Fast and Scalable Rendezvousing
(Extended Abstract)

Yehuda Afek   Michael Hakimi   Adam Morrison

School of Computer Science
Tel Aviv University

Abstract

In an asymmetric rendezvous system, such as an unfair synchronous queue and an elimination array, threads of two types, consumers and producers, show up and are matched, each with a unique thread of the other type. Here we present a new highly scalable, high throughput asymmetric rendezvous system that outperforms prior synchronous queue and elimination array implementations under both symmetric and asymmetric workloads (more operations of one type than the other). It is a fast matching machine. Consequently, we also present a highly scalable and competitive stack implementation.

Intended for Regular Presentation.

Eligible for Best Student Paper Award.

Contact Author:

Yehuda Afek

Phone:	+972-544-797322
Fax:	+972-3-640-9357
Email:	`afek@tau.ac.il`
Post:	School of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel

1 Introduction

A common abstraction in concurrent programming is that of an *asymmetric rendezvous* mechanism. In this mechanism, there are two types of threads, e.g., producers and consumers, that show up. The goal is to match pairs of threads, one of each type, and send them away. Usually the purpose of the pairing is for a producer to hand over a data item (such as a task to perform) to a consumer. The asymmetric rendezvous abstraction encompasses both *unfair synchronous queues* (or *pools*) [14] which are a key building block in Java's thread pool implementation and other message-passing and hand-off designs [4, 14], and the *elimination* technique [15], which is used to scale concurrent stacks and queues [9, 13].

In this paper, we present a highly scalable asymmetric rendezvous algorithm that improves the state of the art in both unfair synchronous queue and elimination algorithms. It is based on a distributed scalable ring structure, unlike Java's synchronous queue which relies on a non-scalable centralized structure. It is *nonblocking*, in the following sense: if both producers and consumers keep taking steps, *some* rendezvous operation is guaranteed to complete. (This is similar to the *lock-freedom* property [10], while taking into account the fact that "it takes two to tango", i.e., both types of threads must take steps to successfully rendezvous.) It is also uniform, in that no thread has to perform work on behalf of other threads. This is in contrast to the flat combining (FC) based synchronous queues of Hendler et. al. [8], which are blocking and non-uniform. Most importantly, our algorithm performs extremely well in practice, outperforming Java's synchronous queue by up to 60× on an UltraSPARC T2 Plus multicore machine, and the FC synchronous queue by up to a factor of six. Using our algorithm as the elimination layer of a concurrent stack yields a factor of three improvement over Hendler et. al.'s FC stack [7].

Our algorithm is based on a simple idea that turns out to be remarkably effective in practice: the *algorithm itself is asymmetric*. While a consumer captures a node in the shared ring structure and waits there for a producer, a producer actively seeks out waiting consumers on the ring. We additionally present two new techniques, that combined with this asymmetric methodology on the ring, lead to extremely high performance and scalability in practice. The first is a new *ring adaptivity scheme* that dynamically adjusts the ring's size, leaving enough room to fit in all the consumers while avoiding empty nodes that producers will needlessly need to search. Having an adaptive ring size, we can expect the nodes to be mostly occupied, which leads us to the next idea: if a producer starts to scan the ring and finds the first node to be empty, it is likely that a consumer will arrive there shortly. Yet simply waiting at this node, hoping that this will occur, makes the algorithm prone to timeouts and impedes progress. We solve this problem by employing a *peeking* technique that lets the producer have the best of both worlds: as the producer traverses the ring, it continues to peek at its initial node; should a consumer arrive there, the producer immediately tries to partner with it, thereby minimizing the amount of wasted work. Our algorithm avoids two problem found in prior elimination algorithms that did not exploit asymmetry. In these works [3, 9, 15], both types of threads would pick a random node in the hope of meeting the right kind of partner. Thus these works suffer from *false matches*, when two threads of the same type meet, and from *timeouts*, when a producer and a consumer both pick distinct nodes and futilely wait for a partner to arrive.

The algorithm is presented in Section 3, after formally defining the asymmetric rendezvous problem and our progress property (Section 2). The empirical evaluation is presented in Section 4, and

Section 5 concludes. Due to space limitations, we survey related work in Appendix A.

2 Preliminaries

Asymmetric rendezvous In the *asymmetric rendezvous* problem there are threads of two types, producers and consumers. Producers perform `put(`x`)` operations, that return either `OK` or a reserved value $\perp$ (explained shortly). Consumers perform `get()` operations, which return some item x handed off by a producer, or the reserved value $\perp$. Producers and consumers show up and must be matched with a unique thread of the other type, such that a producer invoking `put(`x`)` and a consumer whose `get()` returns x must be active concurrently. Since this synchronous behavior inherently requires waiting, we provide threads with the option to give up: the special return $\perp$ is used to indicate such a timeout. We say that an operation *completes successfully* if it returns a value other than $\perp$.

Progress To reason about progress while taking into account that rendezvous inherently requires waiting, we consider both types of operations' combined behavior. An algorithm **A** that implements asymmetric rendezvous is *nonblocking* if, *some* operation completes successfully after both `put()` *and* `get()` take enough steps *concurrently*. Note that, as with the definition of the lock-freedom property [10], there is no fixed a priori bound on the number of steps after which some operation must complete successfully. Rather, we rule out implementations that make no progress at all, i.e., admit executions in which both types of threads take steps infinitely often and yet no operation successfully completes. By only counting successful operations as progress, we also rule out the trivial implementation that always returns $\perp$ (i.e., times out immediately).

3 Algorithm Description

The central data structure in the algorithm is a ring of P nodes, `ring`$[0], \ldots,$ `ring`$[P-1]$, where P is the number of processors in the system. Each ring node contains two pointers, `next` and `item`, and an `index` field that holds the node's index in the ring. The `next` pointer is used to traverse the ring: `ring`$[i]$`.next` points to `ring`$[i-1]$. This orientation allows the ring to be *resized* by changing the `next` pointer of `ring`$[0]$, which we henceforth call the *head*. The `item` pointer in each ring node encodes the node's state:

1. **Free**: `item` points to a global reserved object, `FREE`, that is distinct from any object a producer may enqueue. Initially all nodes are free.
2. **Captured by consumer:** `item` is `NULL`.
3. **Holding data (of a producer):** `item` points to the data.

The complete pseudo code of the algorithm is provided in Figure 1. We proceed to describe the algorithm in two stages. Section 3.1 describes the basics of the algorithm, without adaptivity (i.e., assuming the ring size is fixed to P). In Figure 1, the lines of code responsible for handling adaptivity are colored blue and are described later, in Section 3.2. We discuss correctness and progress in Section 3.3.

3.1 Nonadaptive algorithm

Consumer algorithm A consumer searches the ring for a free node and attempts to capture it using a `compareAndSet` (CAS) to atomically change its `item` pointer from `FREE` to `NULL`. Once a node is captured, the consumer spins, waiting for a producer to arrive and deposit its item. The pseudo code for the consumer's `get()` procedure is presented in Figure 1a. (For the moment, the pseudo code responsible for handling adaptivity, which is colored blue, will be ignored.) The search begins at a node, s, obtained by hashing the thread's id (Line 7). The consumer passes the ring size (read at Line 6) to the hash function as a parameter, to ensure the returned node falls within the ring. The consumer calls the `findFreeNode` procedure to traverses the ring starting at s and return a captured node, u (Lines 28-44). The consumer then waits until a producer deposits an item in u, frees u and returns (Lines 21-22).

Producer algorithm Producers search the ring for waiting consumers, and attempt to hand them their data. The pseudo code is presented in Figure 1b. Similarly to the consumer, a producer hashes its id to obtain a starting point for its search, s. (Line 50-51). It then traverses the ring looking for nodes captured by consumers. Here the producer periodically *peeks* at the initial node s to see if it has a waiting consumer (Line 55-56); if not, it checks the current node in the traversal (Line 58). Once a captured node is detected, the producer tries to deposit its data using a CAS (Lines 60-64). If successful, it returns.

Aborting rendezvous Producers can abort by returning. Consumers must free the node they have captured by CASing it from `NULL` back to `FREE`. If the CAS fails, the consumer has found a match and its rendezvous has completed successfully, otherwise its abort attempt succeeded and it returns.

3.2 Adding adaptivity

The goal of our adaptivity scheme is to correctly size the ring, so that its size is "just right": not too small, to avoid contention on the nodes, and not too large, to avoid excessive searching of empty nodes by producers. The technical work of resizing the ring is done by the procedure `resizeRing`, whose pseudo code appears in Figure 1c. This procedure is passed the node that is currently pointed to by the ring head, i.e., the *tail* and attempts to CAS the head's `next` pointer from the tail to its predecessor (to increment the ring size) or its successor (to decrement the ring size). If the CAS fails, then another thread has resized the ring and so `resizeRing` returns.

The logic driving the resizing process is in the consumer's algorithm. If a consumer fails to capture many nodes in `findFreeNode()` (due to not finding a free node or having its CASes fail), then the ring is too crowded and should be increased. The exact threshold is determined by an *increase threshold* parameter, T_i. If `findFreeNode()` fails to capture more than $T_i \cdot$ `ring_size` it attempts to increase the ring (Lines 38-42).

How can we detect if the ring is sparsely populated? In such a case, we expect that, on one hand, a consumer quickly finds a free node, but on the other hand, it has to wait longer until a producer finds *it*. We take such a scenario as a sign that the ring's size should be decreased. Thus, we have a *decrease threshold*, T_d, and a *wait threshold*, T_w. If it takes a consumer more than T_d iterations

```
1  object get(threadId) {
2      **locals**:
3      node s, u
4      int ring_size , busy_ctr, ctr
5
6      ring_size := head.next.index
7      s := ring[ hash(threadId, ring_size ) ]
8      (u, busy_ctr) := findFreeNode(s, ring_size )
9
10     ctr := 0
11     while (u.item == NULL) {
12         ring_size := head.next.index
13         if (u.index > ring_size) {
14             if (CAS(u.item, NULL, FREE) {
15                 s := ring[hash(threadId, ring_size )]
16                 (u, busy_ctr) := findFreeNode(s, ring_size )
17             }
18         }
19         ctr := ctr + 1
20     }
21     item := u.item
22     u.item := FREE
23     if (busy_ctr < $T_d$ and ctr > $T_w$)
24         resizeRing(ring[ ring_size ], −1)
25     return item
26 }
27
28 node findFreeNode(node s, ring_size) {
29     locals:
30     int busy_ctr := 0
31
32     while (true) {
33         if (s.item == FREE_NODE and
34             CAS(s.item, FREE_NODE, NULL))
35             return (s,busy_ctr)
36         s := s.next
37         busy_ctr := busy_ctr + 1
38         if (busy_ctr > $T_i$· ring_size) {
39             resizeRing(ring[ ring_size ], +1)
40             ring_size := head.next.index
41             busy_ctr := 0
42         }
43     }
44 }
```

(a) Consumer algorithm

```
45 put(threadId, object item) {
46     locals:
47     node s, u, target
48     int ring_size , busy_ctr, ctr
49
50     ring_size := head.next.index
51     node s := ring[hash(threadId, ring_size )]
52
53     node v := s.next
54     while (true) {
55         if (s.item == NULL)
56             target := s
57         else
58             target := v
59
60         if (target.item == NULL) {
61             if (CAS(target.item, NULL, item)) {
62                 return OK
63             }
64         }
65         v := v.next
66     }
67 }
```

(b) Producer algorithm

```
68 void resizeRing(node tail, delta) {
69     index := tail.index
70     /* delta is −1 or +1 */
71     if (delta == +1 and index == $P − 1$)
72         return
73     if (delta == −1 and index == 0)
74         return
75     CAS(head.next, tail, ring[index + delta])
76 }
```

(c) Resizing

Figure 1: Asymmetric rendezvous algorithm. Lines colored blue handle the adaptivity process.

of the loop in Lines 11-20 to successfully complete the rendezvous, but it successfully captured its ring node in up to T_w steps, then it attempts to decrement the ring size (Lines 23-24).

Impact on consumer algorithm As a result of decrementing the ring size, a consumer waiting on a node may be left outside of the ring. To notice this scenario, a consumer checks if its node's index is larger than the current ring size (Line 13). If so, the consumer tries to free its node using a CAS (Line 14) and find itself a new node in the ring (Lines 15-16). However, if the CAS fails, then a producer has already deposited its data in this node and so the consumer can take it and return (this will be detected in the next execution of Line 11). In practice, to improve performance, the consumer performs this check periodically and not on every iteration.

3.3 Correctness

We consider the point at which both `put(`x`)` and the `get()` that returns x take effect as the point where the producer successfully CASes x into some node's `item` pointer (Line 61). The algorithm thus clearly meets the asymmetric rendezvous semantics. We next sketch a proof that, assuming threads do not request timeouts, then our algorithm is nonblocking (using the notion defined in Section 2). Assume towards a contradiction that there is an execution in which both producers and consumers take infinitely many steps and yet no rendezvous successfully completes. Since there is a finite number of threads, P, there must be a producer/consumer pair, p and c respectively, that runs forever without completing. Suppose c never captures a node. After completing a cycle on the ring (say the increase threshold is 1) it increments the ring's size. Recall that the ring size decrementing implies a rendezvous completing, which by our assumption does not occur. Therefore, eventually the ring's size is P. From this point on, c can only fail to capture a node if it encounters another consumer twice at different nodes, implying this consumer completes a rendezvous, a contradiction. Thus, it must be that c captures some node but p never finds c on the ring, implying that c has been left out of the ring by some decrementing resize. But, since from that point on, the ring's size cannot decrease, c eventually executes Lines 13-17 and moves itself into p's range, where either p or another producer rendezvous with it, a contradiction.

4 Evaluation

We first evaluate our algorithm when used as an unfair synchronous queue, and compare its performance to other unfair synchronous queue implementations (Section 4.1). Then we examine the performance of an elimination back-off stack with our algorithm used as the elimination layer, and compare it with other concurrent stack implementations (Section 4.2). Due to space constraints, we relegate the evaluation of the efficacy of the peeking technique to Appendix B.4.

Experimental setup: We collect results on two hardware platforms: a Sun SPARC T5240 and an Intel Core i7. The Sun is a dual-chip machine, powered by two UltraSPARC T2 Plus (Niagara II) chips [1]. The Niagara II is a chip multithreading (CMT) processor, with 8 1.165 GHz in-order cores with 8 hardware strands per core, for a total of 64 hardware strands per chip. The Core i7 920 (Nehalem [2]) processor in the Intel machine holds four 2.67GHz cores that each multiplex 2 hardware threads. We implemented the algorithm in both Java and C++[1]. Unless stated otherwise, reported results are averages of ten 10-second runs of the Java implementation on an otherwise idle machine, resulting in very little variance.

Parameters: The increase threshold is $T_i = 1$ (ring's size is increased once a consumer observes a full ring). The decrease and wait thresholds are set to $T_d = 2$ and $T_w = 256$. We use a simple modulo ring size for a hash function. The rationale for this choice is as follows. If we don't

[1]Java benchmarks were ran with HotSpot Server JVM, build `1.7.0-ea-b137`. C++ benchmarks were compiled with Sun C++ 5.9 on the SPARC machine and with `gcc` 4.3.3 (`-O3` optimization setting) on the Intel machine. In the C++ experiments we used the Hoard 3.8 [5] memory allocator.

know anything about the thread ids showing up, we model them as uniformly random, in which case a modulo provides uniform distribution. A different likely case, providing an optimization opportunity, is when thread ids are *sequential*. Because the ring is sized to fit the contention level, in this case the modulo hash achieves perfect coverage of the ring with few collisions and helps to form pairs of producer/consumers that will often match with each other. Here a hash function that mixes the thread ids is detrimental to performance as it induces collisions in a scenario where they can be avoided. We evaluate these effects by testing our algorithm with both random and sequential thread ids throughout all experiments.

Due to space constraints, we report here only the results obtained on the Sun platform, since it offers more parallelism and thus more insight into the scaling behavior of the algorithm. (The Nehalem results, which are qualitatively similar to the low thread count SPARC results, are presented in Appendix B.5.) While we discuss results obtained both when restricting the benchmarks to run on a single chip and when allowing full use of the machine, we follow Hendler et. al.'s methodology [8] and focus on single chip results.

4.1 Synchronous queues

We compare the algorithm to the best performing algorithms described in the literature, namely FC synchronous queues [8], Elimination-Diffraction (ED) trees [3], and the Java unfair synchronous queue [14]. The Java pool is based on a Treiber-style nonblocking stack [18] that at all times contains rendezvous requests by either producers or consumers. A producer finding the stack empty or containing producers pushes itself on the stack and waits, but if it finds the stack holding consumers, it attempts to partner with the consumer at the top of the stack (consumers behave symmetrically). This creates a sequential bottleneck. Afek et. al.'s ED tree is a randomized distributed data structure where arriving threads follow a path through a binary tree whose internal nodes are *balancer* objects [16] and the leaves are Java synchronous queues. In each internal node a thread accesses an elimination array in attempt to avoid descending down the tree. In FC pools, a thread attempts to become a *combiner* by acquiring a global lock on the queue. Threads that fail to grab the lock instead post their request and wait for it to be fulfilled by the combiner, which matches between the participating threads. In the parallel FC pool there are multiple combiners that each handle a subset of participating threads and then try to satisfy unmatched requests in their subset by entering an *exchange* FC synchronous queue.

We use the original authors' implementation of FC synchronous queues and ED trees [7, 8][2]. We also test a version of the Java synchronous queue that always uses busy waiting instead of yielding the CPU (so-called *parking*), and report its results for the workloads where it improves upon the standard Java pool (as was done in [8]).

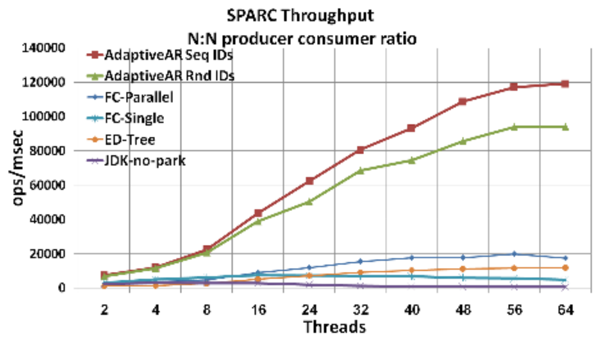
[2]To get the best possible results, we remove all statistics counting from the code and use the latest JVM. Thus, the results we report are usually slightly better than those reported in the original papers. On the other hand, we fixed a bug in the benchmark of [8] that miscounted timed-out operations of the Java pool as successful operations; thus the results we report for it are sometimes lower.

4.1.1 Producer/consumer throughput

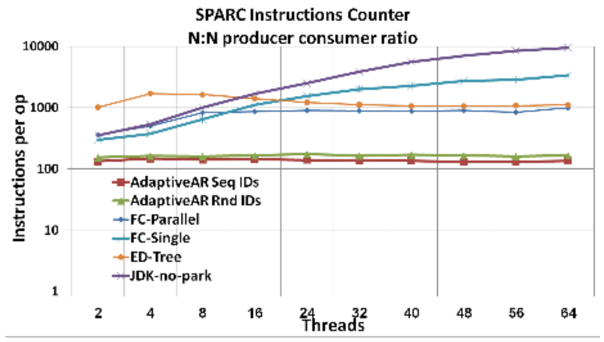
Multiple producers and consumers (symmetric workload) We measure the rate at which data is transferred from multiple producers to an identical number of consumers, and collect hardware performance counters data during the run. The results are shown in Figure 2a. Other than our algorithm, the parallel FC pool is the only algorithm that shows significant scalability. Our rendezvous algorithm outperforms the parallel FC queue by a factor of three in low concurrency settings, and by up to six at high thread counts. It achieves this result by *efficiently* exploiting the available hardware parallelism: a rendezvous completes in less than 200 instructions, of which one is a CAS. The parallel FC pool, while hardly performing CASes, does require between 3× to 6× more instructions to complete an operation. Waiting for the combiner to pick up a thread's request, match it, and report back with the result, all add up. Similarly, while the ED tree scales up to 40 threads, its rendezvous operation are costly, requiring 1000 instructions to complete. Our algorithm therefore outperforms it by at least 6×, and up to 10× at high thread counts. As was previously shown [8], the Java synchronous queue fails to scale in this benchmark. We outperform it by up to 60×. Figure 2b shows the serializing behavior of the Java and single FC algorithms. Both require more instructions to complete an operation as concurrency grows. For the Java pool this is due to the number of failed CAS operations on the head of the stack (and consequent retries), and for FC it is a result of the increased time spent as the combiner struggles to serve all waiting threads.

Due to space limitation, the throughput results of this workload when utilizing both processors of the machine are depicted in Appendix B.1 (Figure 6). In this test, the operating system's default scheduling policy is to place threads round-robin on the chips, and round-robin on the cores within each chip. The costs incurred due to cross-chip communication are therefore noticeable even at low thread counts. These costs are due to an increased number of L1 and L2 cache misses, since the threads no longer share a single L2 cache; each such cache miss is expensive, requiring coherency protocol traffic to the remote chip. The effect is catastrophic for serializing algorithms; for example, the Java pool experiences a 10× drop in throughput. The more parallel algorithms, such as parallel FC and ours, show similar scaling trends but achieve lower throughput. We therefore focus the rest of our evaluation on the more interesting single chip case.

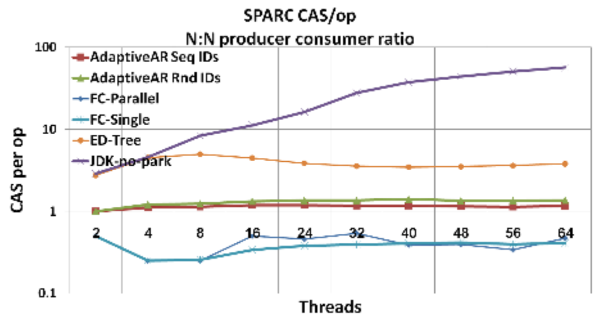
Asymmetric workloads We measure the rate at which a single producer hands off data to a varying number of consumers. Because the pool is synchronous, we expect little scaling in this workload: throughput is bounded by the rate of a single producer. However, as Figure 3a shows, for all algorithms (but the ED tree) it takes several consumers to keep up with a single producer: we see throughput increase when more than one consumer is active. The reason is that in these algorithms the producer hands off an item and completes, whereas the consumer then needs to notice the rendezvous has occurred. In this window of time the producer cannot make progress on a new rendezvous. However, with several consumers, the producer can immediately return and find another (different) consumer ready to rendezvous with. Unfortunately, as shown in Figure 3a, most algorithm do not sustain this peak throughput. The FC pools have the worst degradation (3.74× for the single version, and 3× for the parallel version). The Java pool's degradation is minimal (13%), and it along with the ED tree achieves close to peak throughput even at high thread counts. Yet this throughput is low: our algorithm outperforms the Java pool by up to 3× and the ED tree by 12× for low consumer counts and 6× for high consumer counts, despite degrading by 30% from peak throughput.



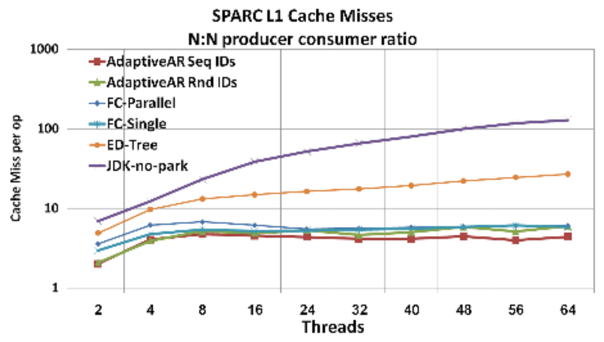
(a) Throughput.



(b) Instructions (per op).



(c) CASes (per op).

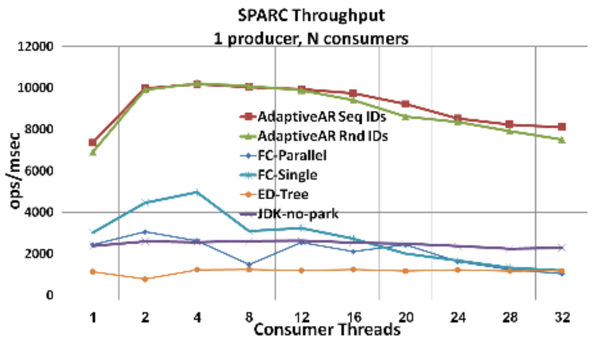


(d) L1 cache misses (per op).

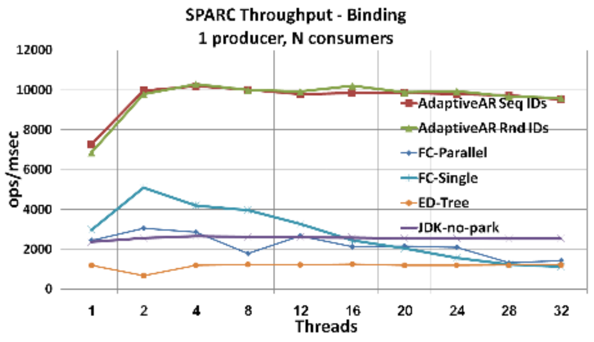
Figure 2: Rendezvousing between N pairs of producers and consumers. Graphs show rate of completed operations per millisecond and hardware performance counter datas per synchronous queue operation. All but throughput are logarithmic scale. L2 misses are not shown; all algorithms but Java had less than one L2 miss/operation on average.

But why does our algorithm degrade? After all, the producer has its pick of waiting consumers in the ring and should be able to complete its hand-off immediately. It turns out that the degradation is not caused by an algorithmic problem, but due to *contention on chip resources*. Each core of the Niagara II has two pipelines, with each pipeline shared by four hardware strands. While the operating system schedules threads on hardware strands in a way that minimizes sharing of pipelines, beyond 16 threads the OS has no choice but to have some threads that share a pipeline — and our algorithm indeed starts to degrade at 16 threads. To demonstrate this issue, we perform another test where we use thread binding to ensure that the producer runs on the first core and consumers are scheduled only on the remaining seven cores. Figure 3b depicts the result of this experiment. While the trends of the other algorithms are unaffected, our algorithm now maintains peak throughput through all consumer counts.

In the opposite workload (Figure 8 in Appendix B.1), where multiple producers try to serve a single consumer, the producers contend over the single node the consumer occupies. As a result, the throughput of our algorithm degrades as the number of producers increases, as do the FC pools. Despite this degradation, our algorithm outperforms the Java pool up to 48 threads (falling behind by 15% at 64 threads) and outperforms the FC pools by about 2×.



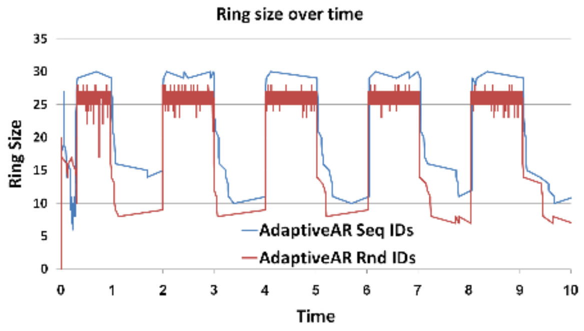
(a)



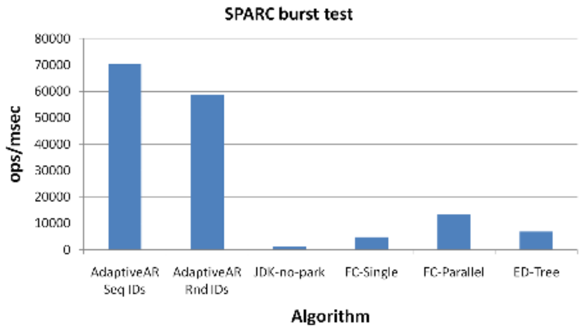
(b)

Figure 3: Synchronous queue throughput with a single producer and N consumers. In the left side, the scheduler freely places threads onto hardware strands. In the right side, scheduling is constrained so that the producer has a core to itself and the consumers are scheduled on the remaining seven cores.

Bursts To evaluate the effectiveness of our adaptivity technique, we measure the rendezvous rate in a workload that experiences bursts of activity. Over the ten second interval of the run, the workloads alternates between 31 thread pairs and 8 pairs, with each phase lasting for a second. We use 31 thread pairs so that we have spare hardware strand to use for sampling the ring size throughout the run. Our sampling thread continuously reads the ring size, and records the time whenever the current read differs from the previous read. Figure 4a depicts the result, showing how the algorithm continuously resizes its ring. Consequently, it successfully benefits from the available parallelism in this workload, outperforming the Java pool by up to 50× and the parallel FC pool by 4× to 5×.



(a)



(b)

Figure 4: Synchronous queue bursty workload throughput. The execution alternates between with 31 and 8 producer/consumer pairs, with each phase lasting one second. The left side shows our algorithm's ring size over time (sampled continuously using a thread that does not participate in the rendezvousing). Throughput is shown on the left side.

Changing arrival rates In practice, we expect threads to do some work between rendezvouses. We therefore measure how the throughput of the 32 prodcuer/consumer pairs workload is affected when the thread arrival rate decreases due to increasingly larger amounts of time spent doing "work" before each rendezvous. We find that as the work period grows and the available parallelism in the data structure decreases, the throughput of all algorithms that exhibit scaling deteriorates. However, our algorithm outperforms the other implementations by a factor of at least three. The full results are relegated to Appendix B.2.

Work uniformity We evaluate how *uniformly* is the distribution of work done by the threads in each algorithm, by comparing the percent of total operations performed by each thread in a multiple producer/multiple consumer workload. The Java pool and our algorithm are close to uniform, but parallel FC is extremely non-uniform, with some threads doing up to 33× the work as others. The reason is that FC inherently puts extra load on combiner threads, leaving them with less time to perform their own work. Due to space constraints, the experiment in fully described in Appendix B.3.

4.2 Concurrent stack

Here we use our algorithm as the elimination layer on top of a Treiber-style nonblocking stack [18] and compare it to the stack implementations evaluated in [7]. We apply our algorithm to two variants of an elimination stack. In the first version, following [9], rendezvous is used as a *backoff* mechanism that threads turn to upon detecting contention on the main stack. This idea is to provide both good performance under low contention and scaling as contention increases. In the second version a thread visits the rendezvous structure first, accessing the main stack only if it fails to find a partner. (If it then encounters contention on the main stack it goes back to try the rendezvous, and so on.)

We compare the C++ implementation of our algorithm to the C++ implementations evaluated in [7]: a lock-free stack (LF-Stack), a lock-free stack with a simple elimination layer (EL-Stack), and an FC based stack. We use the same benchmark as [7], measuring the throughput of an even `push`/`pop` operation mix. Unlike the pool tests, here we want threads to give up if they don't find a partner in a short amount of time, and move to the main stack. We thus use a lower value for the wait threshold (shorter than the timeout), and compensate for it by using a smaller threshold ratio. Figure 5 shows the results. Our second (non-backoff) stack scales well, outperforming the FC and elimination stacks almost by a factor of three. The price it pays is poor performance at low concurrency (2.5× slower than the FC stack with a single thread). The backoff variant fails to scale above 32 threads due to contention on the stack head, illustrating the cost incurred by merely trying (and failing) to CAS a central hotspot.

5 Conclusions

We have presented a highly scalable, high throughput, nonblocking asymmetric rendezvous system that outperforms the most efficient prior synchronous queue, the parallel FC pool. While flat combining has a clear advantage in data structures that are sequential in nature, such as a FIFO queue and priority queue, whose concurrent implementations have single hot spots, FC may lose its advantage in data structures with inherent potential for parallelism. It is therefore interesting whether the FC technique can be improved to match the performance of the asymmetric rendezvous machine.

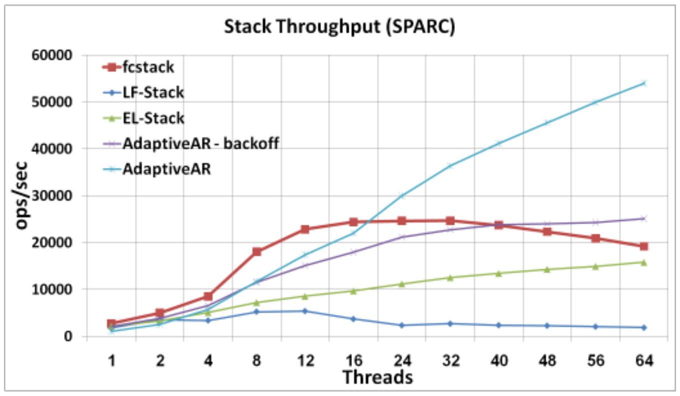


Figure 5: Comparing different stack implementations throughput. Each of N threads performs both `push` and `pop` operations with probability 1/2 for each operation type.

Acknowledgments: We thank Hillel Avni, Nati Linial and Nir Shavit for very helpful discussions.

References

[1] OpenSPARC T2 Core Microarchitecture Specification. `http://www.opensparc.net/pubs/t2/docs/OpenSPARCT2_Core_Micro_Arch.pdf`, 2007.

[2] Intel Microarchitecture, Codenamed Nehalem. `http://www.intel.com/technology/architecture-silicon/next-gen/`, 2009.

[3] Yehuda Afek, Guy Korland, Maria Natanzon, and Nir Shavit. Scalable producer-consumer pools based on elimination-diffraction trees. In Pasqua DAmbra, Mario Guarracino, and Domenico Talia, editors, *Euro-Par 2010 - Parallel Processing*, volume 6272 of *Lecture Notes in Computer Science*, pages 151–162. Springer Berlin / Heidelberg, 2010.

[4] Gregory R. Andrews. *Concurrent programming: principles and practice*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1991.

[5] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: a scalable memory allocator for multithreaded applications. *SIGARCH Computer Architecture News*, 28(5):117–128, 2000.

[6] David R. Hanson. *C interfaces and implementations: techniques for creating reusable software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[7] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA 2010, pages 355–364, New York, NY, USA, 2010. ACM.

[8] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Scalable flat-combining based synchronous queues. In *Proceedings of the 24th International Conference on Distributed Computing*, DISC 2010, pages 79–93, Berlin, Heidelberg, 2010. Springer-Verlag.

[9] Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA 2004, pages 206–215, New York, NY, USA, 2004. ACM.

[10] Maurice Herlihy. Wait-free synchronization. *ACM Transactions Transactions on Programming Languages and Systems (TOPLAS)*, 13:124–149, January 1991.

[11] William N. Scherer III, Doug Lea, and Michael L. Scott. A scalable elimination-based exchange channel. In *Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL 2005)*, Oct 2005.

[12] Doug Lea, William N. Scherer III, and Michael L. Scott. java.util.concurrent.exchanger source code. `http://gee.cs.oswego.edu/cgi-bin/viewcvs.cgi/jsr166/src/main/java/util/concurrent/Exchanger.java`, 2011.

[13] Mark Moir, Daniel Nussbaum, Ori Shalev, and Nir Shavit. Using elimination to implement scalable and lock-free fifo queues. In *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA 2005, pages 253–262, New York, NY, USA, 2005. ACM.

[14] William N. Scherer, III, Doug Lea, and Michael L. Scott. Scalable synchronous queues. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2006, pages 147–156, New York, NY, USA, 2006. ACM.

[15] Nir Shavit and Dan Touitou. Elimination trees and the construction of pools and stacks: preliminary version. In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA 1995, pages 54–63, New York, NY, USA, 1995. ACM.

[16] Nir Shavit and Asaph Zemach. Diffracting trees. *ACM Transactions on Computer Systems (TOCS)*, 14:385–428, November 1996.

[17] Nir Shavit and Asaph Zemach. Combining funnels: A dynamic approach to software combining. *Journal of Parallel and Distributed Computing*, 60(11):1355–1387, 2000.

[18] R. Kent Treiber. Systems programming: Coping with parallelism. Technical Report RJ5118, IBM Almaden Research Center, 2006.

A Related work

Synchronous queues A synchronous queue using three semaphores was described by Hanson [6]. Java 5 includes a coarse-grained locking synchronous queue [14], which was superseded in Java 6 by Scherer, Lea and Scott's algorithm. Their algorithm is based on a Treiber-style nonblocking stack [18] whose top becomes a sequential bottleneck as the level of concurrency increases. Motivated by this, Afek et. al. described *elimination-diffracting (ED) trees* [3], a randomized distributed data structures where arriving threads follow a path through a binary tree whose internal nodes are *balancer* objects [16] and the leaves are Java synchronous queues. Recently, Hendler et. al. applied the *flat combining* paradigm to the synchronous queue problem, describing a single combiner version and a *parallel* version, in which multiple combiners each handle a subset of participating threads and then try to satisfy unmatched requests in their subset by entering an *exchange* FC synchronous queue.

Elimination The elimination technique is due to Touitou and Shavit [15]. Hendler, Shavit and Yerushalmi used elimination with an adaptive scheme inspired by Shavit and Zemach [17] to obtain a scalable linearizable stack [9]. In their scheme threads adapt locally: each thread picks a slot to collide in from sub-range of the collision layer centered around the middle of the array. If no partner arrives, the thread eventually shrinks the range. Alternatively, if the thread sees a waiting partner but fails to collide due to contention, it increases the range. In our adaptivity technique, described in Section 3, threads also make local decisions, but with global impact: the ring is resized. Moir et. al. used elimination to scale a FIFO queue [13]. In their algorithm an enqueuer picks a random slot in an elimination array and waits there for a dequeuer; a dequeuer picks a random slot, giving up immediately if that slot is empty. It does not seek out waiting enqueuers. Moir et. al. also did not discuss techniques to adapt their elimination layer to load. Scherer, Lea and Scott applied elimination in their *symmetric* rendezvous system [11], where there is only one type of a thread and so the pairing is between any two threads that show up. Scherer, Lea and Scott also do not discuss adaptivity, though a later version of their symmetric *exchanger channel*, which is part of Java [12], includes a scheme that resizes the elimination array. However, we are interested in the more difficult *asymmetric* rendezvous problem, where not all pairings are allowed.

B Evaluation omitted from main body

B.1 Omitted graphs of SPARC machine

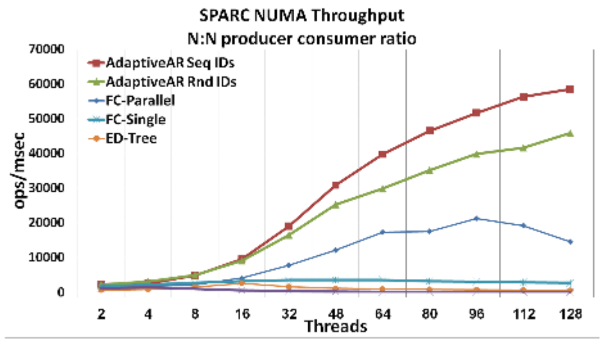


Figure 6: Rendezvousing between N pairs of producers and consumers when using both processors of the Sun machine (i.e., NUMA configuration). Default OS policy is to place threads chips round-robin.

Figure 7

B.2 Changing arrival rates experiment

In practice, we expect threads to do some work between rendezvouses. We therefore measure how the throughput of the 32 prodcuer/consumer pairs workload is affected when the thread arrival rate decreases due to increasingly larger amounts of time spent doing "work" before each rendezvous. Figure 9 shows that as the work period grows and the available parallelism in the data structure decreases, the throughput of all algorithms that exhibit scaling deteriorates: the parallel FC degrades by 2× when going from no work to 1.5μs of work, and our algorithm degrades by 4×. Still, there is sufficient parallelism to allow our algorithm to outperforms the other implementations by

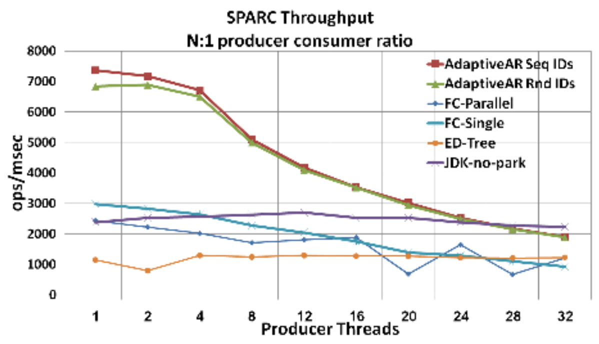


Figure 8: Synchronous queue throughput with N producers and a single consumer.

a factor of at least three.

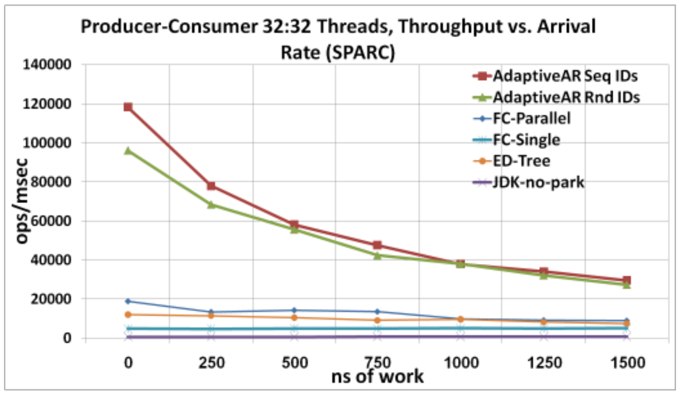


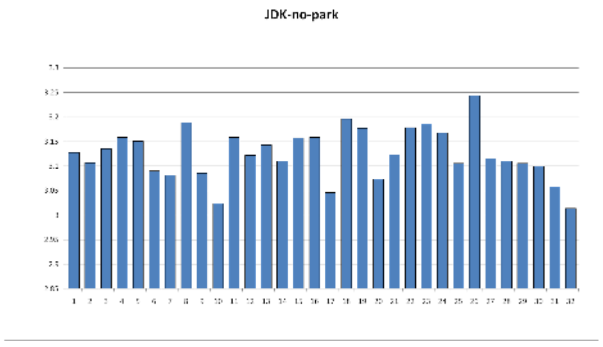
Figure 9: Synchronous queue 32 producer/consumer pairs throughput with decreasing arrival rate due to increasing amount of time working between rendezvouses.

B.3 Work uniformity

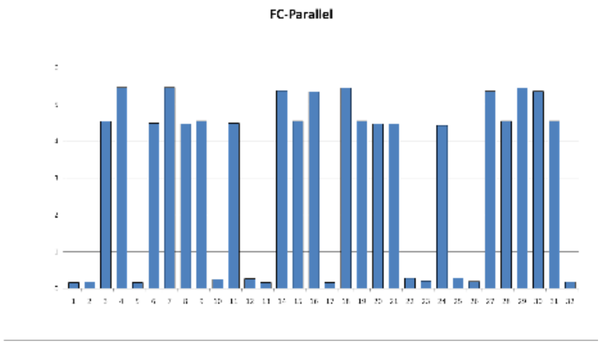
Here we evaluate how *uniformly* do threads operate in the multiple producer/consumer benchmark. We pick the best result from five executions of 16 producer/consumer pairs, and plot the percent of total operations performed by each thread. Figure 10 shows the results. In an ideal setting, each thread would perform 3.125% (1/32) of the work. Our algorithm comes relatively close to this distribution. The distribution of work has standard deviation of 0.47% for random thread ids and 0.10% for sequential ids, and the ratio between the best/worst performers is 1.60 for random ids and 1.17 for sequential ids. The JDK algorithm is also uniform, with a standard deviation of 0.05% and best/worse ratio of 1.07. In contrast, the parallel FC pool has a standard deviation of 2.32% and best/worst ratio of 33.44. The reason is that FC inherently puts extra load on combiner threads, leaving them with less time to perform their own work.

B.4 Impact of peeking

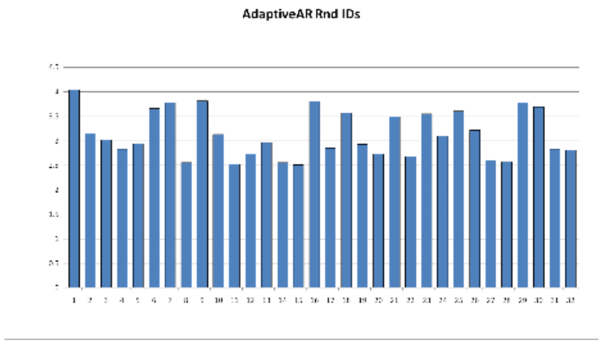
We evaluate the algorithm's peeking technique by measuring what impact does disabling this technique have on throughput. Figure depicts 11a the results for both random and sequential thread ids. While at low thread counts (up to four) peeking has little effect, once concurrency increases



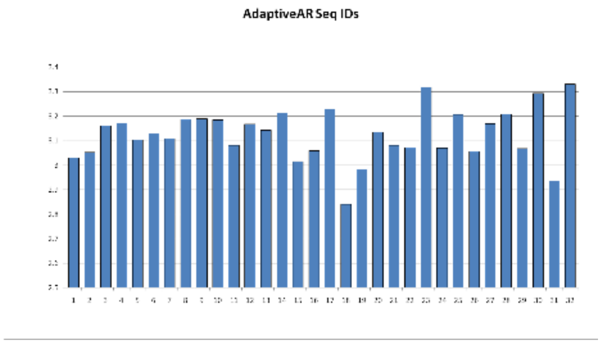
(a)



(b)



(c)



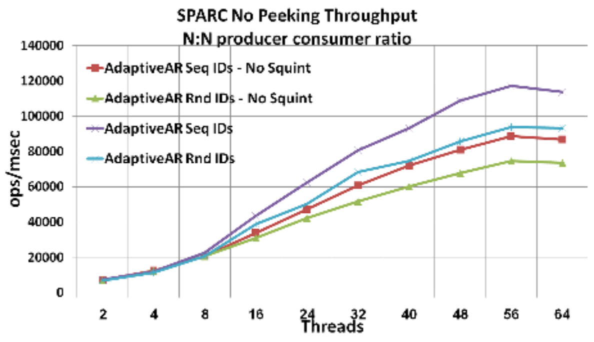
(d)

Figure 10: Work uniformity: Percent of total operations performed by each of 32 threads in a producer/consumer benchmark.

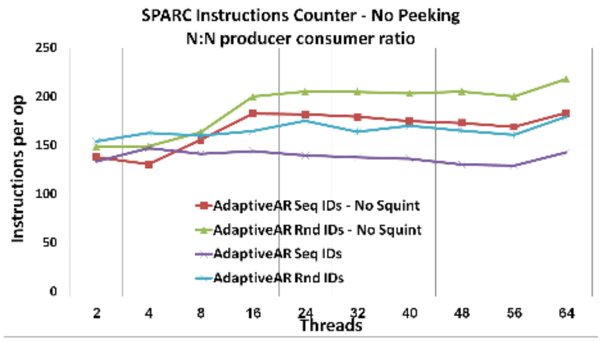
peeking improves performance by as much as 25%. Since the algorithm's ring size adapts to concurrency, a producer's initial node being empty usually means that a consumer will arrive there shortly. Peeking therefore leads to a natural pairing between producers and consumers. Without it, as Figures 11b and 11c show, a producer has a higher chance of colliding with another producer and consequently requires more instruction to complete a rendezvous.

B.5 Results from Intel Nehalem machine

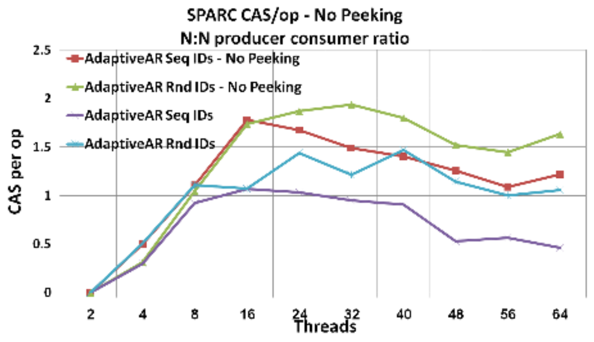
Figure 12 shows the results collected on the Intel Nehalem machine for the following benchmarks: symmetric multiple producers/multiple consumer benchmark (Figure 12a), single producer and multiple consumers (Figure 12b), multiple producers and single consumer (Figure 12c), and throughput in the face of changing arrival rates (Figure 12d). As in [8], we used different "work periods" on the Intel machine due to its faster processor.



(a) Throughput.

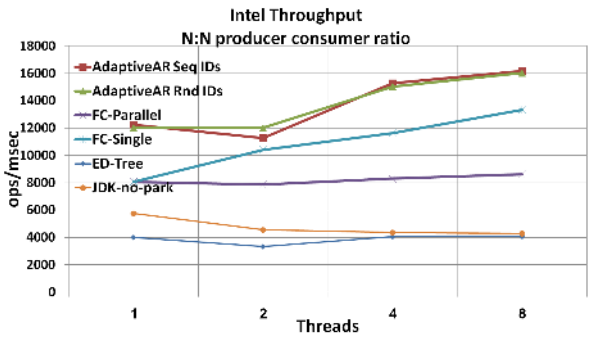


(b) Instructions (per *put*).

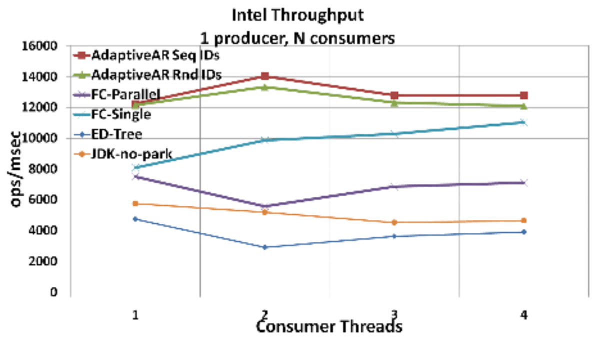


(c) CASes (per *put*).

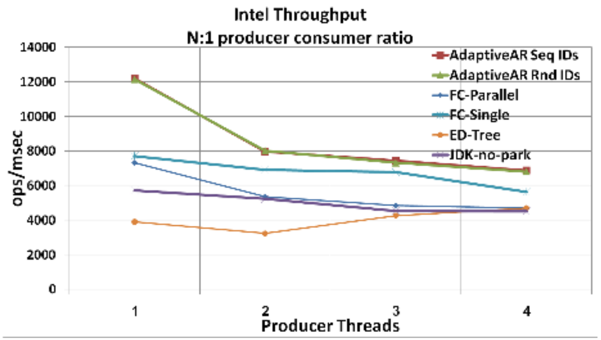
Figure 11: Rendezvousing between N pairs of producers and consumers with and without peeking. Graphs show rate of completed operations per millisecond and hardware performance counter data for *producer* operations.



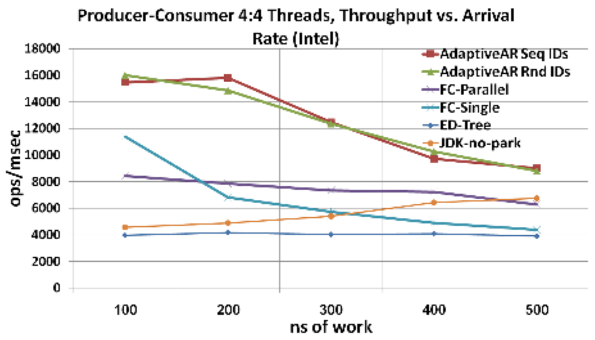
(a) N pairs of producers and consumers.



(b) Synchronous queue throughput with a single producer and N consumers.



(c) Synchronous queue throughput with N producers and a single consumer.



(d) Synchronous queue four producer/consumer pairs throughput with decreasing arrival rate.

Figure 12: Benchmark results on Intel Nehalem machine with 4 cores, each multiplexing 2 hardware threads.