchronous queue; interested readers may find this information in our original publication.<sup>17</sup>

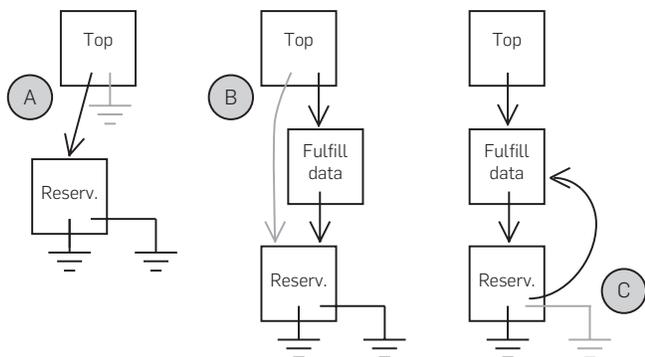
**Pragmatics:** Our synchronous queue implementations reflect a few additional pragmatic considerations to maintain good performance. First, because Java does not allow us to set flag bits in pointers (to distinguish among the types of pointed-to nodes), we add an extra word to nodes, in which we mark mode bits. We chose this technique over two primary alternatives. The class *java.util.concurrent.AtomicMarkableReference* allows direct association of tag bits with a pointer, but exhibits very poor performance. Using runtime type identification (RTTI) to distinguish between multiple subclasses of the Node classes would similarly allow us to embed tag bits in the object type information. While this approach performs well in isolation, it increases long-term pressure on the JVM’s memory allocation and garbage collection routines by requiring construction of a new node after each contention failure.

Time-out support requires careful management of memory ownership to ensure that canceled nodes are reclaimed properly. Automatic garbage collection eases the burden in Java. We must, however, take care to “forget” references to data, nodes, and threads that might be retained for a long time by blocked threads (preventing the garbage collector from reclaiming them).

The simplest approach to time-out involves marking nodes as “canceled,” and abandoning them for another thread to eventually unlink and reclaim. If, however, items are offered at a very high rate, but with a very low time-out patience, this “abandonment” cleaning strategy can result in a long-term build-up of canceled nodes, exhausting memory supplies and degrading performance. It is important to effect a more sophisticated cleaning strategy. Space limitations preclude further discussion here, but interested readers may find more details in the conference version of this paper.<sup>17</sup>

For sake of clarity, the synchronous queues of Figures 5 and 6 blocked with busy-wait spinning to await a counterpart consumer. In practice, however, busy-wait is useless overhead on a uniprocessor and can be of limited value on even a small-scale multiprocessor. Alternatives include descheduling a thread until it is signaled, or yielding the processor within a spin loop.<sup>9</sup> In practice, we mainly choose the spin-then-yield approach, using the *park* and *unpark* methods contained in *java.util.concurrent.locks.LockSupport*<sup>12</sup> to remove threads from and restore threads to the ready list. On multiprocessors (only), nodes next in line for fulfillment spin briefly (about one-quarter the time of a typical context switch) before parking. On very busy synchronous queues, spinning can dramatically improve throughput because it handles the case of a near-simultaneous “flyby” between a producer and consumer without stalling either. On less busy

**Figure 2: Synchronous dual stack: Satisfying a reservation.**



queues, the amount of spinning is small enough not to be noticeable.

#### 4. EXPERIMENTAL RESULTS

We present results for several microbenchmarks and one “real-world” scenario. The microbenchmarks employ threads that produce and consume as fast as they can; this represents the limiting case of producer-consumer applications as the cost to process elements approaches zero. We consider producer-consumer ratios of  $1 : N$ ,  $N : 1$ , and  $N : N$ .

Our “real-world” scenario instantiates synchronous queues as the core of the Java SE 5.0 class `java.util.concurrent.ThreadPoolExecutor`, which in turn forms the backbone of many Java-based server applications. Our benchmark produces *tasks* to be run by a pool of worker threads managed by the `ThreadPoolExecutor`.

We obtained results on a SunFire V40z with four 2.4GHz AMD Opteron processors and on a SunFire 6800 with 16 1.3GHz Ultra-SPARC III processors. On both machines, we used Sun’s Java SE 5.0 HotSpot VM and we varied the level of concurrency from 2 to 64. We tested each benchmark with both the fair and unfair (stack-based) versions of the Java SE 5.0 `java.util.concurrent.SynchronousQueue`, Hanson’s synchronous queue, and our new nonblocking algorithms.

Figure 3 displays the rate at which data is transferred from multiple producers to multiple consumers; Figure 4 displays the rate at which data is transferred from a single producer to multiple consumers; Figure 5 displays the rate at which a single consumer receives data from multiple producers. Figure 6 presents execution time per task for our `ThreadPoolExecutor` benchmark.

As can be seen from Figure 3, Hanson’s synchronous queue and the Java SE 5.0 fair-mode synchronous queue both perform relatively poorly, taking 4 (Opteron) to 8 (SPARC) times as long to effect a transfer relative to the faster algorithms. The unfair (stack-based) Java SE 5.0 synchronous queue in turn incurs twice the overhead of either the fair or unfair version of our new algorithm, both versions of which are comparable in performance. The main reason that the Java SE 5.0 fair-mode queue is so much slower than unfair is that the fair-mode version uses a fair-mode entry lock to ensure FIFO wait ordering. This causes pileups that block the threads that will fulfill waiting threads. This difference supports our claim that blocking and contention surrounding the synchronization state of synchronous queues are major impediments to scalability.

When a single producer struggles to satisfy multiple consumers (Figure 4), or a single consumer struggles to receive data from multiple producers (Figure 5), the disadvantages

Figure 3: Synchronous handoff:  $N$  producers,  $N$  consumers.

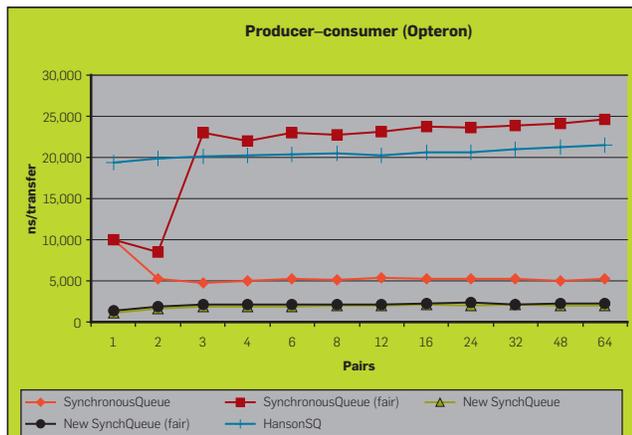
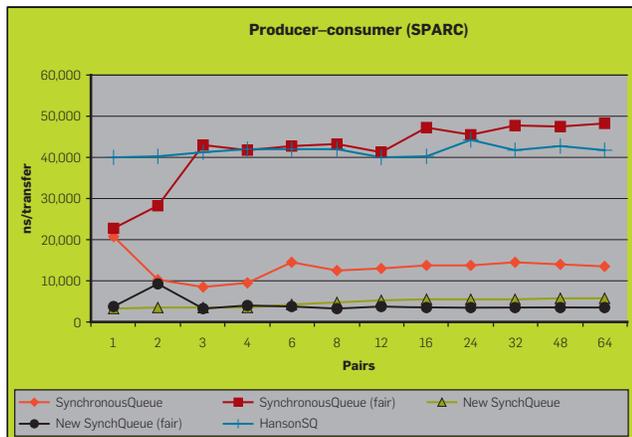


Figure 4: Synchronous handoff: 1 producer,  $N$  consumers.

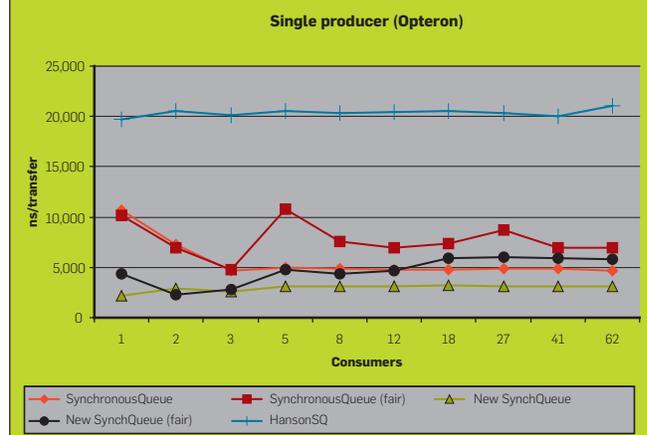
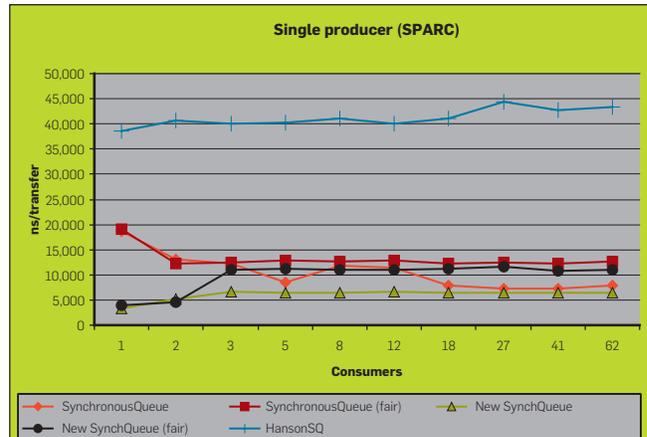


Figure 5: Synchronous handoff:  $N$  producers, 1 consumer.

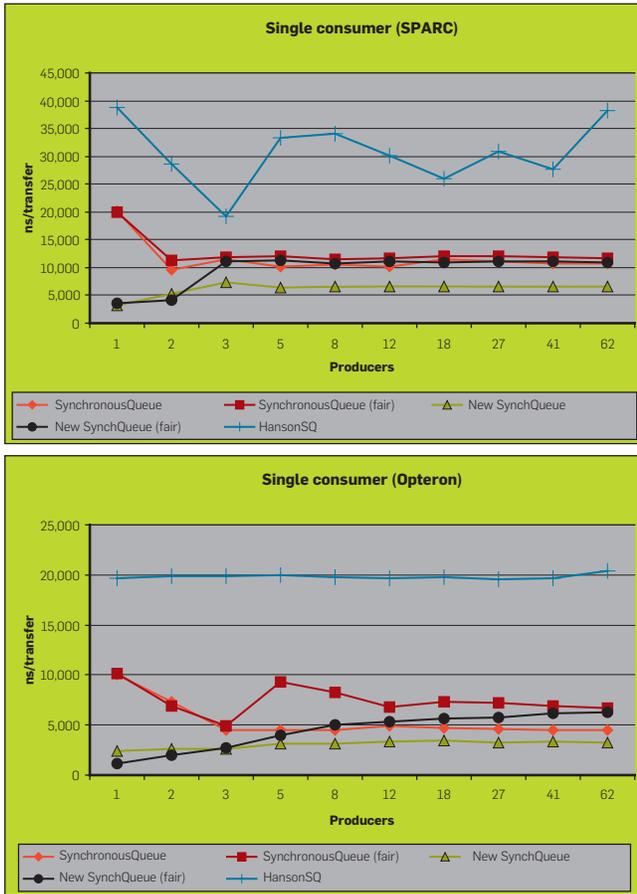
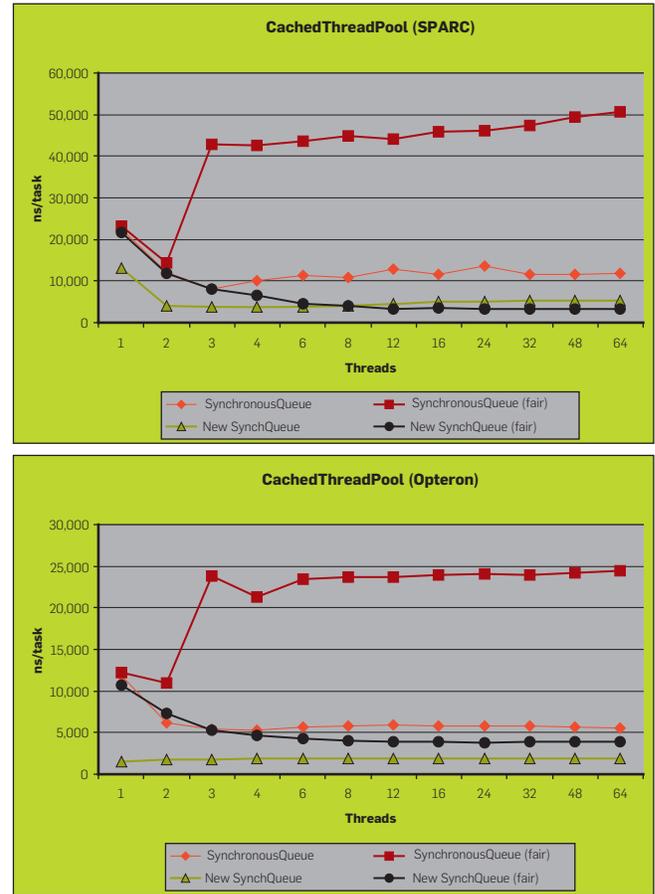


Figure 6: ThreadPoolExecutor benchmark.



of Hanson's synchronous queue are accentuated. Because the singleton necessarily blocks for every operation, the time it takes to produce or consume data increases noticeably. Our new synchronous queue consistently outperforms the Java SE 5.0 implementation (fair vs. fair and unfair vs. unfair) at all levels of concurrency.

Finally, in Figure 6, we see that the performance differentials from *java.util.concurrent*'s *SynchronousQueue* translate directly into overhead in the *ThreadPoolExecutor*: Our new fair version outperforms the Java SE 5.0 implementation by factors of 14 (SPARC) and 6 (Opteron); our unfair version outperforms Java SE 5.0 by a factor of three on both platforms. Interestingly, the relative performance of fair and unfair versions of our new algorithm differs between the two platforms. Generally, unfair mode tends to improve locality by keeping some threads "hot" and others buried at the bottom of the stack. Conversely, however, it tends to increase the number of times threads are scheduled and descheduled. On the SPARC, context switches have a higher relative overhead compared to other factors; this is why our fair synchronous queue eventually catches and surpasses the unfair version's performance. In contrast, the cost of context switches is relatively smaller on the Opteron, so the trade-off tips in favor of increased locality and the unfair version performs best.

Across all benchmarks, our fair synchronous queue universally outperforms all other fair synchronous queues and our unfair synchronous queue outperforms all other unfair synchronous queues, regardless of preemption or level of concurrency.

## 5. CONCLUSION

In this paper, we have presented two new lock-free and contention-free synchronous queues that outperform all previously known algorithms by a wide margin. In striking contrast to previous implementations, there is little performance cost for fairness.

In a head-to-head comparison, our algorithms consistently outperform the Java SE 5.0 *SynchronousQueue* by a factor of three in unfair mode and up to a factor of 14 in fair mode. We have further shown that this performance differential translates directly to factors of two and ten when substituting our new synchronous queue in for the core of the Java SE 5.0 *ThreadPoolExecutor*, which is itself at the heart of many Java-based server implementations. Our new synchronous queues have been adopted for inclusion in Java 6.

More recently, we have extended the approach described in this paper to *TransferQueues*. *TransferQueues* permit producers to enqueue data either synchronously or

asynchronously. TransferQueues are useful for example in supporting messaging frameworks that allow messages to be either synchronous or asynchronous. The base synchronous support in TransferQueues mirrors our fair synchronous queue. The asynchronous additions differ only by releasing producers before items are taken.

Although we have improved the scalability of the synchronous queue, there may remain potential for improvement in some contexts. Most of the inter-thread contention in enqueue and dequeue operations occurs at the memory containing the head (and, for fair queues, tail). Reducing such contention by spreading it out is the idea behind *elimination* techniques introduced by Shavit and Touitou.<sup>20</sup> These may be applied to components featuring pairs of operations that collectively effect no change to a data structure, for example, a concurrent push and pop on a stack. Using elimination, multiple locations (comprising an *arena*) are employed as potential targets of the main atomic instructions underlying these operations. If two threads meet in one of these lower-traffic areas, they cancel each other out. Otherwise, the threads must eventually fall back (usually, in a tree-like fashion) to try the main location.

Elimination techniques have been used by Hendler et al.<sup>4</sup> to improve the scalability of stacks, and by us<sup>18</sup> to improve the scalability of the swapping channels in the *java.util.concurrent.Exchanger* class. Moir et al.<sup>15</sup> have also used elimination in concurrent queues, although at the price of weaker ordering semantics than desired in some applications due to stack-like (LIFO) operation of the elimination arena. Similar ideas could be applied to our synchronous queues. However, to be worthwhile here, the reduced contention benefits would need to outweigh the delayed release (lower throughput) experienced when threads do not meet in arena locations. In preliminary work, we have found elimination to be beneficial only in cases of artificially extreme contention. We leave fuller exploration to future work.

### Acknowledgments

We are grateful to Dave Dice, Brian Goetz, David Holmes, Mark Moir, Bill Pugh, and the PPOPP referees for feedback that significantly improved the presentation of this paper. This work was supported in part by NSF grants numbers EIA-0080124, CCR-0204344, and CNS-0411127, and by financial and equipment grants from Sun Microsystems Laboratories. 

### References

- Anderson, T.E., Lazowska, E.D., Levy, H.M. The performance implications of thread management alternatives for shared-memory multiprocessors. *IEEE Trans. Comput.* 38, 12 (Dec. 1989), 1631–1644.
- Andrews, G.R. *Concurrent Programming: Principles and Practice*. Benjamin/Cummings, 1991.
- Hanson, D.R. *C Interfaces and Implementations: Techniques for Creating Reusable Software*. Addison-Wesley, 1997.
- Hendler, D., Shavit, N., Yerushalmi, L. A scalable lock-free stack algorithm. In *Proceedings of the 16th Annual ACM Symposium on Parallelism in Algorithms and Architectures* (Jun. 2004), 206–215.
- Herlihy, M. Wait-free synchronization. *ACM Trans. Prog. Lang. Syst.* 13, 1 (Jan. 1991), 124–149.
- Herlihy, M., Luchangco, V., Moir, M. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems* (May 2003).
- Herlihy, M.P., Wing, J.M. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Prog. Lang. Syst.* 12, 3 (Jul. 1990), 463–492.
- Hoare, C.A.R. Communicating sequential processes. *Commun. ACM* 21, 8 (Aug. 1978), 666–677.
- Karlin, A.R., Li, K., Manasse, M.S., Owicki, S. Empirical studies of competitive spinning for a shared-memory multiprocessor. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles* (Oct. 1991), 41–55.
- Lampert, L. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM* 17, 8 (Aug. 1974), 453–455.
- Lampert, L. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst.* 5, 1 (Feb. 1987), 1–11.
- Lea, D. The java.util.concurrent Synchronizer Framework. *Sci. Comput. Prog.* 58, 3 (Dec. 2005), 293–309.
- Mellor-Crummey, J.M., Scott, M.L. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.* 9, 1 (Feb. 1991), 21–65.
- Michael, M.M., Scott, M.L. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing* (May 1996), 267–275.
- Moir, M., Nussbaum, D., Shalev, O., Shavit, N. Using elimination to implement scalable and lock-free FIFO queues. In *Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures* (Jul. 2005), 253–262.
- Scherer III, W.N. Synchronization and concurrency in user-level software systems. Ph.D. dissertation, Department of Computer Science, University of Rochester (Jan. 2006).
- Scherer III, W.N., Lea, D., Scott, M.L. Scalable synchronous queues. In *Proceedings of the 11th ACM Symposium on Principles and Practice of Parallel Programming* (Mar. 2006).
- Scherer III, W.N., Lea, D., Scott, M.L. A scalable elimination-based exchange channel. In *Proceedings, Workshop on Synchronization and Concurrency in Object-Oriented Languages* (Oct. 2005). In conjunction with OOPSLA '05.
- Scherer III, W.N., Scott, M.L. Nonblocking concurrent objects with condition synchronization. In *Proceedings of the 18th International Symposium on Distributed Computing* (Oct. 2004).
- Shavit, N., Touitou, D. Elimination trees and the construction of pools and stacks. In *Proceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures* (Jul. 1995).
- Treiber, R.K. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, Apr. 1986.

**William N. Scherer III** (scherer@edu)  
Department of Computer Science  
Rice University, Houston, TX.

**Doug Lea** (dl@cs.oswego.edu)  
Department of Computer Science  
SUNY Oswego, Oswego, NY.

**Michael L. Scott** (scott@cs.rochester.edu)  
Department of Computer Science  
University of Rochester, Rochester, NY.

© 2009 ACM 0001-0782/09/0500 \$5.00