Scalable Synchronous Queues *

William N. Scherer III
University of Rochester
scherer@cs.rochester.edu

Doug Lea
SUNY Oswego
dl@cs.oswego.edu

Michael L. Scott
University of Rochester
scott@cs.rochester.edu

Abstract

We present two new nonblocking and contention-free implementations of *synchronous queues*, concurrent transfer channels in which producers wait for consumers just as consumers wait for producers. Our implementations extend our previous work in dual queues and dual stacks to effect very high-performance handoff.

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 (which in turn is the core of many Java server programs). 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.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming

General Terms algorithms, performance, experimentation

Keywords nonblocking synchronization, dual data structures, synchronous queue, dual stack, dual queue, lock freedom, contention freedom

1. Scalable Synchronous Queues

A synchronous queue (perhaps better known as a "synchronous channel") is one in which each producer presenting an item (via a `put` operation) must wait for a consumer to `take` this item, and vice versa. 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 [8] 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 hand-off designs [2].

Unfortunately, the design-level tractability of synchronous queues has often come at the price of poor performance. "Textbook" algorithms for implementing synchronous queues contain a series of potential contention or blocking points in every `put` or `take`. (We consider in this paper only those synchronous queues operating within a single multithreaded program; not across multiple processes or distributed nodes.) For example, Listing 1 shows one of the most commonly used implementations, due to Hanson [3], which uses three separate semaphores.

Such heavy synchronization burdens are especially significant on contemporary multiprocessors and their operating systems, in which the blocking and unblocking of threads tend to be very expensive operations. Moreover, even a series of uncontended semaphore operations usually requires enough costly atomic and barrier (fence) instructions to incur substantial overhead.

It is also difficult to extend this and other "classic" synchronous queue algorithms to support other common operations. These include `poll`, which takes an item only if a producer is already present, and `offer` which fails unless a consumer is waiting. 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. One of the *java.util.concurrent.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 and implementations

```
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 }
```

Listing 1. Hanson's synchronous queue.

* 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.

PPoPP'06 March 29–31, 2006, New York, New York, USA.

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 footprint retained in the cache and VM system.

In this paper we present synchronous queue algorithms that combine a rich programming interface with very low intrinsic overhead. More specifically, our algorithms avoid all blocking other than that which is intrinsic to the notion of synchronous handoff: A producer thread must wait until a consumer appears (and vice-versa), but 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 in Section 5 and discuss potential extensions to this work.

2. Background

2.1 Nonblocking Synchronization

In contrast to lock-based implementations of concurrent data structures, *nonblocking* implementations never require a call to a total method (one whose precondition is simply `true`) to wait for the execution of any other method. Linearizability [6] is the standard technique for demonstrating that a nonblocking implementation of an object is *correct* (i.e., that it maintains object invariants). Informally, linearizability "provides the illusion that each operation… takes effect instantaneously at some point between its invocation and its response" [6, abstract]. 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 [7]. In a *lock-free* implementation, *some* contending thread is guaranteed to complete its method call within a bounded number of steps (from any thread's point of view) [7]. In an *obstruction-free* implementation, a thread is guaranteed to 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 [5].

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, doesn't 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 try a totalized 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 handoff partner. Inserting this *reservation* could be done in a non-blocking 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. In our treatment of *dual data structures* [17, 19], 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. By analogy with Lamport's bakery algorithm [10], 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.

Given standard definitions of well-formedness, a thread t that wishes to execute a partial method p must first call p`_request` and then call p`_followup` in a loop until it succeeds. This is very different from calling a traditional "totalized" method until it succeeds: Linearization of distinguished request operations is the hook that allows object semantics to address the order in which pending requests will be fulfilled. Moreover unsuccessful follow-ups, unlike unsuccessful calls to totalized methods, can easily be designed to avoid bus or memory contention.

For programmer convenience, we provide `demand` methods that 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 (potentially performing 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 progress in other threads. We define a concurrent data structure as *contention-free* if none of its follow-up operations performs more than a constant number of remote memory accesses across all unsuccessful invocations with the same request ticket. On a cache-coherent machine that can cache remote memory locations, an access by thread t within operation o is said to be *remote* if it writes to memory that may (in some execution) be read or written by threads other than t more than a constant number of times in the interval between o's invocation and return, or if it reads memory that may (in some execution) be written by threads other than t more than a constant number of times in that interval. On a non-cache-coherent machine, an access by thread t is also remote if it refers to memory that t itself did not allocate. Compared to the *local-spin property* [14], 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.

2.3 Synchronous Queue Semantics

We model a fair synchronous queue Q as a tuple $(D, E, \hat{r}, \hat{s}, M)$, where $D = \{r_{d_0}, \ldots, r_{d_i}\}$ is a set of unmatched dequeue requests; $E = \{(v_{e_0}, s_{e_0}), \ldots, (v_{e_j}, s_{e_j})\}$ is a set of unmatched enqueue requests; $\hat{r} \in \mathcal{N}$ indicates the number of `dequeue_reserve` operations that have completed; $\hat{s} \in \mathcal{N}$ indicates the number of `enqueue_reserve` operations that have completed; and $M = \{(r_{m_0}, v_{n_0}, s_{n_0}), \ldots, (r_{m_k}, v_{n_k}, s_{n_k})\}$ is a set of matched enqueue/dequeue pairs. Initially $Q = (\varnothing, \varnothing, 1, 1, \varnothing)$.

The operations `enqueue_reserve`, `enqueue_followup`, `enqueue_abort`, `dequeue_reserve`, `dequeue_followup`, and

```
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);
}
```

Listing 2. Combined operations: `dequeue` pseudocode. `enqueue` is symmetric.

`dequeue_abort`, with arguments as shown below, induce the following state transitions on Q, with the indicated return values. These transitions ensure that at all times either $D = \varnothing$ or $E = \varnothing$.

- `enqueue_reserve(`$\dot{v}$`)`: If $D = \varnothing$, changes Q to be $(\varnothing, E \cup \{(\dot{v}, \hat{s})\}, \hat{r}, \hat{s} + 1, M)$. Otherwise, changes Q to be $(D \setminus \{r_{d_t}\}, \varnothing, \hat{r}, \hat{s}+1, M \cup \{(r_{d_t}, \dot{v}, \hat{s})\})$, where $\forall r \in D, r_{d_t} \leq r$. In either case, returns $\hat{s}$.
- `enqueue_followup(`$\dot{s}$`)`: If $(r, v, \dot{s}) \in M$ for some r and v, changes Q to be $(D, E, \hat{r}, \hat{s}, M \setminus \{(r, v, \dot{s})\} \cup \{(r, v, 0)\})$ and returns *true*. Otherwise, leaves Q unchanged and returns *false*.
- `enqueue_abort(`$\dot{s}$`)`: If $(v, \dot{s}) \in E$ for some v, changes Q to be $(D, E \setminus \{(v, \dot{s})\}, \hat{r}, \hat{s}, M)$ and returns *true*. Otherwise, leaves Q unchanged and returns *false*.
- `dequeue_reserve()`: If $E = \varnothing$, changes Q to be $(D \cup \{\hat{r}\}, \varnothing, \hat{r} + 1, \hat{s}, M)$. Otherwise, changes Q to be $(\varnothing, E \setminus \{(v_{e_t}, s_{e_t})\}, \hat{r} + 1, \hat{s}, M \cup \{(\hat{r}, v_{e_t}, s_{e_t})\})$, where $\forall (v, s) \in E, s_{e_t} \leq s$. In either case, returns $\hat{r}$.
- `dequeue_followup(`$\dot{r}$`)`: If $(\dot{r}, v, s) \in M$ for some v and s, changes Q to be $(D, E, \hat{r}, \hat{s}, M \setminus \{(\dot{r}, v, s)\} \cup \{(0, v, s)\})$ and returns v. Otherwise, leaves Q unchanged and returns the distinguished value *failed*.
- `dequeue_abort(`$\dot{r}$`)`: If $\dot{r} \in D$, changes Q to be $(D \setminus \{\dot{r}\}, E, \hat{r}, \hat{s}, M)$ and returns *true*. Otherwise leaves Q unchanged and returns *false*.

As mentioned in Section 2.2, we combine `dequeue_reserve()` and `dequeue_followup()` into a single `demand` operation that operates as shown in Listing 2.

An unfair synchronous queue has the same semantics, except that `enqueue_reserve` and `dequeue_reserve` can "partner" with an arbitrary request, not necessarily the one with the lowest sequence number. Our use of the dual stack as the basis for our unfair synchronous queue, and its consequent strict LIFO semantics, is stronger than is strictly necessary for this specification.

An *unabortable* synchronous queue follows the same semantics as the *abortable* queue described here, except that it does not support the `enqueue_abort` or `dequeue_abort` operations.

2.4 Memory Model

The algorithms and code we present in this paper assume compliance with the Java Memory Model [13], which entails "Store-Load" fences for `compareAndSet` and `volatile` writes, plus other ordering constraints that are similar to the SPARC TSO and x86 PO models, and requires no further explicit barrier instructions on most processors. Adapting these algorithms to other memory models consists of inserting appropriate memory barriers.

3. Algorithm Descriptions

In this section we discuss various implementations of synchronous queues. We start with classic algorithms that have seen extensive use in production software, then we review newer implementations that improve upon them. Finally, we describe the implementation of our new algorithms.

```
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 }
```

Listing 3. Naive synchronous queue.

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 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 `put` or `take` operation. It is possible to streamline some of these synchronization points in common execution scenarios by using a fast-path acquire sequence [11]. Such a version appears in early releases of the *dl.util.concurrent* package, which later evolved into *java.util.concurrent*. However, such minor incremental changes improve performance by only a few percent in most applications.

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 [1]; it improves considerably on semaphore-based approaches. First, in the case where a consumer finds a waiting producer or a producer finds a waiting consumer, the new arrival needs to perform only one synchronization

```
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 }
```

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 timeout, details of the way in which AbstractQueuedSynchronizers are used, and code to generalize waitingProducers and waitingConsumers to either stacks or queues.)

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 complete handoff thus requires only three synchronization operations, compared to 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 wake up from 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 timeout and for bidirectional synchronous waiting to our previous nonblocking dual queue and dual stack algorithms [17]. We describe the new algorithms in three steps. First, Section 3.3.1 reviews our earlier dual stack and dual queue and presents the modifications needed to make them synchronous. Second, Section 3.3.2 sketches the manner in which we add timeout support. Finally, Section 3.3.3 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`.

```
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 }
```

Listing 5. Synchronous dual queue: Spin-based enqueue; `dequeue` is symmetric except for the direction of data transfer. (For clarity, code to support timeout is omitted.)

3.3.1 Basic Synchronous Dual Queues and Stacks

Our dual stack and dual queue algorithms [17] already block when a consumer arrives before a producer; extending them to support synchronous handoff is thus a matter of arranging for producers to block until a consumer arrives. We can do this in the synchronous dual queue by simply having a producer block on its data pointer until a consumer updates it to null (implicitly claiming the data). For the synchronous dual stack, *fulfilling requests* extend our "annihilating" approach by pairing with data at the top of the stack just as fulfilling data nodes pair with requests.

The Synchronous Dual Queue

Listing 5 shows the `enqueue` method of the synchronous dual queue. (Except for the direction of data transfer, `dequeue` is symmetric.) To enqueue, we first read the head and tail pointers of the queue (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 claimed our data (15–16), which it does by updating our node's data pointer to null. Then we help remove our

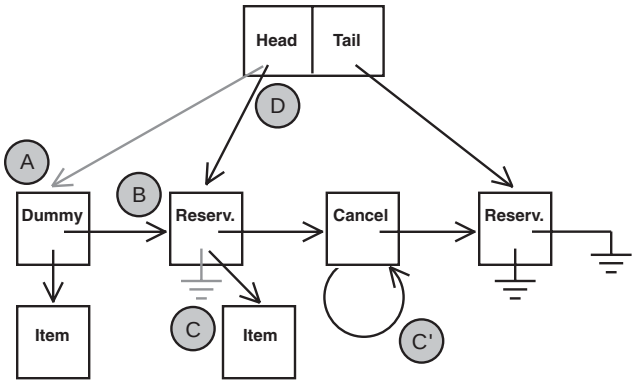

Figure 1. Synchronous dual queue: Enqueuing when reservations are present.

node from the head of the queue and return (18–20). The reservation linearization point for this code path occurs at line 13 when we successfully insert our offering into the queue; a successful followup linearization point occurs when we notice at line 15 that our data has been taken.

The other case occurs when the queue consists of reservations (requests for data), and is depicted in Figure 1 (previous page). (We don't separately depict the first case because except for waiting for a consumer to claim its datum, it is essentially the same as in the original Michael & Scott queue [15].) In this case, 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 head-most 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 reservation linearization point for this code path occurs when we successfully supply data to a waiting consumer at line 27; the followup linearization point occurs immediately thereafter.

The Synchronous Dual Stack

Code for the synchronous dual stack's `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 linearization point for this code path occurs when we push our datum at line 09; a successful followup linearization point occurs when we notice that our data has been taken at line 11.

The second case occurs when the stack contains (only) requests for data (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're 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 linearization point for this code path is at line 19, when we push our fulfilling datum above a reservation; the followup linearization point occurs immediately thereafter.

The remaining case occurs when we find another thread's fulfilling datum or request node (26) at the top of the stack. We must complete the pairing and annihilation of the top two stack nodes

```
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 }
```

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

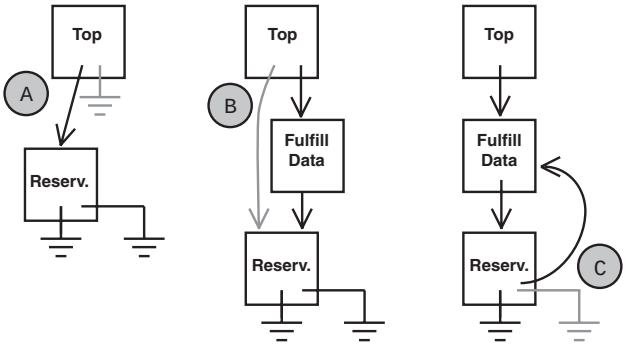

Figure 2. Synchronous dual stack: Satisfying a reservation.

before we can continue our own work. We first read our way down the stack to find the data or request node for which the fulfillment 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 request node into the stack (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 (**B**), it pushes a fulfilling data node to the top of the stack. Then, it swings the request's data pointer to its fulfillment 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 for step **C**; and the consumer thread can additionally help remove itself from the stack in step **D**.

3.3.2 Supporting Timeout

Although the algorithms presented in Section 3.3.1 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 take (provide) our datum. As we noted earlier, Hanson's synchronous queue offers no simple way to do this. A key benefit of our new algorithms is that they support timeout in a relatively straightforward manner.

In the synchronous dual queue, recall that a producer blocks on its data pointer if it finds data in the queue. A consumer that finds the producer's data node head-most in the queue attempts an atomic update to clear the data pointer. If the CAS fails, the consumer assumes that another consumer has taken this datum, so it helps clear the producer's data node from the queue. To support timeout, therefore, it suffices for the producer to clear its own data pointer and leave; when a consumer eventually finds the abandoned node, it will remove it through the existing helping mechanism. To resolve the race wherein the producer attempts to time out at the same time as a consumer attempts to claim its data, the producer needs to clear its data node pointer with an atomic CAS. Similarly, a consumer waiting on the data pointer in a reservation node can simply CAS it back to itself (shown in Figure 1 with the node marked **C'**) and the next producer that finds its request will helpfully remove it from the queue.

Supporting timeout in the synchronous dual stack is similar to supporting it in the synchronous dual queue, but annihilation complicates matters somewhat. Specifically, if a request or data node times out, then a fulfilling node can eventually rest just above it in the stack. When this happens, the abandoned node must be deleted from mid-stack so that the fulfilling node can pair to a request below it (and other threads need to be aware of this need when helping). But what if it was the last data or request node in the stack? We now have a stack consisting of just a fulfillment node. As this case is easy to detect, we handle it by atomically setting the stack pointer to null (which can also be helped by other threads) and having the fulfilling thread start over. Finally, we adopt the requirement that the fulfilling thread cannot time out so long as it has a fulfilling node atop the stack.

3.3.3 Pragmatics

Our synchronous queue implementations reflect a few additional pragmatic considerations to maintain good performance. First, because Java does not allow one to set flag bits in pointers, we add a word to nodes in our synchronous queues within 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 run-time 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.

Adding timeout support to the original dual stack and dual queue [17] requires careful management of memory ownership to ensure that canceled nodes are reclaimed properly. Java's garbage collection removes this burden from the implementations we present in this paper. On the other hand, we must 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).

For sake of clarity, the synchronous queues of Figures 5 and 6 blocked with busy-wait spinning to await a counterpart consumer.

```
00 void clean(Node s) {
01     Node past = s.next;
02     if (past != null && past.isCancelled())
03         past = past.next;
04
05     Node p;
06     while ((p = head) != null && p != past &&
07             p.isCancelled())
08         casHead(p, p.next);
09
10     while (p != null && p != past) {
11         Node n = p.next;
12         if (n != null && n.isCancelled())
13             p.casNext(n, n.next);
14         else
15             p = n;
16     }
17 }
```

Listing 7. Synchronous dual stack: Cleaning canceled nodes (unfair mode).

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 [9]. In practice, we mainly choose the spin-then-yield approach, using the `park` and `unpark` methods contained in *java.util.concurrent.locks.LockSupport* [12] 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 "fly-by" between a producer and consumer without stalling either. On less busy queues, the amount of spinning is small enough not to be noticeable.

Finally, the simplest approach to supporting timeout involves marking nodes 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 timeout 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.

In our implementation, we perform cleaning differently in stacks (unfair mode) and queues. Listing 7 displays the cleaning strategy for stacks; the parameter `s` is a canceled node that needs to be unlinked. This implementation potentially requires an $O(N)$ traversal to unlink the node at the bottom of the stack; however, it can run concurrently with other threads' stack access.

We work our way from the top of the stack to the first node we see past `s`, cleaning canceled nodes as we go. Cleanup occurs in two main phases. First, we remove canceled nodes from the top of the stack (06–08), then we remove internal nodes (10–15). We note that our technique for removing internal nodes from the list is dependent on garbage collection: A node C unlinked from the list by being bypassed can be relinked to the list if its predecessor B becomes canceled and another thread concurrently links B's predecessor A to C. Resolving this race condition requires complicated protocols [20]. In a garbage-collected language, by contrast, unlinked nodes remain unreclaimed while references are extant.

In contrast with our cleaning strategy for stacks, for queues we usually remove a node in $O(1)$ time when it is canceled. However, at any given time, the last node inserted in the list cannot be deleted (because there is no obvious way to update the tail pointer backwards). To accommodate the case where a node is "pinned" at the tail of the queue, we save a reference to its predecessor (in a `cleanMe` field of the queue) after first unlinking any node that was

```
00 void clean(Node pred, Node s) {
01     while (pred.next == s) {
02         Node h = head;
03         Node hn = h.next;
04         if (hn != null && hn.isCancelled()) {
05             advanceHead(h, hn);
06             continue;
07         }
08         Node t = tail;
09         if (t == h)
10             return;
11         Node tn = t.next;
12         if (t != tail)
13             continue;
14         if (tn != null) {
15             advanceTail(t, tn);
16             continue;
17         }
18         if (s != t) {
19             Node sn = s.next;
20             if (sn == s || pred.casNext(s, sn))
21                 return;
22         }
23         Node dp = cleanMe;
24         if (dp != null) {
25             Node d = dp.next;
26             Node dn;
27             if (d == null || d == dp ||
28                 !d.isCancelled() ||
29                 (d != t && (dn = d.next) != null &&
30                  dn != d && dp.casNext(d, dn)))
31                 casCleanMe(dp, null);
32             if (dp == pred)
33                 return;
34         } else if (casCleanMe(null, pred))
35             return;
36     }
37 }
```

Listing 8. Synchronous dual queue: Cleaning canceled nodes (fair mode).

previously saved. Since only one node can be pinned at any time, at least one of the node to be unlinked and the cached node can always be reclaimed.

Listing 8 presents our cleaning strategy for queues. In this code, `s` is a canceled node that needs to be unlinked, and `pred` is the node known to precede it in the queue. We begin by reading the first two nodes in the queue (lines 02–03) and unlinking any canceled nodes from the head of the queue (04–07). Then, we check to see if the queue is empty (08–10), or has a lagging tail pointer (11–17), before checking whether `s` is the tail-most node. Assuming it is not pinned as the tail node, we unlink `s` unless another thread has already done so (18–22). The check in line 20 (`sn == s`) queries whether a canceled node has been removed from the list: We link a canceled node's next pointer back to itself to flag this state.

If `s` *is* pinned, we read the currently saved node `cleanMe` (23). If no node was saved, we save a reference to `s` predecessor and are done (34–35). Otherwise, we must first remove `cleanMe`'s canceled successor (25). If that successor is gone (27, `d == null`) or no longer in the list (27, `d == dp`; recall that nodes unlinked from the list have their next pointers aimed back at themselves), or uncanceled (28), we simply clear out the `cleanMe` field. Else, if the successor is not currently tail-most (29), and is still in the list (30, `dn != d`), we remove it (30, `casNext` call).

4. Experimental Results

4.1 Benchmarks

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 $N{:}N$. Separately, we stress the timeout code by dynamically adjusting patience between the longest that fails and the shortest that succeeds.

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*.

4.2 Methodology

We obtained results on a SunFire V40z with 4 2.4GHz AMD Opteron processors and on a SunFire 6800 with 16 1.3GHz UltraSPARC 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 threads. We tested each benchmark with both the fair and unfair (stack-based) versions of the Java SE 5.0 *java.util.concurrent.SynchronousQueue* and our new nonblocking algorithms. For tests that do not require timeout, we additionally tested with Hanson's synchronous queue.

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 transfered from a single producer to multiple consumers; Figure 5 displays the rate at which a single consumer receives data from multiple producers. Figure 6 displays the handoff attempt rate, given very limited patience for producers and consumers. Figure 7 presents execution time per task for our *ThreadPoolExecutor* benchmark.

4.3 Discussion

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 pile-ups 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 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.

The dynamic timeout test (Figure 6) reveals another deficiency in the Java SE 5.0 *SynchronousQueue* implementation: Fair mode has a pathologically bad response to timeout, due mainly to the cost of time-out in its fair-mode reentrant locks. Meanwhile, either version of our new algorithm outperforms the unfair-mode Java SE 5.0 *SynchronousQueue* by a factor of five.

Finally, in Figure 7, we see that the performance differentials we observe in *java.util.concurrent.SynchronousQueue* translate directly into overhead in the *java.util.concurrent.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

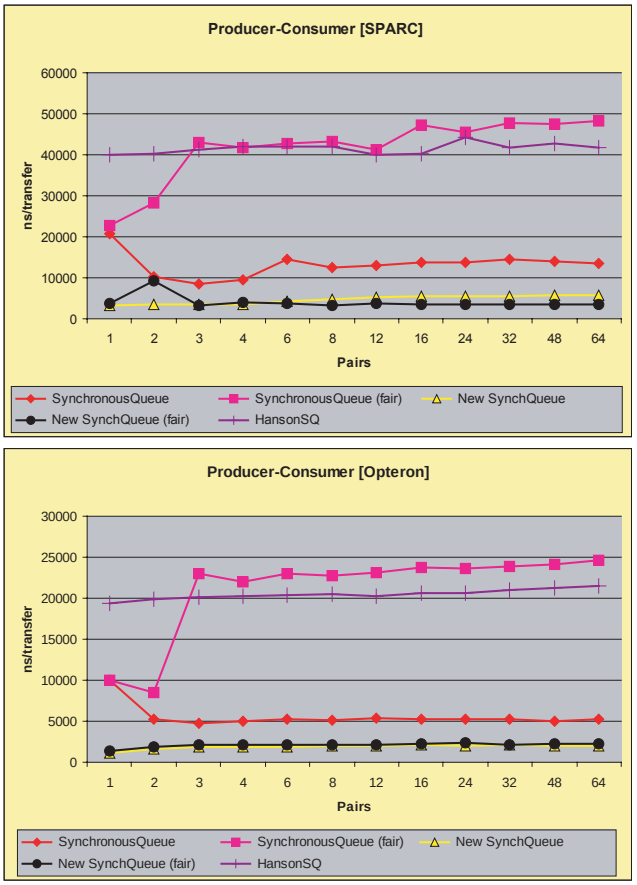

Figure 3. Synchronous handoff: N producers, N consumers.

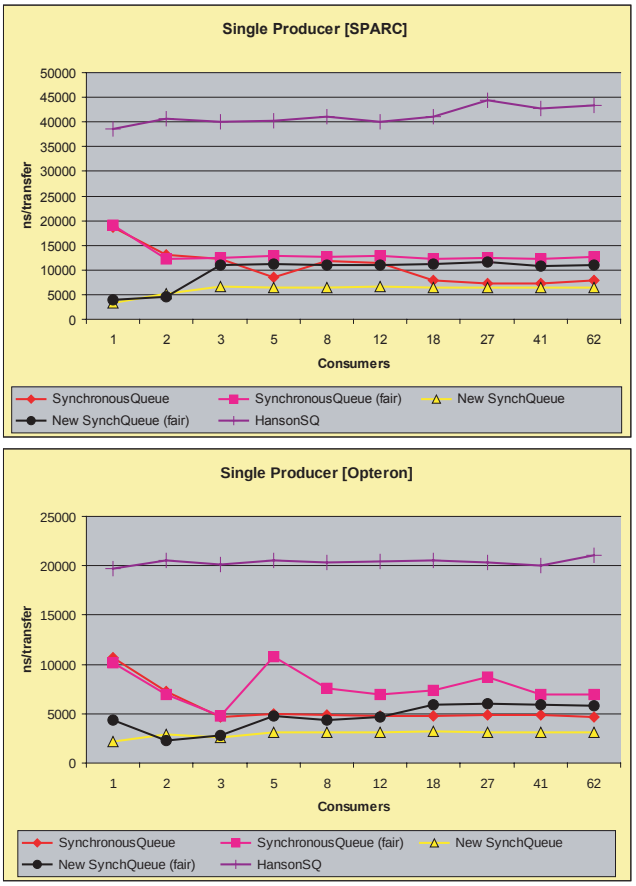

Figure 4. Synchronous handoff: 1 producer, N consumers.

versions of our new algorithm differs between the SPARC and Opteron 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 platform, 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. By way of comparison, the cost of context switches is relatively smaller on the Opteron platform, 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 the level of concurrency.

5. Conclusions

In this paper, we have presented two new lock-free and contention-free synchronous queue implementations 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.

That our new lock-free synchronous queue implementations are more scalable than the Java SE 5.0 *SynchronousQueue* class is unsurprising: Nonblocking algorithms often scale far better than corresponding lock-based algorithms. More surprisingly, our new implementations are faster even at low levels of contention; nonblocking algorithms often exhibit greater base-case overhead than do lock-based equivalents. Based on our experiments, we recommend our new algorithms anywhere synchronous handoff is needed.

Although we have improved the scalability of the synchronous queue, we believe that there is room for further improvement. In particular, the *elimination* technique introduced by Shavit and Touitou [21] seems promising. This technique arranges for pairs of operations that collectively effect no change to a data structure to meet in a lower-traffic memory location and "cancel" each other out. This simultaneously allows us to reduce contention on the central data structure and to effect multiple concurrent operations (which otherwise would have been serialized as a consequence of manipulating a single memory location). For example, in a stack, a push and a pop can meet in a backoff *arena* to eliminate each other without requiring any modifications to the top-of-stack pointer.

The elimination technique has been used by Hendler et al. [4] to improve the scalability of stacks, by Moir et al. [16] to improve the scalability of queues, and by ourselves [18] to improve the scalability of exchange channels. We believe elimination can yield modest gains in scalability for synchronous queues by reducing contention on the endpoints of component data structures.

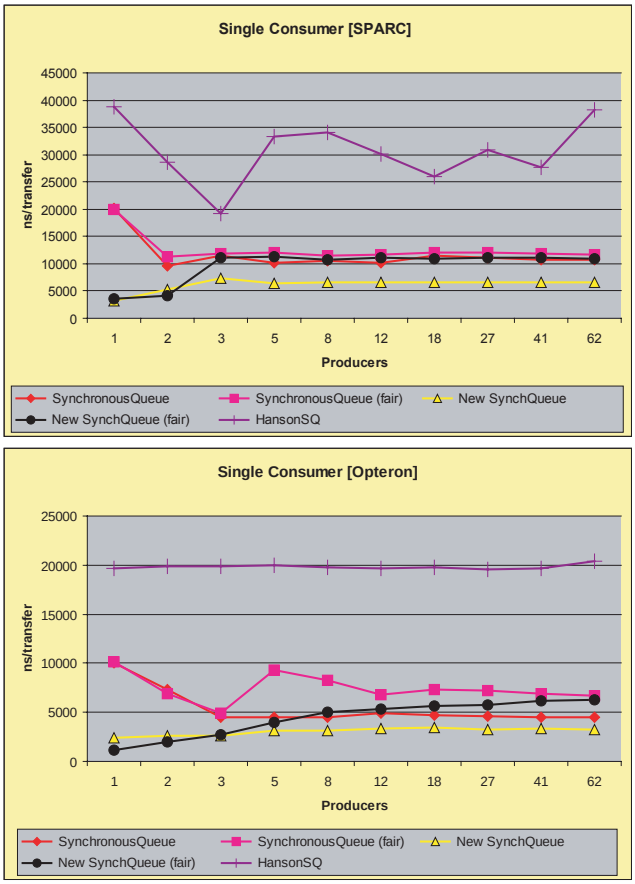

Figure 5. Synchronous handoff: N producers, 1 consumer.

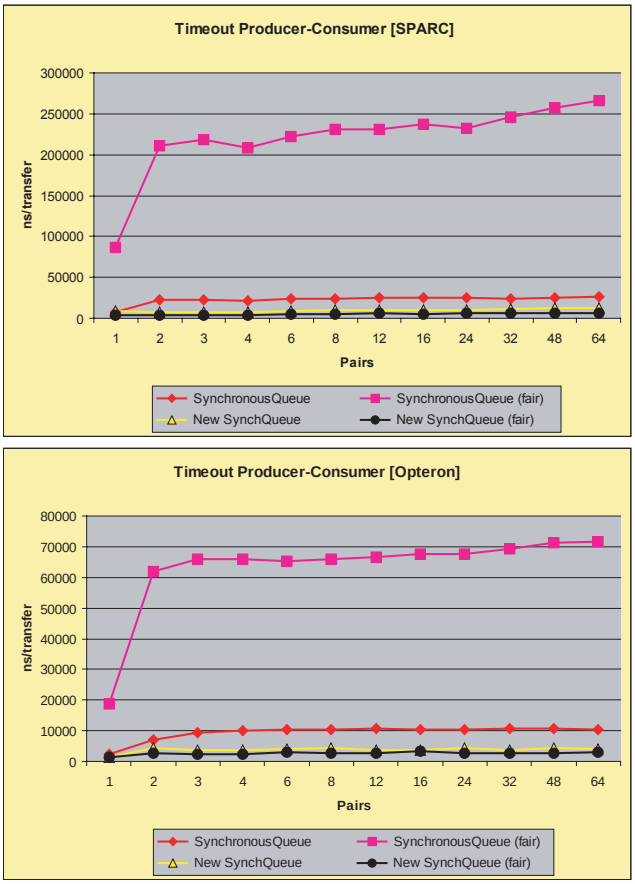

Figure 6. Synchronous handoff: low patience transfers.

Acknowledgments

We are grateful to Dave Dice, Brian Goetz, David Holmes, Mark Moir, Bill Pugh, and the anonymous referees for giving feedback that significantly improved the presentation of this paper.

References

[1] T. E. Anderson, E. D. Lazowska, and H. M. Levy. The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors. *IEEE Trans. on Computers*, 38(12):1631–1644, Dec. 1989.

[2] G. R. Andrews. *Concurrent Programming: Principles and Practice*. Benjamin/Cummings, Redwood City, CA, 1991.

[3] D. R. Hanson. *C Interfaces and Implementations: Techniques for Creating Reusable Software*. Addison-Wesley, Menlo Park, CA, 1997.

[4] D. Hendler, N. Shavit, and L. Yerushalmi. A Scalable Lock-Free Stack Algorithm. In *Proc. of the 16th Annual ACM Symp. on Parallelism in Algorithms and Architectures*, pages 206–215, Barcelona, Spain, June 2004.

[5] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-Free Synchronization: Double-Ended Queues as an Example. In *Proc. of the 23rd Intl. Conf. on Distributed Computing Systems*, Providence, RI, May, 2003.

[6] M. P. Herlihy and J. M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463–492, July 1990.

[7] M. Herlihy. Wait-Free Synchronization. *ACM Trans. on Programming Languages and Systems*, 13(1):124–149, Jan. 1991.

[8] C. A. R. Hoare. Communicating Sequential Processes. *Comm. of the ACM*, 21(8):666–677, Aug. 1978.

[9] A. R. Karlin, K. Li, M. S. Manasse, and S. Owicki. Empirical Studies of Competitive Spinning for a Shared-Memory Multiprocessor. In *Proc. of the 13th ACM Symp. on Operating Systems Principles*, pages 41–55, Pacific Grove, CA, Oct. 1991.

[10] L. Lamport. A New Solution of Dijkstra's Concurrent Programming Problem. *Comm. of the ACM*, 17(8):453–455, Aug. 1974.

[11] L. Lamport. A Fast Mutual Exclusion Algorithm. *ACM Trans. on Computer Systems*, 5(1):1–11, Feb. 1987.

[12] D. Lea. The java.util.concurrent Synchronizer Framework. *Science of Computer Programming*, 58(3):293–309, Dec. 2005.

[13] J. Manson, W. Pugh, and S. Adve. The Java Memory Model. In *Conf. Record of the 32nd ACM Symp. on Principles of Programming Languages*, Long Beach, CA, Jan. 2005.

[14] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Trans. on Computer Systems*, 9(1):21–65, Feb. 1991.

[15] 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*, pages 267–275, Philadelphia, PA, May 1996.

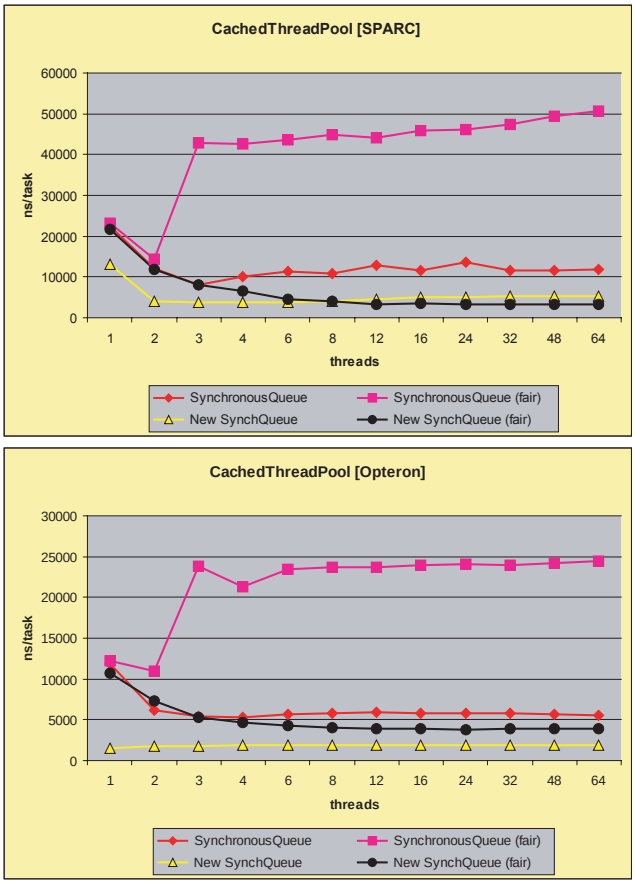

Figure 7. ThreadPoolExecutor benchmark.

[16] M. Moir, D. Nussbaum, O. Shalev, and N. Shavit. Using Elimination to Implement Scalable and Lock-Free FIFO Queues. In *Proc. of the 17th Annual ACM Symp. on Parallelism in Algorithms and Architectures*, pages 253–262, Las Vegas, NV, July 2005.

[17] W. N. Scherer III and M. L. Scott. Nonblocking Concurrent Objects with Condition Synchronization. In *Proc. of the 18th Intl. Symp. on Distributed Computing*, Amsterdam, The Netherlands, Oct. 2004.

[18] W. N. Scherer III, D. Lea, and M. L. Scott. A Scalable Elimination-based Exchange Channel. In *Proc., Workshop on Synchronization and Concurrency in Object-Oriented Languages*, San Diego, CA, Oct. 2005. In conjunction with OOPSLA'05.

[19] W. N. Scherer III. Synchronization and Concurrency in User-level Software Systems. Ph. D. dissertation, Dept. of Computer Science, Univ. of Rochester, Jan. 2006.

[20] M. L. Scott and W. N. Scherer III. Scalable Queue-Based Spin Locks with Timeout. In *Proc. of the 8th ACM Symp. on Principles and Practice of Parallel Programming*, Snowbird, UT, June 2001.

[21] N. Shavit and D. Touitou. Elimination Trees and the Construction of Pools and Stacks. In *Proc. of the 7th Annual ACM Symp. on Parallel Algorithms and Architectures*, Santa Barbara, CA, July 1995.