Distrib. Comput. (2013) 26:243–269
DOI 10.1007/s00446-013-0185-0

# Fast and scalable rendezvousing

**Yehuda Afek · Michael Hakimi · Adam Morrison**

Received: 26 December 2011 / Accepted: 23 November 2012 / Published online: 28 March 2013


**Abstract** In an asymmetric rendezvous system, such as an unfair synchronous queue or 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 new highly scalable, high throughput asymmetric rendezvous systems that outperform prior synchronous queue and elimination array implementations under both symmetric and asymmetric workloads (more operations of one type than the other). Based on this rendezvous system, we also construct a highly scalable and competitive stack implementation.

## 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. Asymmetric rendezvous is exactly the task performed by *unfair synchronous queues* (or *synchronous channels*) [17], in which producers and consumers handshake to exchange data. Synchronous queues are a key building block in Java's thread pool implementation and other message-passing and hand-off designs [2,17]. The asymmetric rendezvous abstraction also encompasses the *elimination* technique [20], which is used to scale concurrent stacks and queues [9,16].

In this paper we present two highly scalable asymmetric rendezvous algorithms that improve the state of the art in both unfair synchronous queue and elimination algorithms. Our starting point is the *elimination technique* [1,9,20] where an arriving thread picks a random slot in a *collision array*, hoping to meet the right kind of partner; however, if the slot picked is empty the thread waits for a partner to arrive. This standard elimination approach is vulnerable to *false matches*, when two threads of the same type meet, and to *timeouts*, when a producer and a consumer each pick a different slot and futilely wait for a partner to arrive. Consequently, elimination arrays provide no progress guarantee and are unsuitable for use in a synchronous queue implementation.

Our first algorithm, named AdaptiveAR (Adaptive Asymmetric Rendezvous), 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 a shared ring structure and waits there for a producer, a producer actively seeks out waiting consumers on the ring. As a result, AdaptiveAR does not suffer from false matches and timeouts and we show that it is *nonblocking* in the following sense: if both producers and consumers keep taking steps, *some* rendezvous operation is guaranteed to complete. (This progress property, which we refer to as *pairwise nonblocking*, is similar to *lock-freedom* [10] while taking into account the fact that "it takes two to tango", i.e., both types of threads must take steps to guarantee a successful rendezvous.) We present two new techniques, that combined with this asymmetric

This work was supported by the Israel Science Foundation under grant 1386/11 and by machine donations from Intel and Oracle. Adam Morrison is supported by an IBM PhD Fellowship.

Y. Afek (✉) · M. Hakimi · A. Morrison
Blavatnik School of Computer Science, Tel Aviv University, Tel Aviv, Israel
e-mail: afek@tau.ac.il

M. Hakimi
e-mail: mi5@tau.ac.il

A. Morrison
e-mail: adamx@tau.ac.il

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 minimizing as much as possible 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, there is a good chance that a consumer will arrive there shortly. However, simply waiting at this node, hoping that a consumer arrives, would make the algorithm prone to timeouts and impede progress. Rather, we employ 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 second algorithm extends AdaptiveAR with two new forms of adaptivity. Together with AdaptiveAR's original ring adaptivity scheme, this algorithm is triple adaptive and hence named TripleAdp. We present TripleAdp separately from AdaptiveAR because the adaptivity techniques it employs add some overhead over AdaptiveAR.

The major new dimension of adaptivity in TripleAdp is that it adapts the *roles performed by the threads*. AdaptiveAR fixes the roles played by consumers and producers in the algorithm: consumers wait for producers which seek them on the ring. Unfortunately, in asymmetric workloads, with more producers than consumers, the producers collide with each other when competing for the few consumers in the ring. As a result, in such workloads AdaptiveAR's throughput degrades as the number of producers increases. However, exactly this property makes AdaptiveAR shine in workloads where consumers outnumber producers: the few producers have their pick among the many consumers occupying the ring, and AdaptiveAR maintains the peak throughput achievable by the producers. The main observation behind TripleAdp is that the *mirror image* of AdaptiveAR, where consumers seek producers that wait on the ring, will do well in the asymmetric workloads where AdaptiveAR fails to maintain peak throughput. TripleAdp therefore adapts to the access pattern and switches between (essentially) AdaptiveAR and its mirror image, enjoying the best of both worlds.

The second dimension of adaptivity added in TripleAdp is that of *overall memory consumption*. For simplicity, we designed AdaptiveAR to use a predetermined static ring size and only adapt its *effective* size, i.e., the subset of the ring actually used by the currently active threads. However, this requires advance knowledge of the maximum number of threads that can simultaneously run, which may not be known a priori. TripleAdp addresses this limitation with a technique to adapt the *total* ring size by allocating and deallocating ring nodes as necessary.

Both AdaptiveAR and TripleAdp have several features that no prior synchronous queue algorithm possesses together. They are based on a distributed scalable ring structure, unlike Java's synchronous queue which relies on a non-scalable centralized structure. They are nonblocking and 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 algorithms perform extremely well in practice on a number of hardware architectures with different characteristics. On an UltraSPARC T2 Plus multicore machine with 64 hardware threads and a write-through cache architecture AdaptiveAR outperforms Java's synchronous queue by up to 120×, the FC synchronous queue by up to 5.5×, and the Java Exchanger algorithm [12,18] (an elimination-array implementation) by up to 2.5×. On an Intel Xeon E7 multicore with 20 hardware threads and a write-back cache architecture our algorithms outperform or match the performance of prior synchronous queues in most benchmarks.

Finally, we adapt our rendezvousing algorithms for use as an elimination layer on top of Treiber's lock-free stack [24], yielding a highly scalable stack that outperforms existing concurrent stack implementations. Here `push` operations act as producers and `pop` operations as consumers. A pair of operations that successfully rendezvous can be linearized together without having to access the main stack [9].

In addition to using elimination as a *backoff* scheme that threads enter after sensing contention on the main stack, as was done in prior work [9], we present an *optimistic* elimination stack. Here a thread enters the elimination layer *first*, accessing the main stack only if it fails to find a match. If it then encounters contention on the main stack it goes back to try the rendezvous, and so on.

Our AdaptiveAR optimistic elimination stack yields more than a factor of three improvement over the prior elimination-array stack [9] and FC stack [7] algorithms on the UltraSPARC T2 Plus architecture.

*Outline:* We review related work in Sect. 2. Section 3 provides the computational model and formal definitions of the asymmetric rendezvous problem and our progress property. Section 4 describes AdaptiveAR and Sect. 5 describes TripleAdp. Both algorithms are empirically evaluated together in Sect. 6. We describe and evaluate our rendezvous-based elimination stacks in Sect. 7. We conclude in Sect. 8.

## 2 Related work

### 2.1 Synchronous queues

A synchronous queue using three semaphores was described by Hanson [6]. The Java 5 library improved on this using

a coarse-grained locking synchronous queue, which was superseded in Java 6 by Scherer, Lea and Scott's algorithm [17]. Their algorithm is based on a Treiber-style nonblocking stack [24] 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. Motivated by this, Afek, Korland, Natanzon, and Shavit described *elimination-diffracting (ED) trees* [1], a randomized distributed data structure where arriving threads follow a path through a binary tree whose internal nodes are *balancer* objects [21] and the leaves are Java synchronous queues. In each internal node a thread accesses an elimination array in an attempt to avoid descending down the tree. Recently, Hendler et al. applied the *flat combining* paradigm of [7] to the synchronous queue problem [8], describing single combiner and parallel versions. In a Single FC channel a thread attempts to become a *combiner* by acquiring a global lock. Threads that fail to grab the lock instead post their request and wait for it to be fulfilled by the combiner. The combiner continuously scans the pending requests. As it does so it keeps a private stack which contains pending operations of one kind. If it reads a request that can be matched to the top operation in the stack, it pops that operation and releases the two matched threads. Otherwise, it pushes the request on the stack and continues. In the Parallel FC version there are multiple combiners, each handling a subset of participating threads. Each combiner posts leftover operations that remain in its subset to an *exchange* Single FC synchronous channel, in an attempt to match them with operations from other combiners' subsets.

### 2.2 Concurrent stacks

Several works exploit the *elimination* idea to obtain a scalable stack implementation. Originally, Touitou and Shavit [20] observed that concurrent `push` and `pop` stack operations can be *eliminated* by having the `push` pass its item directly to the `pop`. Their stack algorithm is not linearizable [11] as it is based on diffracting trees [21]. Shavit and Zemach described a linearizable stack using the elimination concept, but their algorithm is blocking [22].

Hendler, Shavit and Yerushalmi [9] applied elimination as a *backoff scheme* on top of Treiber's classic lock-free stack algorithm [24] to obtain a scalable, lock-free, linearizable stack. They used a collision array in which each thread picks a slot trying to collide with a partner. Thus, the collision array is (implicitly) a form of rendezvous.

In Hendler, Shavit and Yerushalmi's adaptive scheme threads adapt locally: each thread picks a slot to collide in from sub-range of the collision array 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 Sect. 4, threads also make local decisions, but with global impact: the ring is resized.

Several concurrent stacks have been shown in works on *combining*-based constructions. The idea in combining is to have one thread perform the combined work of other threads, saving them from accessing the main structure. Hendler et al. implemented a stack using the flat combining approach [7]. Fatourou and Kallimanis recently presented a combining technique that provides bounds on the number of remote memory references performed by waiting threads, and demonstrated a stack implementation using that technique [5]. Both FC and Fatourou and Kallimanis' techniques are blocking, whereas we are interested in nonblocking algorithms.

Fatourou and Kallimanis have also described an efficient *wait-free* universal construction [4], and showed that a concurrent stack implemented using this construction performs as well as the (blocking) FC based stack.

### 2.3 Other applications of elimination

Moir et al. used elimination to scale a FIFO queue [16]. 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. Scherer, Lea and Scott applied elimination in their *symmetric exchange channel* [18], 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 an improved version of their algorithm that is part of the Java concurrency library [12] includes a scheme that resizes the elimination array. However, here we are interested in the more difficult *asymmetric* rendezvous problem, where not all pairings are allowed.

### 2.4 Rendezvous semantics

Scherer and Scott proposed to model concurrent operations such as rendezvousing, that must wait for another thread to establish a precondition, by splitting them into two: a *reservation* (essentially, announcing that the thread is waiting) and a subsequent *followup* at which the operation takes effect after its request is fulfilled [19]. In contrast, our model does not require splitting an operation into two. In fact, in our algorithms some operations do not have a clear reservation point. Thus, we demonstrate that asymmetric rendezvous can be both reasoned about and implemented efficiently without using the reservation/followup semantics.

## 3 Preliminaries

### 3.1 Model

We use a standard shared memory system model with a set of sequential asynchronous *threads* that communicate via shared memory.

The memory locations in our model provide atomic `read`, `write` and `compareAndSet` (`CAS`) operations. Formally, each memory cell $m$ has a state $v$ and supports the following operations: (1) a `read` operation that returns $v$, (2) a `write(`$x$`)` operation which changes the state to $x$ and returns `OK`, (3) a `CAS(`$o$, $n$`)` operation whose effect depends on $v$: if $v = o$ it changes the state to $n$ and returns `TRUE`; otherwise, it returns `FALSE`.

Using the memory operations the threads implement a high-level *object* defined by a *sequential specification*, a state machine consisting of a set of *states*, a set of *operations* used to transition between the states, and *transitions* between the states. A transition $T(s, op) = (s', r)$ specifies that if $op$ is applied at state $s$, the object state changes to $s'$ and $op$ returns $r$. An operation $op$ is *total* if a transition $T(s, op)$ is defined for every state $s$. Otherwise $op$ is *partial* [11]. An *implementation* is an algorithm specifying the shared memory operations each thread has to perform in order to complete an operation on the high-level object. We consider only linearizable [11] implementations.

The computation is modeled as a sequence of events. A *high-level event* consists of a thread invoking or returning from a high-level operation. A *primitive event* is a computation step in which a thread invokes a memory operation, receives a response and changes its state according to the algorithm, all in one step. An *execution* is a (possibly infinite) sequence of events. We consider only *well formed* executions, in which the subsequence of thread $t$'s events consists of (1) zero or more instances of a high-level operation invocation followed by primitive events and finally a high-level response, (2) a possible final *pending* operation that consists of a high-level invocation without a matching high-level response.

Thread $t$ is *active* in an execution $E$ if $E$ contains an event by $t$. Two high-level operations are *concurrent* in an execution if one of them starts before the other one returns.

We consider two variants of the above model. In one (Sect. 4) the number of threads in the system is known by the algorithm to be a constant $T$. In the second (Sect. 5) the maximum number of threads active in an execution remains finite, but is not known in advance and can differ between executions. (This is essentially the *bounded concurrency* model [13]).

#### 3.1.1 Stack object

A *stack* is an object supporting the operations `push()` and `pop()`. Its state is a sequence of $m \geq 0$ items $\alpha = x_m, \ldots, x_1$ where $x_m$ is called the *top* of the stack. A stack whose state is the empty sequence, $\epsilon$, is said to be *empty*. A `push(`$y$`)` operation on state $\alpha$ makes $y$ the top of the stack by changing the state to $y\ \alpha$. A `pop()` operation on a non-empty state $y\ \alpha$ returns $y$, the top of the stack, and changes the state to $\alpha$. A `pop()` operation on an empty stack returns a reserved value $\perp$.

### 3.2 Asymmetric rendezvous

Informally, in the *asymmetric rendezvous* problem there are threads of two types, *producers* and *consumers*. Producers perform `put(`$x$`)` operations which return `OK`. Consumers perform `get()` operations that return some item $x$ passed by a producer. Producers and consumers show up and must be matched with a unique thread of the other type, such that a `put(`$x$`)` and the `get()` that returns $x$ must be *concurrent*.

A rendezvous operation cannot always complete: for example, a thread running alone cannot rendezvous. We formally capture this issue by modeling the rendezvous problem using a sequential specification of an object that supports *partial* operations [11] which are undefined for some states. An implementation of a partial operation requires having the operation wait until the object reaches a state in which the operation is defined. We now define an object with partial operations whose specification captures the handoff semantics of rendezvous, in which producers pass data to consumers.

**Definition 3.1** A *handoff* object consists of a single variable, $v$, that initially contains a reserved value $\perp$. It supports two partial operations, `put(`$x$`)` and `get()`. A `put(`$x$`)` operation is defined only when $v = \perp$; it sets $v$ to $x$ and returns `OK`. A `get()` operation is defined only when $v = x$ for any $x \neq \perp$; it stores $\perp$ back in $v$ and returns $x$. For simplicity, we assume each `put(`$x$`)` hands off a unique value.

Even with partial operations a sequential specification alone cannot capture the synchronous nature of rendezvous, i.e., that threads need to wait for each other. To see this, notice that there must be an operation that is defined in the initial state and such an operation can therefore complete alone. The following definition addresses this limitation by reasoning directly about the concurrent rendezvous implementation.

**Definition 3.2** A concurrent algorithm **A** *implements asymmetric rendezvous* if the following hold:

– Linearizability: **A** is a linearizable implementation of the handoff object.
– Synchrony: For any execution $E$ of **A**, and any `put(`$x$`)` operation that completes in $E$, $E$ can be extended (adding zero or more events) into an execution $E'$ which contains a `get()` operation that returns $x$ and is concurrent to `put(`$x$`)`.

The idea behind these definitions is that when many producers and consumers show up, each matched producer/consumer pair can be linearized [11] as a `put(`$x$`)` followed by a `get()` returning $x$.

### 3.3 Pairwise nonblocking progress

To reason about progress for rendezvous we must take into account that rendezvous inherently requires waiting—a thread cannot complete until a partner shows up. We therefore consider the *joint* behavior of producers and consumers, using the following notion:

**Definition 3.3** Let $E$ be an execution of asymmetric rendezvous, $e \in E$ an execution step, and $e' \in E$ the step preceding $e$. We say that $e$ is a *pairwise step* if one of the following holds: (1) $e$ is a step by a `put()` operation and $e'$ is a step by a `get()` operation, (2) $e$ is a step by a `get()` operation and $e'$ is a step by a `put()` operation.

For example, if we denote an execution step of producer $p_i$ with $p_i$'s id, and analogously use $c_i$ for consumers, then in the following execution fragment the pairwise steps are exactly the steps marked in bold:

$$\ldots, p_1, p_5, p_1, \boldsymbol{c_3}, \boldsymbol{p_5}, \boldsymbol{c_4}, c_4, c_4, \boldsymbol{p_1}, \ldots.$$

We now define our notion of nonblocking progress, which requires that if both producers and consumers take enough steps, some rendezvous operation completes:

**Definition 3.4** An algorithm **A** implementing asymmetric rendezvous is *pairwise nonblocking* if, at any point in time, after a finite number of pairwise steps *some* operation completes.

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. Rather, we rule out implementations that make no progress at all, i.e., that admit executions in which both types of operations take steps infinitely often and yet no operation successfully completes.

## 4 AdaptiveAR

### 4.1 Algorithm description

The pseudo code of the AdaptiveAR algorithm is provided in Fig. 2. The main data structure (Fig. 2a) is a ring of nodes.

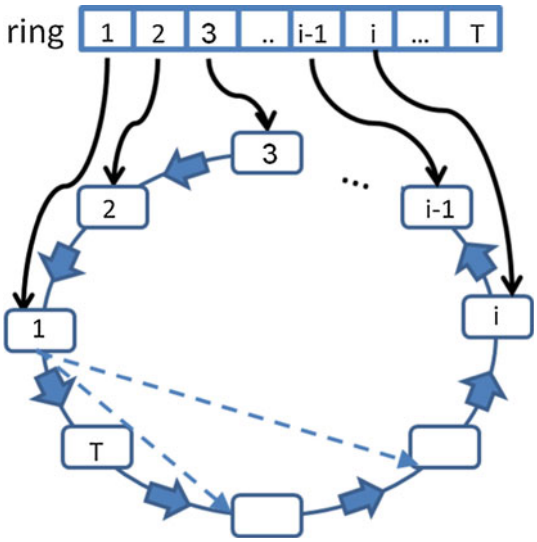


**Fig. 1** The ring data structure with the array of pointers. Node $i$ points to node $i-1$. The ring maximum size is $T$ and is resized by changing the head's `prev` pointer

The ring is accessed through a central array `ring`, where `ring[`$i$`]` points to the $i$th node in the ring (Fig. 1).[1] A consumer attempts to capture a node in the ring. It traverses the ring searching for an empty node, and once a node is captured, waits there for a producer. A producer scans the ring, seeking a waiting consumer. Once a consumer is found, it tries to match with it. For simplicity, in this section we assume the number of threads in the system, $T$, is known in advance and pre-allocate a ring of size $T$. However, the effective size of the ring is adaptive and it is dynamically adjusted as the concurrency level changes. (We expand on this in Sect. 4.3.) In Sect. 5 we handle the case in which the maximum number of threads that can show up is not known in advance; in this case the nodes are allocated and deallocated dynamically, i.e., the total ring size adapts to the number of threads.

Conceptually, each ring node contains a pointer that encodes the node's state:

1. **Free**: pointer 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:** pointer is `NULL`.
3. **Holding data (of a producer):** pointer points to the data.

In practice, ring traversal is more efficient by following a pointer from one node to the next rather than the alternative of traversing the array. Array traversal in Java suffers from two inefficiencies. First, in Java reading from the next array cell may result in a cache miss (it is an array of pointers),

[1] This reflects Java semantics, where arrays are of *references* to objects and not of objects themselves.

```
1  struct node {
2    item : pointer to object
3    index : node's index in the ring
4    prev : pointer to previous ring node
5  }
6
7  shared vars:
8    ring : array [1,...,T] of pointers to nodes,
9           ring [1]. prev = ring[T],
10          ring[i]. prev = ring[i−1] (i > 1)
11
12 utils:
13 getRingSize() {
14   node tail := ring [1]. prev
15   return tail.index
16 }
```

**(a)** Global variables

```
17 put(threadId, object item) {
18
19   node s := ring[hash(threadId,
20                       getRingSize())]
21   node v := s.prev
22
23   while (true) {
24     if (s.item == NULL) {
25       if (CAS(s.item, NULL, item))
26         return OK
27     } else if (v.item == NULL) {
28       if (CAS(v.item, NULL, item))
29         return OK
30       v := v.prev
31     }
32   }
33 }
```

**(b)** Producer code

```
34 get(threadId) {
35   int ringsize, busy_ctr, ctr := 0
36   node s, u
37
38   ringsize := getRingSize()
39   s := ring[hash(threadId, ringsize)]
40   (u,busy_ctr) := findFreeNode(s,ringsize)
41   while (u.item == NULL) {
42 ▷   ringsize := getRingSize()
43 ▷   if (u.index > ringsize and
44 ▷       CAS(u.item, NULL, FREE) {
45 ▷     s := ring[hash(threadId, ringsize)]
46 ▷     (u,busy_ctr) := findFreeNode(s, ringsize)
47 ▷   }
48 ▷   ctr := ctr + 1
49   }
50
51   item := u.item
52   u.item := FREE
53 ▷ if (busy_ctr < T_d and ctr > T_w and ringsize>1)
54 ▷   // Try to decrease ring
55 ▷   CAS(ring[1].prev, ring[ringsize], ring[ringsize−1])
56   return item
57 }
58
59 findFreeNode(node s, int ringsize) {
60 ▷ int busy_ctr := 0
61   while (true) {
62     if (s.item == FREE and
63         CAS(s.item, FREE, NULL))
64       return (s,busy_ctr)
65     s := s.prev
66 ▷   busy_ctr := busy_ctr + 1
67 ▷   if (busy_ctr > T_i·ringsize) {
68 ▷     if (ringsize < T) // Try to increase ring
69 ▷       CAS(ring[1].prev, ring[ringsize], ring[ringsize+1])
70 ▷     ringsize := getRingSize()
71 ▷     busy_ctr := 0
72 ▷   }
73   }
74 }
```

**(c)** Consumer code

**Fig. 2** Asymmetric rendezvous algorithm. Lines beginning with ▷ handle the adaptivity process

whereas reading from the (just accessed) current node does not. Second, maintaining a running array index requires an expensive test+branch to handle index boundary conditions or counting modulo the ring size, while reading a pointer is cheap. The pointer field is named `prev`, reflecting that node $i$ points to node $i-1$. This allows the ring to be *resized* with a single atomic `CAS` that changes `ring[1]`'s (the head's) `prev` pointer. To support mapping from a node to its index in the array, each node holds its index in a read-only `index` field.

For the sake of clarity we start in Sect. 4.2 by discussing the non-adaptive ring size algorithm. The adaptivity code, however, is included in the pseudo code, marked by a ▷ symbol. It is explained in Sect. 4.3. Finally, Sect. 4.4 discusses correctness and progress.

## 4.2 Nonadaptive algorithm

*Producers (Fig. 2b)*: A producer searches the ring for a waiting consumer, and attempts to pass its data to it. The search begins at a node, $s$, obtained by hashing the thread's id (Line 18). The producer passes the ring size to the hash function as a parameter, to ensure the returned node falls within the ring. It then traverses the ring looking for a node captured by a consumer. Here the producer periodically *peeks* at the initial node $s$ to see if it has a waiting consumer (Lines 23–25); if not, it checks the current node in the traversal (Lines 26–29). Once a captured node is found, the producer tries to deposit its data using a `CAS` (Lines 24 and 27). If successful, it returns.

*Consumers (Fig. 2c)*: A consumer searches the ring for a free node and attempts to capture it by atomically changing its `item` pointer from `FREE` to `NULL` using a `CAS`. Once a node is captured, the consumer spins, waiting for a producer to arrive and deposit its item. Similarly to the producer, a consumer hashes its id to obtain a starting point, $s$, for its search (Line 37). The consumer calls the `findFreeNode` procedure to traverse the ring from $s$ until it captures and returns node $u$ (Lines 57–72). (Recall that the code responsible for handling adaptivity, which is marked by a ▷, is ignored

for the moment). The consumer then waits until a producer deposits an item in $u$ (Lines 39–47), frees $u$ (Lines 49–50) and returns (Line 54).

### 4.3 Adding adaptivity

If the number of active consumers is smaller than the ring size, producers may need to traverse through a large number of empty nodes before finding a match. It is therefore important to *decrease* the ring size if the concurrency level is low. On the other hand, if there are more concurrent threads than the ring size (high contention), it is important to dynamically *increase* the ring. The goal of the adaptivity scheme is to keep the ring size "just right" and allow threads to complete their operations with a small number of steps.

The logic driving the resizing process is in the consumer's code, which detects when the ring is overcrowded or sparsely populated and changes the size accordingly. If a consumer fails to capture a node after passing through many nodes in `findFreeNode()` (due to not finding a free node or having its `CAS`es fail), then the ring is too crowded and should be increased. The exact threshold is determined by an *increase threshold* parameter, $T_i$ ($0 < T_i \leq 1$). If in `findFreeNode()`, a consumer fails to capture a node after passing through more than $T_i \cdot$ `ring_size` nodes, it attempts to increase the ring size (Lines 65–70). To detect when to decrease the ring, we observe that when the ring is sparsely populated, a consumer usually finds a free node quickly, but then has to wait longer until a producer *finds it*. Thus, we add a *wait threshold*, $T_w$, and a *decrease threshold*, $T_d$. If it takes a consumer more than $T_w$ iterations of the loop in Lines 39–47 to successfully complete the rendezvous, but it successfully captured its ring node in up to $T_d$ steps, then it attempts to decrease the ring size (Lines 51–53).

Resizing the ring is made by `CAS`ing the `prev` pointer of the ring head (`ring[1]`) from the current tail of the ring to the tail's successor (to increase the ring size by one) or its predecessor (to decrease the ring size by one). If the `CAS` fails, then another thread has resized the ring and the consumer continues. The head's `prev` pointer is not a sequential bottleneck because resizing is a rare event in stable workloads. Even if resizing is frequent, the thresholds ensure that the cost of the `CAS` is negligible compared to the other work performed by the algorithm, and resizing pays-off in terms of better ring size which leads to improved performance.

Notice that if some consumer is waiting at the tail of the ring before the ring size decreases, this consumer will be left outside the new ring and so new arriving producers will not find it, which may impact progress. We next describe how to solve this problem.

*Handling consumers left out of the ring:* Each consumer periodically checks if its node's index is larger than the current ring size (Line 41). If so, it tries to free its node using a `CAS` (Line 42) and find itself a new node in the ring (Lines 43–44). However, if the `CAS` fails, then a producer has already deposited its data in this node and so the consumer can take the data and return (this will be detected in the next execution of Line 39).

### 4.4 Correctness proofs

**Theorem 4.1** *AdaptiveAR implements asymmetric rendezvous.*

*Proof* We need to show: (1) linearizability (that AdaptiveAR is a linearizable implementation of the handoff object) and (2) synchrony (that a `get()` returning $x$ executes concurrently with `put(`$x$`)`).

*Synchrony:* Let $p$ be some `put(`$x$`)` operation which completes. Then $p$ reads `NULL` from $s$.`item`, for some node $s$ in the ring (Line 23 or 26), and deposits $x$ in $s$.`item` using `CAS` (Line 24 or 27). Thus, there exists a consumer $c$ who changed $s$.`item` from `FREE` to `NULL` (Line 61) and is therefore executing the loop in Lines 39-47 when $p$ `CAS`es $x$ to $s$.`item`.

We prove that if $c$ continues to take steps after $p$'s `CAS`, it returns $x$. When $c$ executes Line 39, it notices $x$ written to $s$.`item` and returns $x$ (Lines 49–54). Notice that even if $c$ attempts to free its node due to a ring resize (Line 42), its `CAS` fails since $s$.`item` contains $x$ and thus $c$ proceeds to execute Line 39.

*Linearizability:* We assign linearization points as follows. Let $p$ be a `put(`$x$`)` that completes its operation. Then $p$ is linearized at its successful `CAS` of $x$ into $s$.`item` for some node $s$ (Line 24 or 27). As shown above, there exists a unique concurrent `get()` operation, $c$, that captured $s$ before $p$'s linearization point and which returns $x$ if it takes enough steps. We linearize $c$ returning $x$ immediately after $p$, at the same `CAS` event as $p$. Conversely, let $c$ be a `get()` operation which completes its operation. Then $c$ captures a node $s$ and then spins until observing some item $x$ stored in $s$.`item` (Line 39). This occurs as a result of some producer `CAS`ing $x$ into $s$.`item`, and so it follows that in the process described above of assigning linearization points to `put(`$x$`)`/`get()` pairs, we assign a linearization point to $c$. Thus, any operation that completes has been assigned a linearization point.

Because we always linearize a `put()` followed immediately by its matching `get()` at the `put()`'s successful `CAS` operation, our linearization points assignment yields a sequence of matching `put()`/`get()` pairs. It remains to show that this sequence is a valid execution of the handoff object. To do this, we show using induction on the number of `put()`/`get()` pairs linearized that our linearization point assignment constructs a valid execution of the handoff object that leaves the object in state $\perp$. In the base case no operation is linearized, and this is the (valid) empty execution.

For the induction step, we have a valid execution of the handoff object that ends with the object in state $\perp$. In each `put()`/`get()` linearized, we linearize the `put()` first. Thus the next operation linearized is a `put(`$x$`)`, which is valid in state $\perp$, and changes the object's state to $x$. We next linearize a `get()` returning $x$, which is valid and changes the object's state to $\perp$. □

**Theorem 4.2** *AdaptiveAR is pairwise nonblocking.*

*Proof* We say that the rendezvous of a `put()` and `get()` pair *occurs* at the linearization point of the pair, as defined in the proof of Theorem 4.1. Assume towards a contradiction that there is an execution $E$ with infinitely many pairwise steps and yet no rendezvous occurs. Consider $E_1$, the (infinite) suffix of $E$ in which any operation whose rendezvous occurs in $E$ no longer takes steps. ($E_1$ is well defined because any thread whose rendezvous occurs performs a finite number of steps before completing, and so after some point in $E$ any such operation either completes or never takes another step).

The following two lemmas show that eventually the ring size does not change, allowing us to consider an execution where the ring size remains fixed. □

**Lemma 4.1** *If the ring size changes in $E_1$, it only increases.*

*Proof* The ring size is decreased only after a rendezvous occurs (Lines 51–53). By definition of $E_1$, no thread whose rendezvous completes takes steps in $E_1$, so the ring size cannot decrease in $E'$. □

**Lemma 4.2** *From some point onwards in $E_1$, the ring size does not change.*

*Proof* Consider an event $e \in E_1$ by some operation *op* that changes the ring size. There are two possible cases: (1) $e$ is a `CAS` decreasing the ring size, which is impossible by Lemma 4.1, or (2) $e$ is a `CAS` increasing the ring size (Line 67). Before executing $e$, *op* executes Line 66, and so $e$ increases the ring size to some $r \leq T$. Therefore there can be only finitely many such ring increasing events. Since $E_1$ is infinite, from some point onwards the ring size does not change. □

Let $E'$ be the (infinite) suffix of $E_1$ in which the ring size does not change. Since there is a finite number of threads, $T$, there must be a producer/consumer pair, $p$ and $c$, each of which runs forever without completing in $E'$. We now prove that $c$ cannot run forever without capturing some ring node.

**Lemma 4.3** *At some point in $E'$, there is a node that has been captured by $c$.*

*Proof* If $c$ captures a node before $E'$ and holds it captured in $E'$, we are done. Otherwise, suppose $c$ never captures a node in $E'$. Then the ring size must be $T$. Otherwise, when $c$ tries to increase the ring size (Lines 65–67) it either succeeds or fails because another thread changes the ring size, both of which are impossible by definition of $E'$.

A ring of size $T$ has room for $c$, as $T$ is the maximum number of threads. Thus $c$ fails to capture a node only by encountering another consumer, $c'$, twice at different nodes. This $c'$ left one node and captured another only as a result of the following: (1) noticing the ring decreasing or (2) completing its operation and returning again, both of which are impossible by definition of $E'$. It therefore cannot be that $c$ never captures a node. □

Consider now the execution fragment in $E'$ in which $c$ holds some node $s$ captured. Then $s$.`item` = `NULL`. There are two possible cases: (1) $p$ visits $s$ or (2) $p$ never visits $s$.

Suppose that $p$ visits $s$ while executing its loop. Then $p$ attempts to rendezvous with $c$ by executing Line 24 or 27. If $p$'s `CAS` succeeds, a rendezvous has occurred. Otherwise, either another producer rendezvouses with $c$ or $c$ leaves $s$. As shown above, $c$ leaving $s$ implies that a rendezvous has occurred. In any case we reach a contradiction.

Therefore, suppose $p$ never visits $s$. This implies that $s$ has been left out of the ring by some decreasing resize. Since in $E'$ the ring does not decrease, $c$ eventually executes Lines 41–44 and moves itself into $p$'s range. Because the ring size does not change, this means that $p$ eventually visits $s$, a contradiction.

## 5 TripleAdp

Here we present TripleAdp, an algorithm that adds two new forms of adaptivity to address some of AdaptiveAR's limitations. First, TripleAdp adapts the roles performed by the threads to solve the problem wherein AdaptiveAR's throughput degrades as the number of producers grows in workloads with more producers than consumers. Second, TripleAdp adapts its *total* ring size to match the changing number of threads. Thus TripleAdp can run without knowing the maximum number of threads in the system. In addition, adapting the total ring size makes TripleAdp's memory consumption adaptive as well.

### 5.1 Algorithm description

The main idea behind TripleAdp is to switch between running AdaptiveAR and its *mirror image*, where *consumers* seek *producers* that wait on the ring. Consequentially, TripleAdp's main data structure is a ring of nodes with similar structure to AdaptiveAR's node (Fig. 3, Lines 1–5); the difference from AdaptiveAR is explained below. To support reallocating the ring we use indirection: a global shared `ring` variable points

```
1  struct node {
2    item[] : array of two pointers to objects
3    index : node's index in the ring
4    prev : pointer to previous ring node
5  }
6
7  shared variables:
8    ring : pointer to an array of nodes, initially
9           of size 1, and ring[1].prev = ring[1]
10
11   mode : int, initially 1.
12
13 utils:
14 allocateRing(int size, int max) {
15   node[] r := new node[max]
16   for (i in 1... max)
17     r[i] = new node
18    initialize r's nodes such that:
19     r[i].index = i
20     r[1].prev = r[max]
21     r[i].prev = r[i-1] (i > 1)
22   return r
23 }
24
25 getRingSize(node[] r) {
26   node tail := r[1].prev
27   return tail.index
28 }
29 // The node item a thread in wait() uses
30 waitItemIndex(object value)
31   if (value = null)
32     return 1
33   else
34     return 2
35 }
36
37 // The node item a thread in seek() uses
38 seekItemIndex(object value)
39   if (value != null)
40     return 1
41   else
42     return 2
43 }
44
45 // Was value deposited by a waiting thread?
46  hasWaiter(object value, int idx) {
47   if (idx = 1)
48     return value = NULL
49   else
50     return (value != NULL and value != FREE)
51 }
```

**Fig. 3** Algorithm TripleAdp: definitions and helper procedures

```
52 get(int threadId) {
53   if (mode = 1)
54     return wait(threadId, NULL)
55   else
56     return seek(threadId, NULL)
57 }
58
59 put(int threadId, object item) {
60   if (mode = 1)
61     return seek(threadId, item)
62   else
63     return wait(threadId, item)
64 }
```

**Fig. 4** Pseudo code of `get()` and `put()`

to the ring (Lines 8–9).[2] This allows a thread to install a new ring/array pair. The thread first allocates a new ring (Lines 14–23) and then installs it with a single atomic `CAS` to point `ring` at the new array and thereby the new ring. Naturally, threads waiting in the old ring need to move to the new ring, a procedure we describe later.

A thread in TripleAdp is either a *waiter* which captures a node in the ring and waits for a partner, or a *seeker* that scans the ring looking for waiters. To determine what role an arriving thread plays the thread consults a global `mode` variable (Fig. 4). If `mode`=1 then consumers wait and producers seek, as in AdaptiveAR. If `mode`=2 then roles are reversed, i.e., AdaptiveAR's mirror image is run. The mode can change during the execution in response to the workload and so the `wait` and `seek` procedures periodically check for a mode switch, upon which the thread's role is reversed. This means that role reversal does not occur atomically with the mode change, so threads in different modes can be active concurrently. To prevent such threads from stepping on each other in a ring node, each node has a two member array field named `item`, where `item[`$m$`]` is accessed only by threads in mode $m$ and encodes the node's state in mode $mode$:

1. **Free**: `item[`$mode$`]` points to a global reserved object, `FREE`, distinct from any object a producer may enqueue.
2. **Captured (by a waiter):** If $mode = 1$ (consumers waiting), `item[1]=NULL`. Otherwise (producers waiting), `item[2]` holds producer's item.
3. **Rendezvous occurred:** If $mode = 1$ `item[1]` holds producer's item. Otherwise, `item[2]=NULL`.

Figure 6 shows the pseudo code of the `wait` and `seek` procedures. For the sake of clarity, we initially describe the code without going into the details of adapting the ring size and memory consumption, to which we return later. The relevant lines of code are shown in the pseudo code marked by a □ but are ignored for the moment.

*Waiters (Fig. 6a)*: First (Line 96), a waiter determines its operation mode by invoking the helper procedure `waitItemIndex()` whose code is given in Fig. 3 (Lines 29–34). If the waiter does not have a `value` to hand off then it is a consumer, implying it is in the consumers waiting mode ($m = 1$). Otherwise, it is a producer and $m = 2$. The waiter then attempts to capture a node in the ring and waits there

[2] This is standard array semantics in Java, but not in C++.

```
65 findFreeNode(value, node[] r, int ringsize, node s) {
66 □ int busy_ctr := 0
67   int i := waitItemIndex(value)
68   while (true) {
69     if (s.item == FREE and
70         CAS(s.item[i], FREE, value))
71       return (s,busy_ctr,OK)
72     s := s.prev
73     if (mode != i)
74       return (*,*,MODE−CHG)
75 □   if (ring != r)
76 □     return (*,*,RING−CHG)
77 □   busy_ctr := busy_ctr + 1
78 □   if (busy_ctr > T_i·ringsize) {
79 □     if (ringsize < size(r)) {
80 □     // Increase effective ring size
81 □       CAS(r[1].prev, r[ringsize],
82 □           r[ringsize+1])
83 □     } else { // Allocate larger ring
84 □       node[] nr = allocateRing(
85 □                     ringsize+1,
86 □                     2·len(r))
87 □       CAS(ring, r, nr)
88 □     }
89 □     ringsize := getRingSize(r)
90 □     busy_ctr := 0
91 □   }
92   }
93 }
```

**Fig. 5** Pseudo code of `findFreeNode()`. Lines marked by a □ handle ring size adaptivity

for a seeker to rendezvous with it. This process generally follows that of AdaptiveAR. The waiter starts by searching the ring for free nodes to capture. It hashes its id to obtain a starting point for this search (Lines 99–100) and then invokes the `findFreeNode` procedure to perform the search (Line 101). The pseudo code of `findFreeNode` is shown in Fig. 5. Here the waiter traverses the ring and upon encountering a `FREE` node, attempts to `CAS` the node's `item` to its `value` (Lines 66–69). (Recall that `value` holds the thread's item if it is a producer, and `NULL` otherwise.) The traversal is aborted if the global mode changes from the current mode (Lines 70–71) and the waiter then reverses its role (Lines 102–103). Once `findFreeNode` captures a node $u$ (Line 101), the waiter spins on $u$ and waits for a seeker. If the waiter detects a global mode change during this period it releases $u$ and reverses its role (Lines 121–122). The waiter returns (Line 199) once it notices that a seeker deposited a value in $u$ (Lines 107–109). If the waiter is a consumer, it returns the seeking producer's deposited value, or `NULL` otherwise.

*Seekers (Fig. 6b)*: A seeker traverses the ring, looking for a waiter to rendezvous with. Symmetrically to the waiter code, the seeker starts by determining its operation mode (Line 137). The seeker then begins traversing the ring from an initial node $s$ obtained by hashing the thread's id (Lines 139–140). As in the waiter case, the traversal process follows that of AdaptiveAR: the seeker proceeds while *peeking* at $s$ to see if a waiter has arrived there in the meantime (Lines 148–153). If $s$ has no waiter, the next node in the traversal is checked. To determine if a node is captured by a waiter, the seeker uses the helper procedure `hasWaiter()` (Fig. 3, Lines 45–50). Once a waiter is found, the seeker tries to match with it by atomically `CAS`ing the value stored in the node's `item` with its `value`. If successful, it returns the retrieved value. During each iteration of the loop, the seeker validates that it is in the correct mode and switches roles if not (Lines 144–145).

*Adapting effective and total ring size* (□*-marked lines*): The waiter adjusts both the effective and total ring size. Deciding when to resize the ring and adjusting its effective ring size is performed as in AdaptiveAR (Lines 74–88, 76–78, 110 and Line 116). Where AdaptiveAR would fail to increase the effective ring size due to using a statically sized ring, in TripleAdp the waiter allocates a new ring of double the size and tries to install it using a `CAS` on `ring` (Lines 81–84). In the other direction, when the effective ring size is a quarter of the ring size the waiter swings `ring` to point to a new allocated ring of half the old size (Lines 111–115). Seekers and waiters check whether a new ring has been allocated (Lines 146–147 and 124, 72–73). If so they move themselves to the new ring by restarting the operation.

*Adapting to the access pattern*: Seekers are responsible for adapting the operation mode to the access pattern. The idea is to assign the role of waiter to the majority thread type (producer or consumer) in an asymmetric access pattern. Ring size adaptivity then sizes the ring according to the number of waiters. This reduces the chance for seekers (which are outnumbered by the waiters) to collide with each other on the ring. Following this intuition, a seeker deduces that the operation mode should be changed if it starts to experience collisions with other seekers in the form of `CAS` failures. A seeker maintains a counter of failed `CAS` attempts (Lines 155, 165). A seeker that completes a rendezvous checks if its counter is above a *contention threshold* $T_f$, and if so it toggles the global operation mode using a `CAS` on `mode` (Lines 151–152, 161–162).

*Memory reclamation*: We assume the runtime environment supports garbage collection (GC).[3] Thus, once the pointer to an old ring is overwritten and all threads move to the new ring, the garbage collector reclaims the old ring's memory.

We assume GC for simplicity. A different memory management scheme that allows safe memory reclamation for concurrent nonblocking code will work as well, but may require changes to TripleAdp to interface with the memory manager. For example, consider the use of hazard pointers [14]. An operation must point a hazard pointer to the ring it will access after reading the global ring pointer (Lines 98, 139). After pointing the hazard pointer at the ring, the

[3] GC is part of modern environments such as C# and Java, in which most prior synchronous queue algorithms were implemented [1,8,17].

```
94  wait(int threadId, object value) {
95    int i, ctr, busy_ctr, ringsize
96    node s, u
97    node[] r
98
99  □ ctr := 0
100   i := waitItemIndex(value)
101   while (true) {
102 □   r := ring // private copy of ring pointer
103     ringsize := getRingSize(r)
104     s := r[hash(threadId, ringsize)]
105     (u,busy_ctr,ret) := findFreeNode(value, r, ringsize, s)
106     if (ret = MODE−CHG)
107       return seek(threadId, value)
108 □   if (ret = RING−CHG)
109 □     continue; // Restart loop
110     while (true) {   // Wait for partner
111       if (u.item[i] != value) {
112         object item := u.item[i]
113         u.item[i] := FREE
114 □       if (busy_ctr < T_d and ctr > T_w and ringsize>1) {
115 □         // Try to decrease ring
116 □         if (ringsize−1 = size(r)/4) {
117 □           node[] nr = allocateRing(ringsize−1, size(r)/2)
118 □           CAS(ring, r, nr)
119 □         } else {
120 □           CAS(r[1].prev, r[ringsize], r[ringsize−1])
121 □         }
122         }
123         return item
124       }
125       if (mode != i and CAS(u.item[i], value, FREE))
126           return seek(threadId, value)
127 □     ringsize := getRingSize(r)
128 □     if ((u.index > ringsize or ring != r) and
129 □         CAS(u.item[i], value, FREE)) {
130 □         break; // Restart outer loop
131 □     }
132 □     ctr := ctr + 1
133     }
134   }
135 }
```

**(a)**

```
136 seek(int threadId, object value) {
137   object item
138   int i, ctr
139   node s, v
140   node[] r
141
142   i := seekItemIndex(value)
143   while (true) {
144 □   r := ring // private copy of ring pointer
145     s := r[hash(threadId, getRingSize(r))]
146     v := s.prev
147     ctr := 0
148     while (true) {
149       if (mode != i)
150         return wait(threadId, value)
151 □     if (ring != r)
152 □       break; // Restart outer loop
153       item := s.item[i]
154       if (hasWaiter(item, i)) {
155         if (CAS(s.item[i], item, value)) {
156           if (ctr > T_f)
157             CAS(mode, i, 3−i) // toggle 1,2
158           return item
159         }
160         ctr := ctr + 1
161         continue  // Restart inner loop
162       }
163       item := v.item[i]
164       if (hasWaiter(item, i)) {
165         if (CAS(v.item[i], item, value)) {
166           if (ctr > T_f)
167             CAS(mode, i, 3−i) // toggle 1,2
168           return item
169         }
170         ctr := ctr + 1
171         v := v.prev
172       }
173     }
174   }
175 }
```

**(b)**

**Fig. 6** TripleAdp pseudo code for seeking and waiting threads. Lines marked by a □ handle ring size adaptivity

operation must check that the global `ring` pointer has not changed and retry the process if it has. If the ring has not changed since the operation wrote its hazard pointer, it is guaranteed that the ring will not be reclaimed until the operation's hazard pointer changes [14]. The operation must therefore nullify its hazard pointer once it completes. The proof of Theorem 5.2 (Sect. 5.2) can be extended to show that TripleAdp remains pairwise-nonblocking when using hazard pointers.

## 5.2 Correctness proofs

**Theorem 5.1** *TripleAdp implements asymmetric rendezvous.*

*Proof* We prove linearizability and synchrony of TripleAdp:

*Synchrony:* Let $p$ be some `put(`$x$`)` operation that successfully `CAS`es $x$ into `item[`$i$`]` pointer while in `seek(`$p, x$`)` (Fig. 6b, Lines 150,160). It follows that $i = 1$, since that is the index `seekItemIndex()` returns when passed $x \neq$ `NULL` (Fig. 3, Lines 38–39). This, in turn, implies that $p$ observes `NULL` in `item`[1] before proceeding with its `CAS`, as that is what `hasWaiter()` returns in this case (Lines 46–47). Thus the waiter found by $p$ is some `get()` operation, $c$, that invokes `wait(`$c$, `NULL)` before $p$ `CAS`es $x$ into the node.

We prove that if $c$ continues to take steps after $p$'s `CAS`, it returns $x$. When $c$ executes Line 107, it notices $x$ written to $s$.`item`[1] and returns $x$ (Lines 107–119). Notice that even if $c$ attempts to free its node to a mode change (Lines 121–122) or ring resize (Lines 124–127), its `CAS` will fail and thus $c$ will proceed to execute Line 107.

A symmetric argument shows that a `get()` operation which successfully `CAS`es `NULL` into some node's `item`[$i$] pointer must have $i = 2$ and must therefore observe a producer waiting in some `wait(`$p, x$`)` call.

*Linearizability:* We define linearization points for the operations as follows, depending on the roles they play in TripleAdp. Let $p$ be a `put(`$x$`)` operation that completes a rendezvous while executing `seek(`$p, x$`)`. Then $p$ is linearized at its successful `CAS` of $x$ into $s$.`item`[1], for some node $s$. As shown above, there is then a unique concurrent `get()` operation, $c$, that `CAS`es `NULL` to $s$.`item`[1] from `wait(`$c$, `NULL)` before $p$'s linearization point, and which returns $x$ if it takes enough steps. We linearize $c$ returning $x$ immediately after $p$, at the same `CAS` event. Conversely, let $c$ be a `get()` operation which completes its operation in `wait(`$c$, `NULL)`. Then $c$ captures a node $s$ by `CAS`ing `NULL` into $s$.`item`[1], and then spins until observing some item $x$ stored in $s$.`item`[1] (Line 107). This occurs as a result of some producer `CAS`ing $x$ into $s$.`item`[1]. It follows that in the process described above of assigning linearization points to `put(`$x$`)`/`get()` pairs, we assign a linearization point to $c$.

Symmetrically, let $c$ be a `get()` operation that completes a rendezvous from `seek(`$c, NULL$`)`. Then $c$ successfully `CAS`es `NULL` into $s$.`item`[2], for some node $s$, and there exists a `put(`$x$`)`, $p$, executing `wait(`$p, x$`)` which previously `CAS`es $x$ into $s$.`item`[2]. We linearize $p$ at $c$'s successful `CAS`, and linearize $c$ returning $x$ immediately after $p$, at the same `CAS` event. Conversely, let $p$ be a `put(`$x$`)` operation that completes a rendezvous while in `wait(`$p, x$`)`. Then $p$ captures a node $s$, `CAS`es $x$ into $s$.`item`[2], and then spins until observing that $x$ is not stored in $s$.`item`[2] (Line 107). This occurs as a result of some `get()` running `seek()` `CAS`ing `NULL` into $s$.`item`[2]. It follows that in the process described above of assigning linearization points to `get()`/`put(`$x$`)` pairs, we assign a linearization point to $p$.

Thus, any operation that completes has a linearization point assigned to it. Because we always linearize a `put()` followed immediately by its matching `get()` at a successful `CAS` operation by one member of the pair, our linearization points assignment yields a sequence of matching `put()`/`get()` pairs. It remains to show that this sequence is a valid execution of the handoff object. To do this, we show using induction on the number of `put()`/`get()` pairs linearized that our linearization point assignment constructs a valid execution of the handoff object that leaves the object in state $\perp$. In the base case no operation is linearized, and this is the (valid) empty execution.

For the induction step, we have a valid execution of the handoff object that ends with the object in state $\perp$. The next operation linearized is a `put(`$x$`)`, which is valid and changes the object's state to $x$. We then linearize a `get()` returning $x$, which is valid and changes the object's state to $\perp$. □

**Theorem 5.2** *TripleAdp is pairwise nonblocking.*

*Proof* Similarly to Theorem 4.2's proof, we say that the rendezvous of a `put()` and `get()` pair *occurs* at the linearization point of the pair, as defined in the proof of Theorem 5.1. The proof is by contradiction. Assume that there is an execution $E$ in which there are infinitely many pairwise steps and yet no rendezvous occurs. We consider $E'$, the infinite suffix of $E$ in which any operation whose rendezvous occurs in $E$ no longer takes steps. Notice that $E'$ cannot contain mode switches, since a mode switch happens after a seeker's rendezvous occurs (Lines 150,160). Thus, in $E'$ TripleAdp either stays in producers waiting mode (AdaptiveAR mirror image) or consumers waiting mode (essentially AdaptiveAR). We first prove that if no ring reallocations occur in $E'$, then these modes are pairwise nonblocking, leading to a contradiction (Lemmas 5.1 and 5.2). Thus TripleAdp must reallocate the ring in $E'$. In fact, we show that TripleAdp must reallocate the ring infinitely often (Lemma 5.3). We conclude by showing that this too implies a contradiction. □

**Lemma 5.1** *If from some point onwards in $E'$, no ring reallocations occur and TripleAdp runs with $mode = 2$ then a rendezvous occurs.*

*Proof* Assume towards a contradiction that the lemma is false. Then, as in Theorem 4.2's proof, there is a producer/consumer pair, $p$ and $c$, each of which runs forever without completing. Let $r$ be the ring after the last ring reallocation in $E'$. Then $p$ and $c$ must eventually execute on $r$: eventually each of them executes Lines 146–147 or 124,72–73) and, if its current ring differs from $r$, moves itself to $r$. Once $p$ and $c$ are on the same ring, the same arguments as Theorem 4.2's proof show that $p$ must eventually capture a node. This implies that $c$ doesn't find $p$ on the ring. This can only occur if $p$ was left out of the ring because some producer decreased the ring. But, since from that point on, the ring size cannot decrease (otherwise a rendezvous completes), $p$ eventually moves itself into $c$'s range, where either $c$ or another consumer rendezvous with it, a contradiction. □

**Lemma 5.2** *If from some point onwards in $E'$, no ring reallocations occur and TripleAdp runs with $mode = 1$ then a rendezvous occurs.*

*Sketch of proof* Following Lemma 5.1's proof. □

**Lemma 5.3** *If $E'$ contains a finite number of ring reallocations, a rendezvous occurs.*

*Proof* After the last ring reallocation, either Lemma 5.1 or 5.2 applies. □

It follows that $E'$ must contain infinitely many ring reallocations. A ring reallocation that reduces the total ring size (Lines 111–115) is performed after a waiter's rendezvous occurs, which is impossible. Thus, the only ring reallocations possible are ones that increase the total ring size. To complete the proof, we show that there cannot be infinitely many such ring reallocations. Here we use the assumption that the concurrency is finite. Let $T$ be the maximum number of threads

that can be active in $E$; we show that the ring size cannot keep growing indefinitely. Assume w.l.o.g. that the increase threshold $T_i$ is 1. Let $R$ be the initial ring to be installed with size $size(R) \geq T$. Then $R$'s effective size always remains $\geq T$, because a ring size decrease implies that a rendezvous has occurred. Consider the first waiter $w$ that tries to reallocate $R$. Then $w$ observes $size(R)$ occupied nodes, i.e., it observes another waiter $w'$ in two different nodes. $w'$ leaves one node and returns to capture another node due to one of the following reasons: a mode switch (impossible), a ring reallocation (impossible, since $w$ is the first to reallocate the ring), or because $w'$ successfully rendezvouses, a contradiction.

## 6 AdaptiveAR and TripleAdp evaluation

We study the performance of our algorithms on a number of rendezvous workloads. We compare them to Java's unfair synchronous queue (or *channel*) (JDK), ED tree, and FC synchronous channels. We also compare our algorithms to a rendezvous implementation based on Java's Exchanger algorithm [18,12]. The Exchanger is an adaptively sized collision array whose design is inspired by the collision arrays used for elimination in stacks and queues [9,16], but with an optimized implementation as appropriate for a package in Java's concurrency library. As in the original collision array, the Exchanger is *symmetric* and any two arriving threads can be matched. Therefore, our Exchanger-based rendezvous implements the `put()` and `get()` operations by having the thread repeatedly invoke the Exchanger until obtaining a match with an opposite operation. Since threads may continuously obtain false matches, this algorithm is not a practical approach for implementing rendezvous. However, its similar characteristics to our algorithm make it an interesting yardstick to compare to.

*Experimental setup*: Evaluation is carried out on both a Sun SPARC T5240 and an Intel Xeon E7-4870. The Sun has two UltraSPARC T2 Plus (Niagara II) chips. Each is a 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. Each core has a private L1 write-through cache and the L2 cache is shared. The Intel Xeon E7-4870 (Westmere EX) processor has 10 2.40 GHz cores, each multiplexing 2 hardware threads. Each core has private write-back L1 and L2 caches and the L3 cache is shared.

Both Java and C++ implementations are tested.[4] For the prior algorithms, we use the original authors' implementation.[5] Following Hendler et al. [8], we test both JDK and JDK-no-park, a version that always uses busy waiting instead of yielding the CPU (parking), and report the results of the variant that performs best in each workload.

Unless stated otherwise our benchmarks use dedicated threads, each performing only `put()` or `get()` operations in a loop over a period of 10 s, and results are averages of ten such runs of the Java implementation on an idle machine. Except when stated otherwise, our results had very little variance. Throughout the evaluation we use a modulo hash function ($\texttt{hash}(t) = t \bmod \texttt{ringsize}$) in AdaptiveAR and the Exchanger to map thread ids to ring nodes. Thread ids are assigned uniformly at random at the start of each run, to model the fact that in practice little is known about which threads try to rendezvous at any point in time. The adaptivity thresholds we use for both AdaptiveAR and TripleAdp are $T_i = 1, T_d = 2, T_w = 64$. In addition, we set $T_f = 10$ for TripleAdp.

*Thread placement*: Unless stated otherwise, we ran the experiments without pinning threads to specific cores. While pinning threads to cores can sometimes be useful to reduce measurement noise, it also poses a problem: which thread-to-core placement to use? Any fixed policy may help or harm a specific algorithm [23], and the optimal policy for each algorithm is an open research question. We therefore let the operating system perform thread scheduling, relying on averaging many executions to reduce noise. As noted above, our results have very little variance.

### 6.1 Symmetric producer/consumer workload

We measure the throughput at which data is transferred from $N$ producers to $N$ consumers. Figure 7a shows the results from running on a single multi-core chip of the SPARC machine. Only AdaptiveAR, TripleAdp, Exchanger and Parallel FC show meaningful scalability. In low concurrency settings AdaptiveAR outperforms the Parallel FC channel by $2\times - 3\times$ and the Exchanger by $2\times$. At high thread counts AdaptiveAR obtains 5.5 times the throughput of Parallel FC and 2.7 times that of the Exchanger. TripleAdp falls below AdaptiveAR by 20 % for all thread counts (this difference becomes noticeable in the graphs as the thread count increases). Despite this, TripleAdp outperforms Parallel FC by $4.45\times$ and the Exchanger by $2.3\times$.

The remaining algorithms scale poorly or not at all. Hardware performance counter data (Fig. 7b–d) shows that these implementations exhibit some kind of serialization: the

[4] Java benchmarks were ran with HotSpot Server JVM, build `1.7.0_05-b05`. 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 [3] memory allocator.

[5] 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 channel as successful operations; thus the results we report for it are sometimes lower.

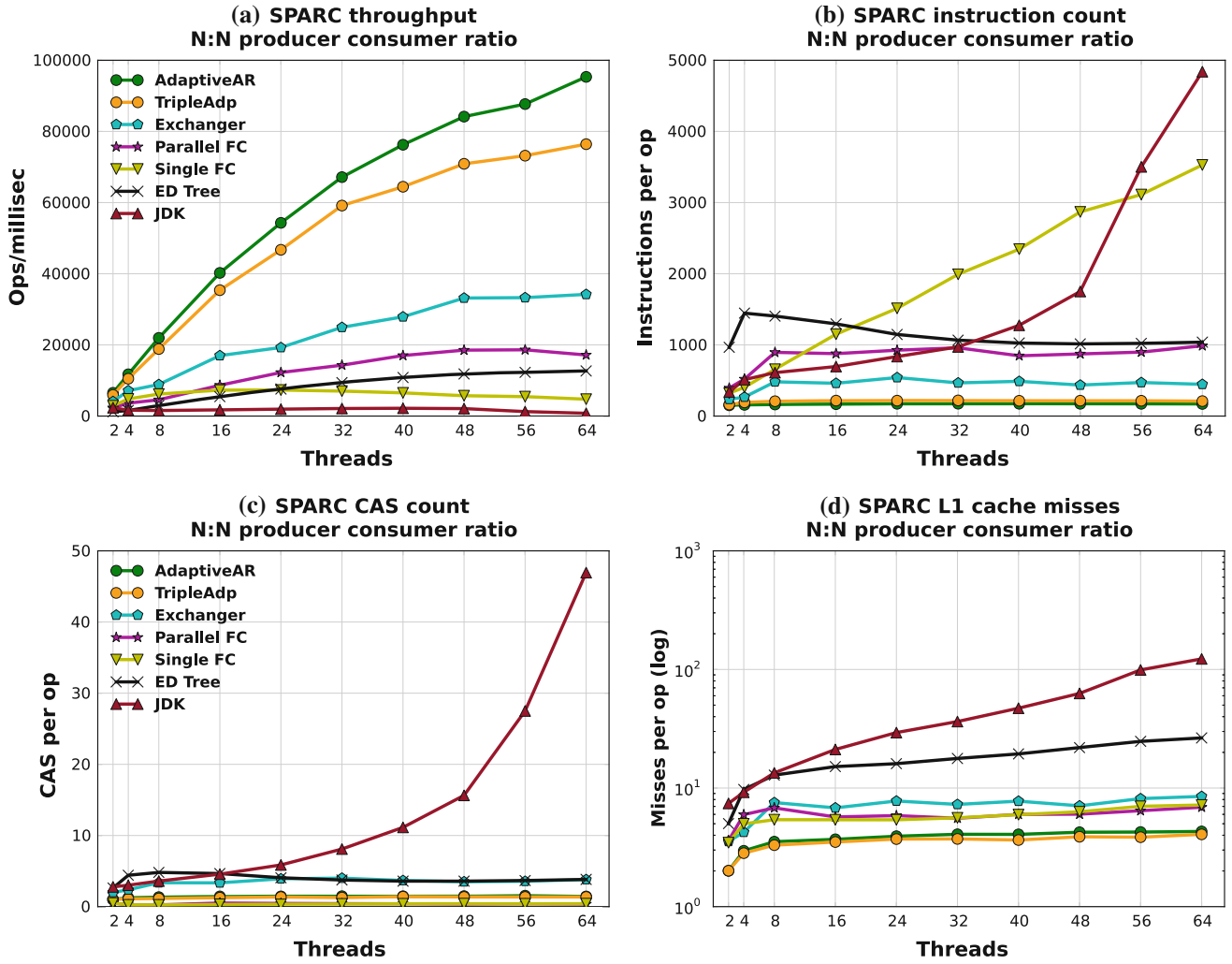


**Fig. 7** Rendezvousing between producer/consumer pairs: SPARC results. L2 misses are not shown; all algorithms but JDK had less than one L2 miss/operation on average

number of instructions per rendezvous for JDK and Single FC increases as concurrency grows; for the ED tree, the cache miss plot shows a growing trend (Fig. 7d). Taking JDK for example, as concurrency grows a thread requires more attempts until its CAS to the top of the stack succeeds (Fig. 7c) which causes the overall instruction count to grow too. Consequently, as concurrency increases JDK's throughput deteriorates while AdaptiveAR's increases, resulting in around 120× throughput difference at 64 threads (32 producer/consumer pairs).

Turning back to the more scalable algorithms, we find their throughput differences correlated with the implementations' *path length*, i.e., the average number of instructions per successful rendezvous (Fig. 7b). The path length is crucial on the SPARC, which is an in-order processor. AdaptiveAR completes a rendezvous in less than 170 instructions of which one is a CAS, and this number remains stable as concurrency grows. Compared to AdaptiveAR, TripleAdp performs extra work in each rendezvous when checking for mode switches and ring reallocations. This extra work is wasted in this workload, in which mode switches and ring reallocations are rare, leading to the 20 % throughput drop. In comparison, while Parallel FC hardly performs CASes (Fig. 7c) the time spent waiting for the combiner adds up and Parallel FC requires 2.6× to 5.8× more instructions than AdaptiveAR to complete a rendezvous. Finally, an Exchanger rendezvous takes 1.5× to 2.74× more instructions. The main reason is false matches which occur with probability 1/2, so on average two exchanges are required to complete a rendezvous. The remaining throughput gap is due to other algorithmic differences between AdaptiveAR and the Exchanger, such as peeking, which we expand upon in Sect. 6.1.4.

The throughput results on the Intel processor (Fig. 8a) are similar to the low concurrency SPARC results for all algorithms but AdaptiveAR and TripleAdp, which now obtain comparable throughput to the Exchanger. Here the average

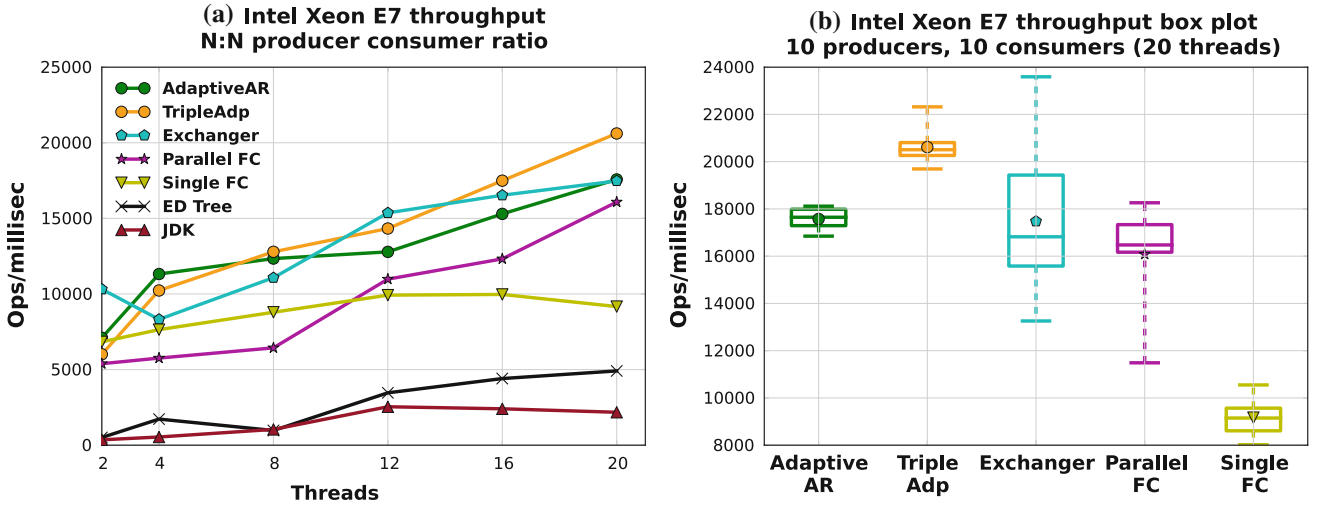


**Fig. 8** Rendezvousing between producer/consumer pairs: Intel results. *Box plot* on the *right* zooms in on 10:10 results. The *bottom* and *top* of a *box* are the 25th and 75th percentiles of the data, with a line at the median and a marker at the average. Whiskers extend to the minimum and maximum of the data

results, shown in Fig. 8a, do not tell the whole story. Figure 8b, which includes more details about the results, shows that AdaptiveAR's throughput is much more stable than the Exchanger. Still, why does the Exchanger surpass AdaptiveAR in some executions?

It turns out that on this processor and workload the Exchanger's symmetric behavior can be advantageous due to the write-back caching architecture. In the Exchanger algorithm a thread captures a slot by pointing that slot's pointer field (using CAS) to a private memory location on which the thread spins. A thread finding an occupied slot tries to nullify the pointer using CAS and, if successful, to rendezvous at the private location. Now, consider a *stable* producer/consumer pair, $p$ and $c$, whose ids hash to the same array slot $s$. Because thread ids remain constant throughout the run, this pair is likely to repeatedly rendezvous at $s$. As we now explain, this results in $s$ moving between these threads' caches, allowing them to complete a rendezvous more efficiently. When an Exchanger match occurs at $s$, the thread that arrives later performs a CAS on $s$ and leaves shortly after, whereas the thread that occupies $s$ can only leave after noticing that a match has occurred in its private location. Thus the thread that arrives last at $s$, say $p$, will return first and when it does its cache will have exclusive access to $s$'s cacheline. This enables it to immediately capture $s$; when its partner $c$ shows up shortly after it will see $s$ occupied and complete the match. Thus the Exchanger's symmetry allows a stable thread pair $p$ and $c$ to repeatedly change their roles in the collision array. In different executions there are different number of stable pairs, hence the Exchanger's erratic performance.

In AdaptiveAR, however, thread roles are fixed. So while in a stable pair the consumer $c$ always writes to the slot $s$ last, by then its partner $p$ has already returned for another rendezvous. When $p$ reads $s$, $c$'s cache loses exclusive access to $s$, which means that when $c$ returns and tries to CAS $s$, a cache miss will occur. Furthermore, $p$ which sees that $s$ is not occupied may proceed to other nodes in the ring, experiencing cache misses as a result and causing cache misses for threads whose ids map to these nodes. Because this workload is symmetric, TripleAdp essentially fixes the thread roles and thus behaves similarly to AdaptiveAR.

Figures 9a, b demonstrate this effect. In Fig. 9a we additionally plot the results of running the benchmark with an artificial *sequential* ids distribution that creates $N$ stable pairs. Thus the Exchanger no longer wastes work on false matches, resulting in a 2.4× reduction of instructions spent per operation. Every match is also more efficient due to the previously described effect, yielding an 2.6× reduction in cache misses and almost 3× better throughput. With AdaptiveAR and TripleAdp, however, there is no improvement in throughput. The SPARC (Fig. 9c) has a write-through cache architecture in which all CAS operations are applied in the shared L2 cache and invalidate the L1 cacheline. There is therefore no advantage for being the last thread to CAS a slot. The Exchanger's only benefit from the elimination of false matches is in the amount of work done, yielding a 1.6× throughput improvement. Similarly, AdaptiveAR's and TripleAdp's throughput increases by 12–14 % because consumers capture slots immediately without contention and producers find them faster.

*Varying arrival rate:* How important are the effects described above in practice, where threads do some work between rendezvouses? To study this question we measure how the throughput of the producer/consumer pairs workload is affected when the thread arrival rate decreases due to increasingly larger amounts of time spent doing "work"

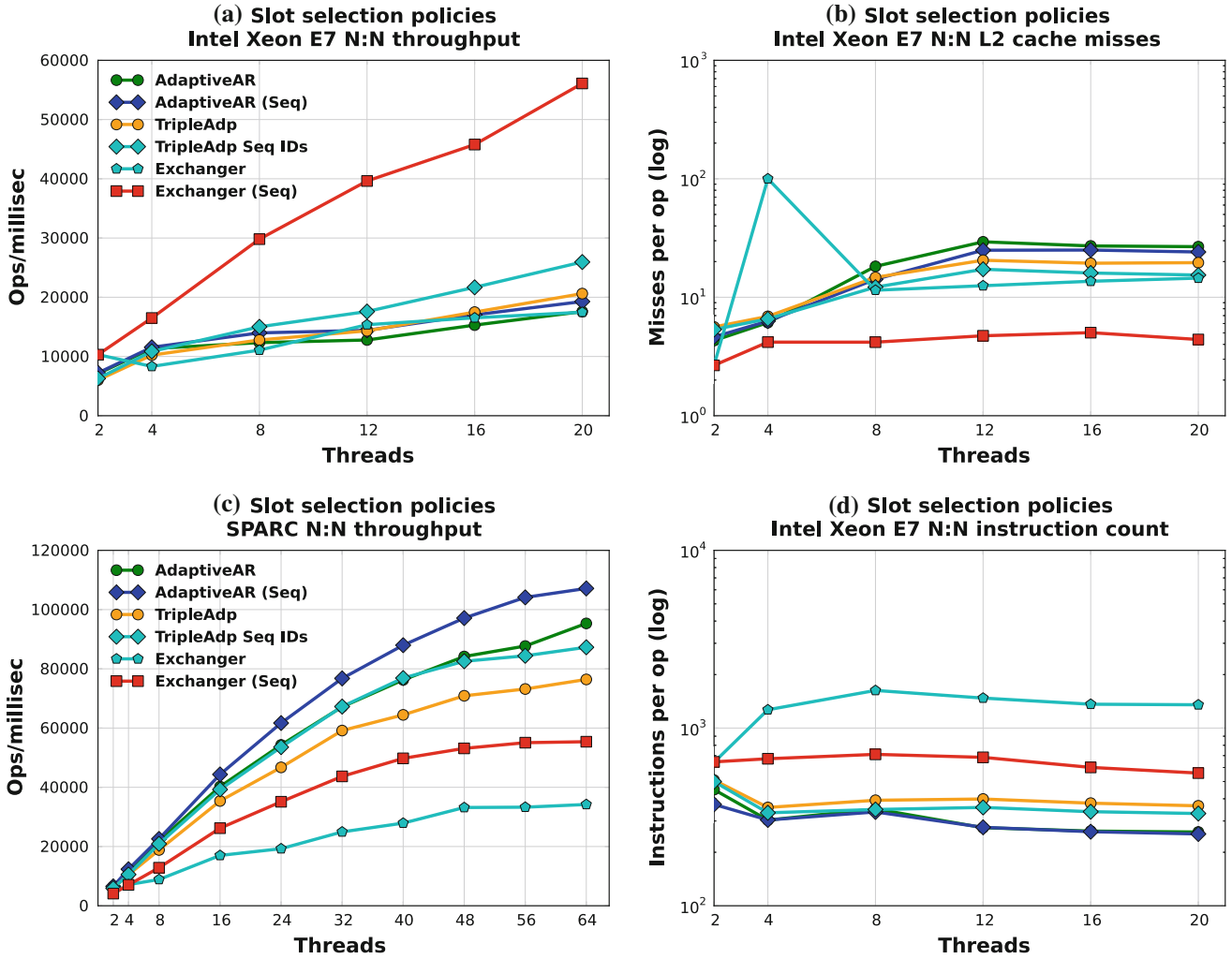


**Fig. 9** Effect of initial slot selection on $N : N$ workload throughput. Performance counter plots are logarithmic scale. There were no L3 (shared) cache misses on the Intel processor

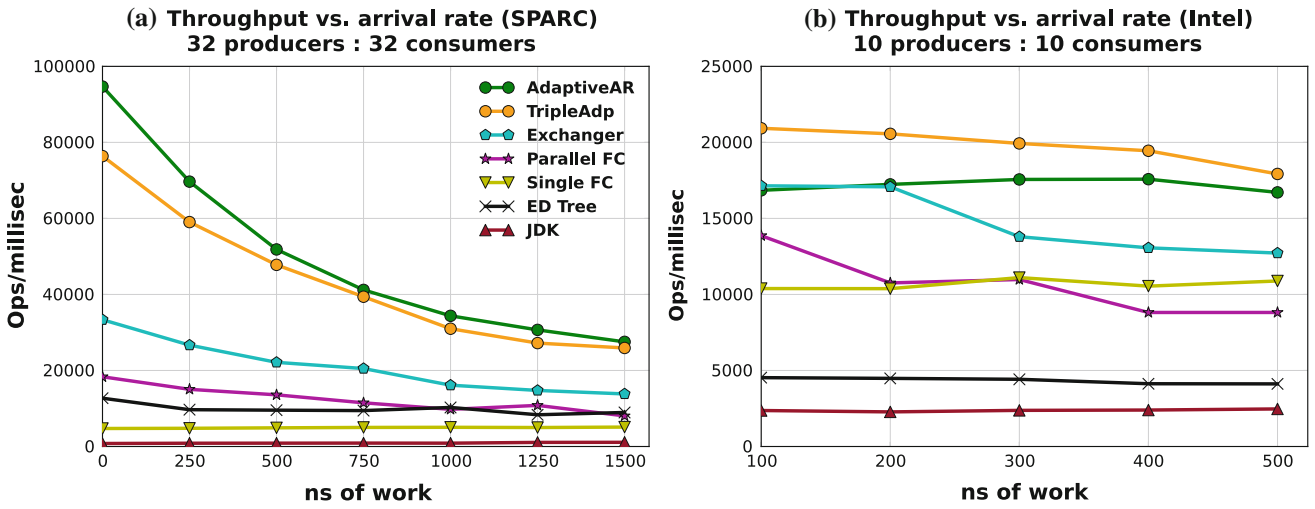


**Fig. 10** $N : N$ rendezvousing with decreasing arrival rate due to increasing amount of work time between operations

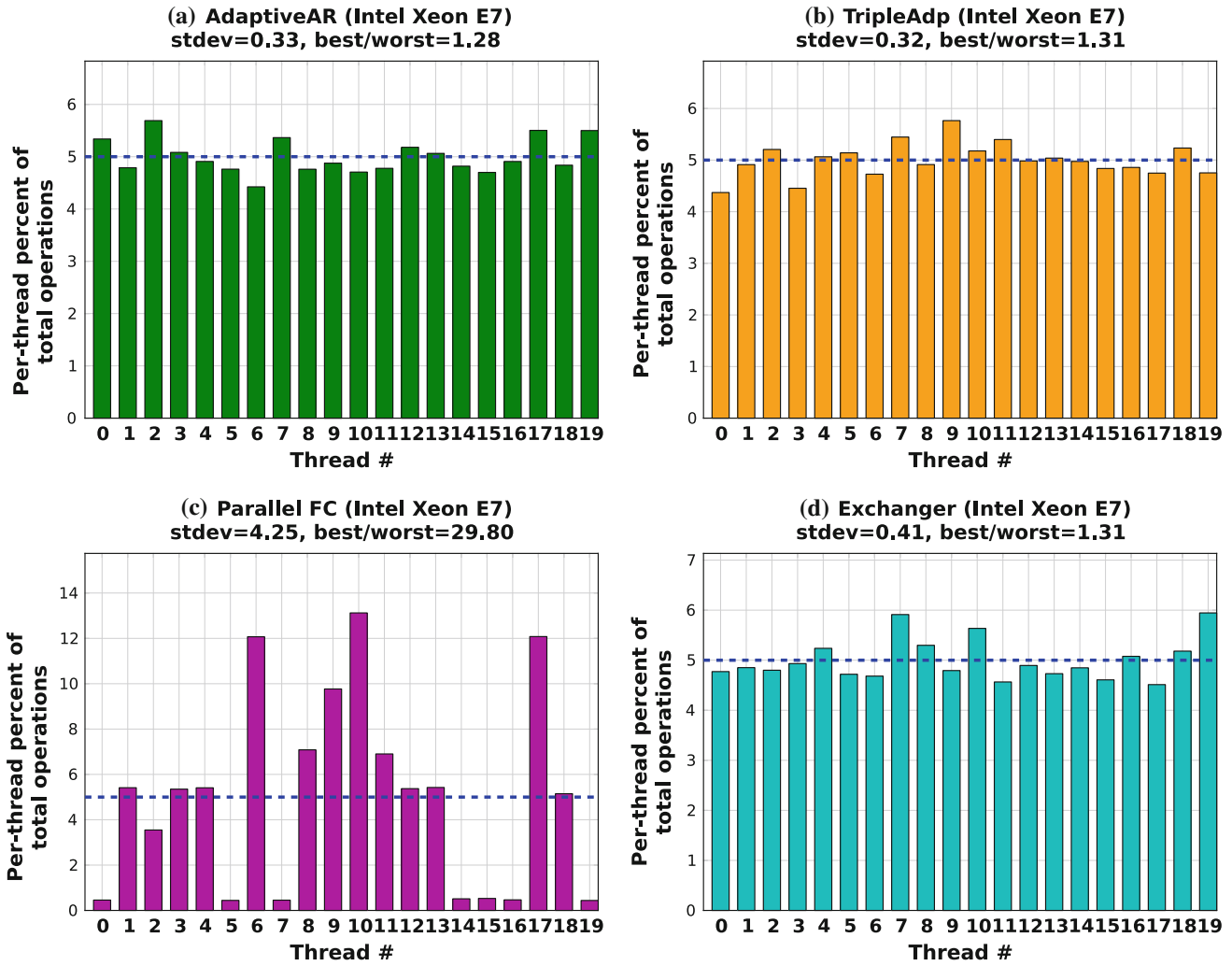


**Fig. 11** Percent of operations performed by each thread in an $N : N$ test with maximum concurrency. The *horizontal line* marks ideal behavior

before each rendezvous. Figure 10a, b show that as the work period grows the throughput of all algorithms that exhibit scaling deteriorates, due to the increase in the sequential part of the workload at the expense of the parallel time. Going from no work to 1.5$\mu$s of work, Parallel FC on the SPARC degrades by 2×, the Exchanger by 2.5×, and AdaptiveARand TripleAdp degrades by about 3× (because they start much higher). Still, the SPARC has sufficient parallelism to allow AdaptiveAR and TripleAdp to outperform the other implementations by at least 2×. On the Westmere EX even a minimal delay between rendezvousing diminishes the impact that stable pairs have on the Exchanger's throughput, and as the amount of work increases AdaptiveAR's and TripleAdp's throughput increases up to 1.30× that of the Exchanger.

### *6.1.1 Work uniformity*

One clear advantage of AdaptiveAR over FC is work *uniformity*, since the combiner in FC spends time doing work for other threads at the expense of its own work. We show this by comparing the percent of total operations performed by each thread in a multiple producer/multiple consumer workload. In addition to measuring uniformity, this test is a yardstick for progress *in practice*: if a thread starves we will see it as performing very little work compared to other threads. In Fig. 11 we pick the best result from five executions with maximum concurrency and plot the percent of total operations performed by each thread. On the Intel the ideal uniform behavior is for each thread to perform 5% (1/20) of the work. AdaptiveAR (Fig. 11a) and TripleAdp (Fig. 11b) are relatively close to uniform, with the highest performing thread completing about 1.28× (1.31× for TripleAdp) the operations as the lowest performer. However, the other algorithms show non-uniform behavior. Parallel FC (Fig. 11c) has a best/worst ratio of 30×, showing the impact that performing combining has on the combiner. On the SPARC the results are mostly similar and we therefore omit them.

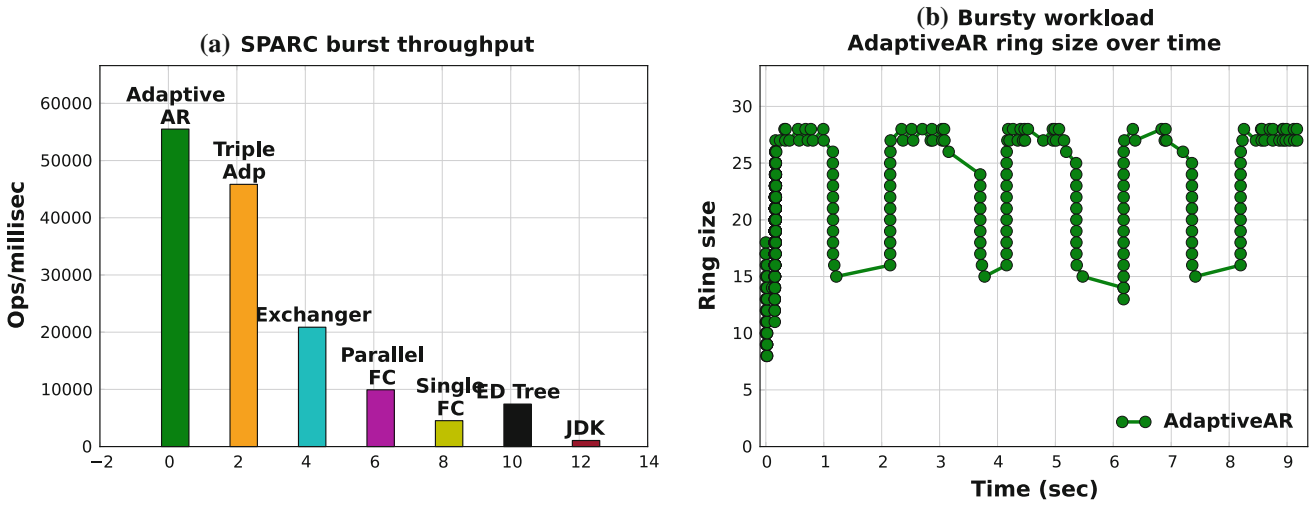


**Fig. 12** Throughput of synchronous queue bursty workload. AdaptiveAR's ring size is sampled continuously using a thread that does not participate in the rendezvousing

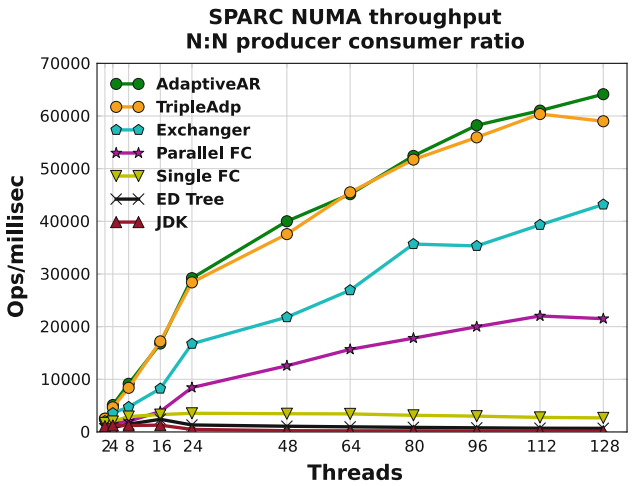


**Fig. 13** $N$ : $N$ rendezvousing on both processors of the SPARC machine

### 6.1.2 NUMA architecture

When utilizing both processors of the SPARC machine the operating system's default scheduling policy is to place threads round-robin on both chips. Thus, the cross-chip overhead is noticeable even at low thread counts, as Fig. 13 shows. Since the threads no longer share a single L2 cache, they experience an increased number of L1 and L2 cache misses; each such miss is expensive, requiring coherency protocol traffic to the remote chip. The effect is catastrophic for serializing algorithms; for example, the Java channel's throughput at 128 threads is 5× worse than its throughput with 64 threads running on a single chip.

Despite the cross-chip communication overhead, the algorithms with a scalable distributed design, namely AdaptiveAR, TripleAdp, Parallel FC and the Exchanger, show scaling trends similar to the single chip ones but with overall lower throughput. AdaptiveAR's NUMA throughput with 32 producer/consumer pairs is about half of its single-chip 32 pair throughput, TripleAdp's drops by 67 %, the Exchanger by 30 % and Parallel FC by 10 %.

We believe that the throughput of all the scalable algorithms can be improved using NUMA-aware scheduling to minimize cross-chip communications; we leave this for future work. In this paper we are interested in the performance opportunities opened up by the shift from classic SMP-style parallelism to multicore chips with low synchronization and communication overheads, and so we focus our evaluation on the single chip case.

### 6.1.3 Bursts

To evaluate the effectiveness of our concurrency level adaptivity technique, we measure the rendezvous rate on the SPARC machine in a workload that experiences bursts of concurrency changes. For 10 s the workload alternates every second between 31 thread pairs and 8 pairs. The 63rd hardware strand is used to take ring size measurements. This sampling thread continuously reads the ring size and records the time whenever the current read differs from the previous read. Figure 12a shows the throughput results; here AdaptiveAR obtains 2.62× the throughput of the Exchanger, 5× that of Parallel FC and 40× that of JDK. TripleAdp lags behind AdaptiveAR by 20 %. Figure 12b depicts how AdaptiveAR successfully adapts its ring size as the concurrency level changes over time. The results for TripleAdp are essentially identical, so we omit them to keep the graph readable.

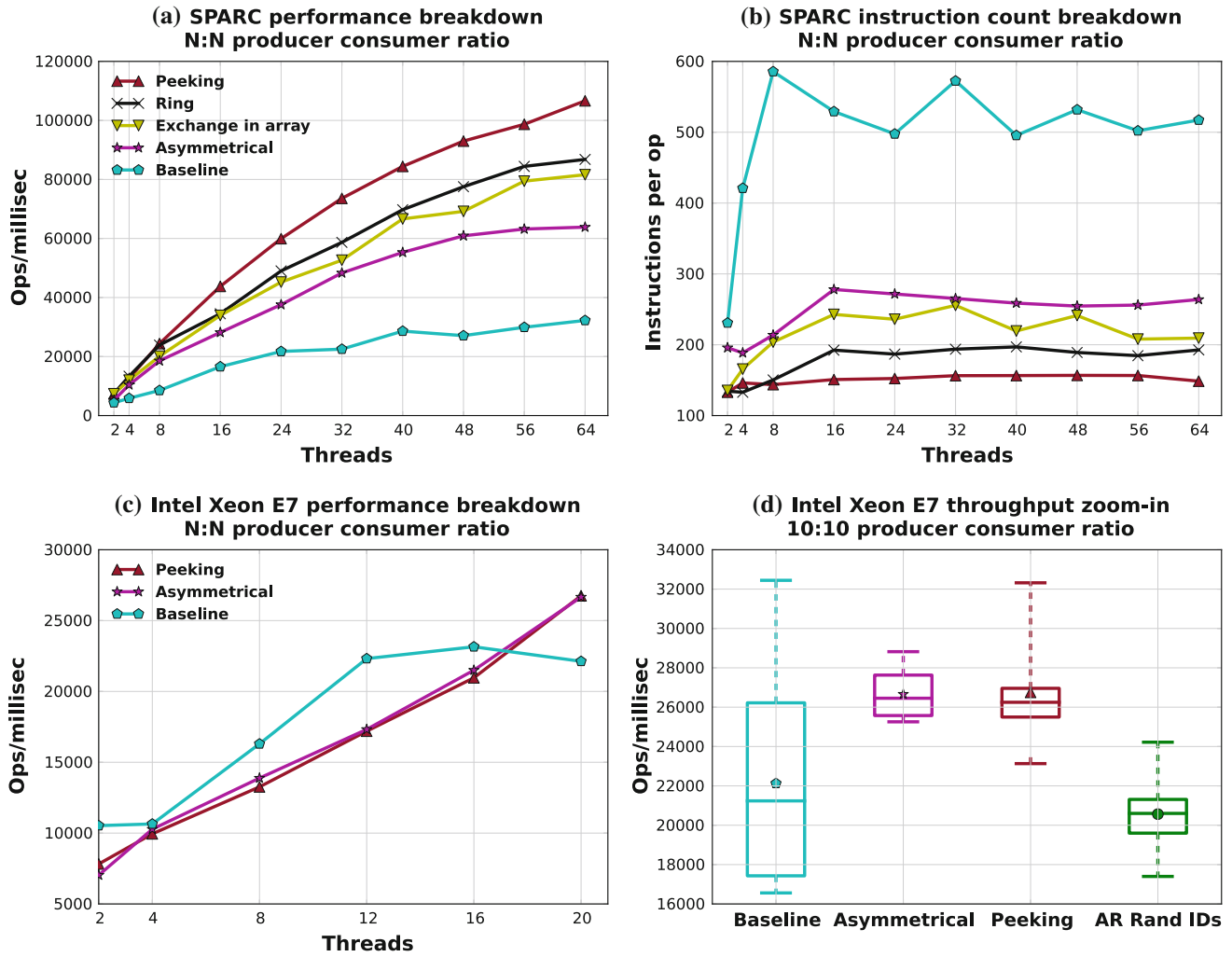


**Fig. 14** Rendezvousing between producer/consumer pairs: contribution of each technique

*6.1.4 AdaptiveAR performance breakdown*

In this section (and Fig. 14) we quantify how much each of the techniques used in AdaptiveAR improves performance. For the purpose of this test (and only in this section), we start with the Exchanger-based collision array implementation, referred to as **Baseline**, and introduce our techniques one by one (each on top of all the previous ones), eventually obtaining a new implementation of AdaptiveAR from a different code base (as opposed to our from-scratch implementation of AdaptiveAR). This methodology also provides further assurance that AdaptiveAR's performance advantages are due to our ideas and not implementation artifacts. To focus solely on rendezvousing performance we disable the algorithms' adaptivity schemes and use an array/ring whose size equals the number of producer/consumer pairs taking part in the test. Our modifications to **Baseline** yield the following versions:

– **Asymmetrical:** Distinguishes between producers and consumers. A consumer thread attempts to occupy a slot and then waits for a partner. A producer thread searches the array for a waiting consumer.
– **Exchange in array:** In the Exchanger algorithm a thread captures a slot using CAS to change a pointer field in the slot to point to a private memory location on which the thread spins. A thread finding an occupied slot attempts to nullify this pointer using a CAS and, if successful, to rendezvous at the private location. Here we change the protocol so that rendezvous occurs in the array slot.
– **Ring:** In the Exchanger a thread maintains an index to traverse the array. Here we turn the array into a ring and traverse it using pointers.
– **Peeking:** Implements peeking (Sect. 4.2). This version is algorithmically identical to **AR**, our from-scratch AdaptiveAR implementation with adaptivity disabled as in Sect. 4.2.

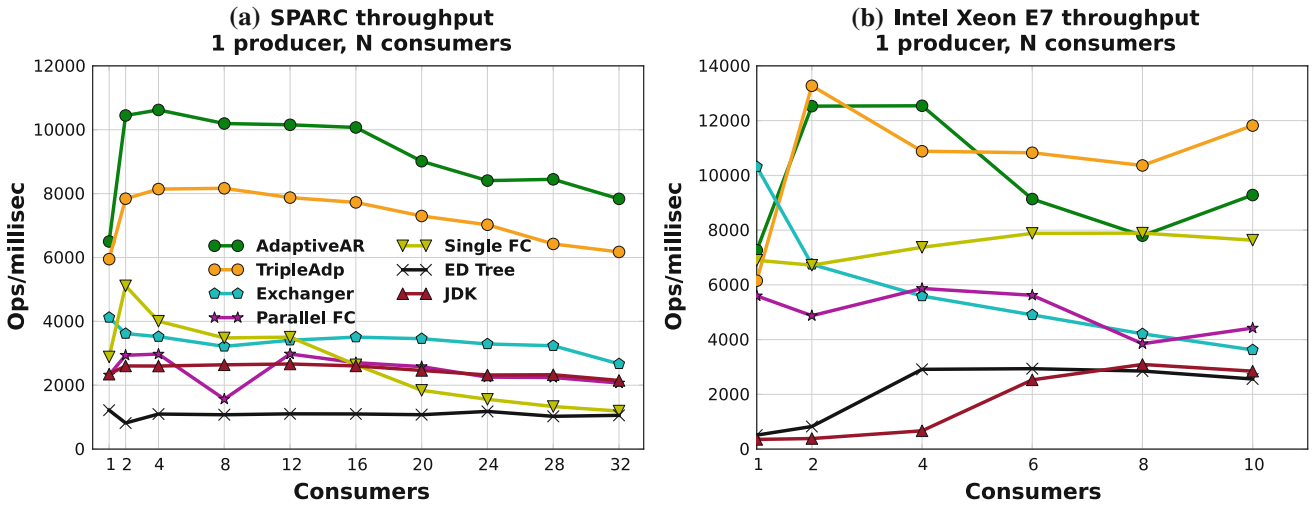


**Fig. 15** Single producer and $N$ consumers rendezvousing

Figure 14a shows the throughput results on the SPARC. **Asymmetrical** obliterates false matches as evidenced by the reduced instruction count shown in Fig. 14b. As a result, throughput increases by 2×. **Exchange in the array** adds up to 30 % by simplifying the code and reducing the number of CASes. Traversing pointers (**Ring**) saves a conditional on each iteration and yields a 5 % gain. **Peeking** gives a final 20 % boost. On the Intel (Fig. 14c) processor **Asymmetrical** serves to stabilize the algorithm's performance (Fig. 14d). Because the processor is an out-of-order processor with 2× the clock rate of the SPARC the cost of cache misses dominates and the remaining techniques have less impact.

### 6.2 Asymmetric workloads

The throughput of one producer rendezvousing with a varying number of consumers is presented in Fig. 15a (SPARC) and b (Intel Westmere EX). We focus on the SPARC results as the results from the Intel processor are similar. Since the throughput is bounded by the rate of a single producer, little scaling is expected and observed. However, for AdaptiveAR and TripleAdp it takes several consumers to keep up with a single producer. This is because the producer hands off an item and completes, whereas the consumer needs to notice the rendezvous has occurred. And while the single consumer is thus busy the producer cannot make progress on a new rendezvous. However, when more consumers are active, the producer can immediately return and find another (different) consumer ready to rendezvous with. Figure 15a shows that most algorithms do not sustain their peak throughput. Exchanger's throughput degrades by 50 % from its peak to 32 consumers; on the Intel the degradation is a much more severe 2.8×. The FC channels have the worst degradation on the SPARC: 4× for Single FC and 1.4× for Parallel FC. JDK's degradation is minimal (20 % from peak to 32 consumers), and it along with the ED tree achieves close to peak throughput even at high thread counts. Yet this throughput is low: AdaptiveAR outperforms JDK by up to 3.6× and the ED tree by 5× to 8× despite degrading by 33 % from peak throughput. As in the $N$:$N$ workload, TripleAdp pays a 20 % throughput penalty compared to AdaptiveAR on the SPARC and achieves comparable performance on the Intel processor. Both TripleAdp and AdaptiveAR maintain peak throughput up to 16 threads before experiencing some degradation.

Why do AdaptiveAR and TripleAdp degrade at all? After all, the producer has its pick of consumers in the ring and should be able to complete a hand-off immediately. The reason for this degradation is not algorithmic but due to *contention on chip resources*. A Niagara II core has two pipelines, each shared by four hardware strands. Thus, beyond 16 threads some threads must share a pipeline—and our algorithm indeed starts to degrade at 16 threads. To prove that this is the problem, we present in Fig. 17a results when the producer runs on the first core and consumers are scheduled only on the remaining seven cores. While the trends of the other algorithms are unaffected, our algorithms now maintains peak throughput through all consumer counts.

Results from the opposite workload, where multiple producers try to serve a single consumer, are given in Fig. 16a (SPARC) and b (Intel Westmere EX). Here the producers contend over the single consumer node and as a result AdaptiveAR's throughput degrades as the number of producers increases, as do the Exchanger and FC channels. Despite this degradation, on the SPARC AdaptiveAR outperforms the FC channels by 2× −2.5× and outperforms the Exchanger up to 8 producers and the JDK up to 48 producers (falling behind by 35 and 15 %, respectively, at 32 producers).

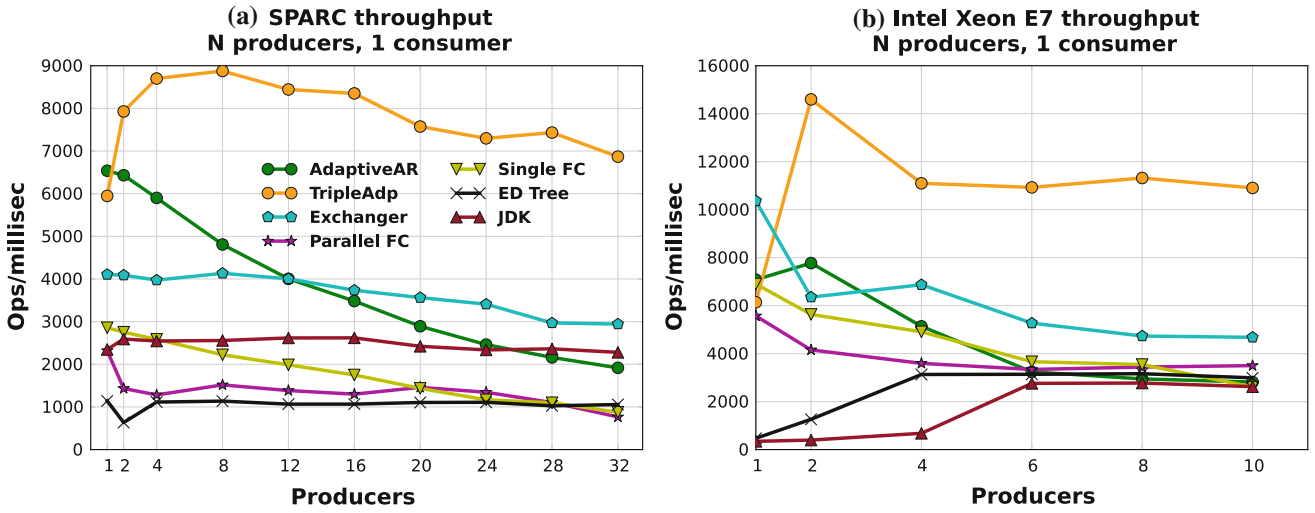


**Fig. 16** Rendezvousing between $N$ producers and a single consumer

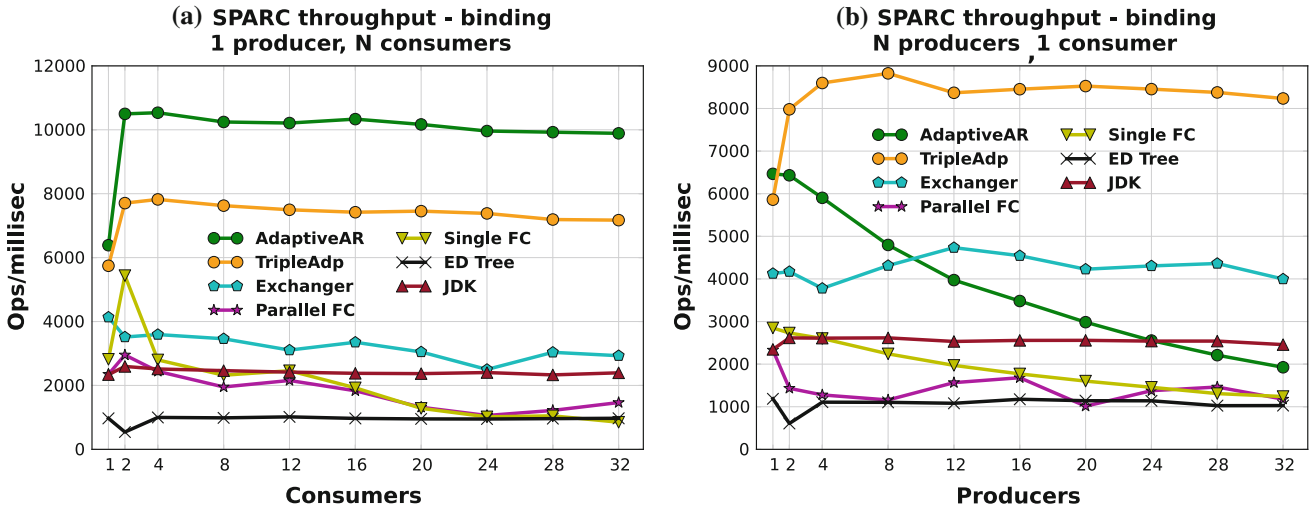


**Fig. 17** Asymmetric rendezvousing (single producer and $N$ consumers, single consumer and $N$ producers) with OS constrained to place the single producer (or consumer) alone on a core, so that threads performing the reverse operation do not interfere with its execution

The advantage of TripleAdp becomes apparent in this workload. TripleAdp maintains its peak throughput (modulo chip resource contention, as Fig. 17b shows), outperforming all other algorithms by at least 2× on both the SPARC and Intel Xeon E7 architectures. The 20 % penalty in the 1 : $N$ workload was for this benefit.

### 6.3 Oversubscribed workloads

Here we explore the algorithms' performance in *oversubscribed* scenarios, in which the number of threads exceeds the number of hardware execution strands and forces the operating system to context switch between the threads.

Figure 19 shows the results when the number of producers in the system equals the number of consumers. For reference, the first point in the plots shows the performance when the number of threads equals the number of hardware threads. The JDK and ED tree show little degradation from this point. Both FC variants experience a sharp drop in throughput as concurrency grows because the odds for the combiner threads being descheduled by the operating system and causing the algorithm to block increase. On the SPARC, AdaptiveAR's and TripleAdp's throughput degrades from the starting point with no oversubscription (2.3× for AdaptiveAR and 1.67× for TripleAdp).

This degradation occurs because when the operating system switches out a thread waiting at some ring node, that node remains captured until the thread runs again. Even if a match occurs in the mean time, that thread must wake up to mark the node as `FREE`. The Exchanger does not show a similar degradation because its exchanges are done outside of the nodes. Recall that in the Exchanger a captured node

points to a private memory location where the waiting thread spins. A thread encountering a captured node nullifies the node pointer using CAS before proceeding to rendezvous at the private location. Thus, a captured Exchanger node can be freed even if its waiting thread is not running.

Despite their throughput degradation, AdaptiveAR and TripleAdp obtain the best throughput results and outperform the Exchanger, which is the next best performer, by 1.55×. On the Intel the throughput of AdaptiveAR, TripleAdp and the Exchanger remain stable, with the Exchanger outperforming AdaptiveAR by 56 % and TripleAdp by 26 %.

Figure 20 depicts throughput results for asymmetric oversubscribed workloads, where the threads performing one type of operation outnumber the threads invoking the reverse operation by a factor of three. The performance trends follow those of the $N : N$ case, with the exception that TripleAdp outperforms AdaptiveAR on the Intel by adapting its ring structure to the scenario with more consumers than producers.

Finally, Fig. 18 shows the throughput results in an oversubscribed bursty workload on the SPARC machine. For 10 s the workload alternates every second between 256 thread pairs and 8 pairs. TripleAdp, AdaptiveAR and the Exchanger are the best performers, achieving orders of magnitude better throughput than the other algorithms. TripleAdp outperforms the Exchanger by 1.88× and AdaptiveAR by 17 %. TripleAdp outperforms AdaptiveAR because TripleAdp maximal ring size is not bounded, allowing its ring to grow and adapt itself to the high concurrency level in this test.

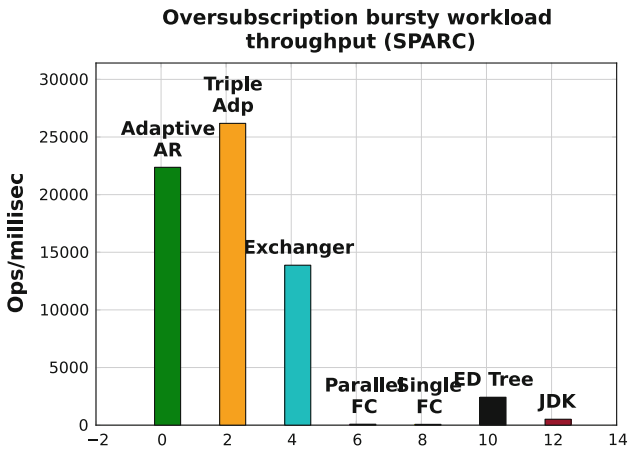


**Fig. 18** Throughput of synchronous queue bursty $N : N$ workload in an oversubscribed setting. Workload alternates every second between 512 threads (256 pairs) and 16 threads (8 pairs)

## 7 Concurrent stack

Hendler et al. improved the performance and scalability of a Treiber-style lock-free stack [24] by adding an *elimination layer* as a backoff scheme for the lock-free stack [9]. In their algorithm an arriving thread that fails to complete its operation (due to contention in the main stack) enters a collision array where it hopes to collide with an operation of the opposite type. If such a match occurs, the resulting `Push()`/`Pop()` operation pair can be linearized next to each other without affecting the state of the stack. Thus, the collision array implements a form of asymmetric rendezvous.

In this section we use AdaptiveAR as an elimination layer on top of a Treiber-style stack. This requires extending AdaptiveAR with support for *aborts*: the ability to give up on a pending rendezvous operation after a certain period of time. Without aborts a single thread might remain in the elimination layer forever, making the resulting stack algorithm blocking.

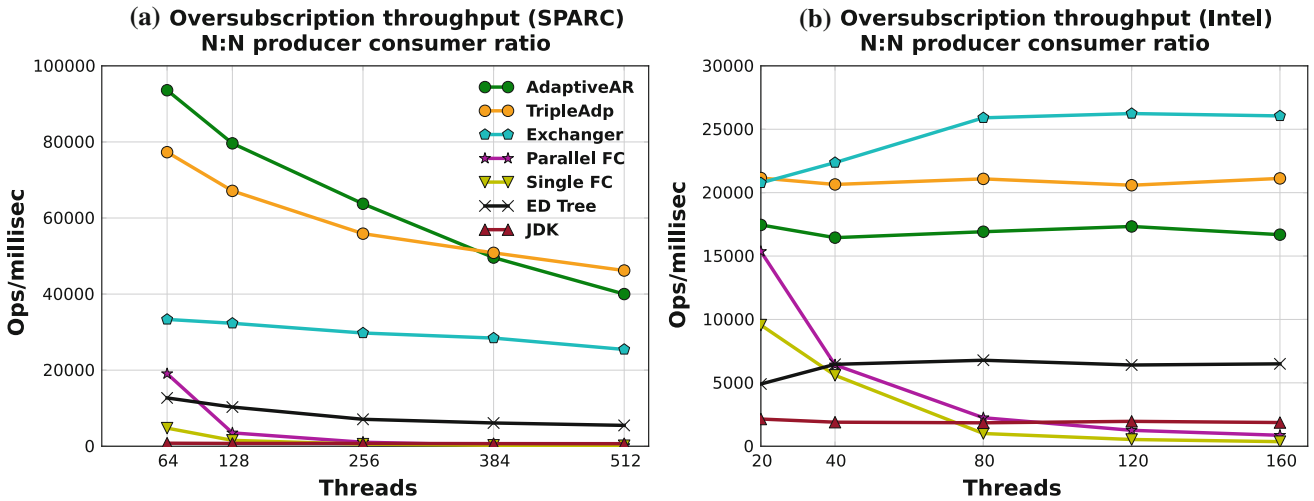


**Fig. 19** Oversubscribed (more threads than available hardware threads) symmetric workload. Total number of threads grows, starting from the number of available threads in the hardware

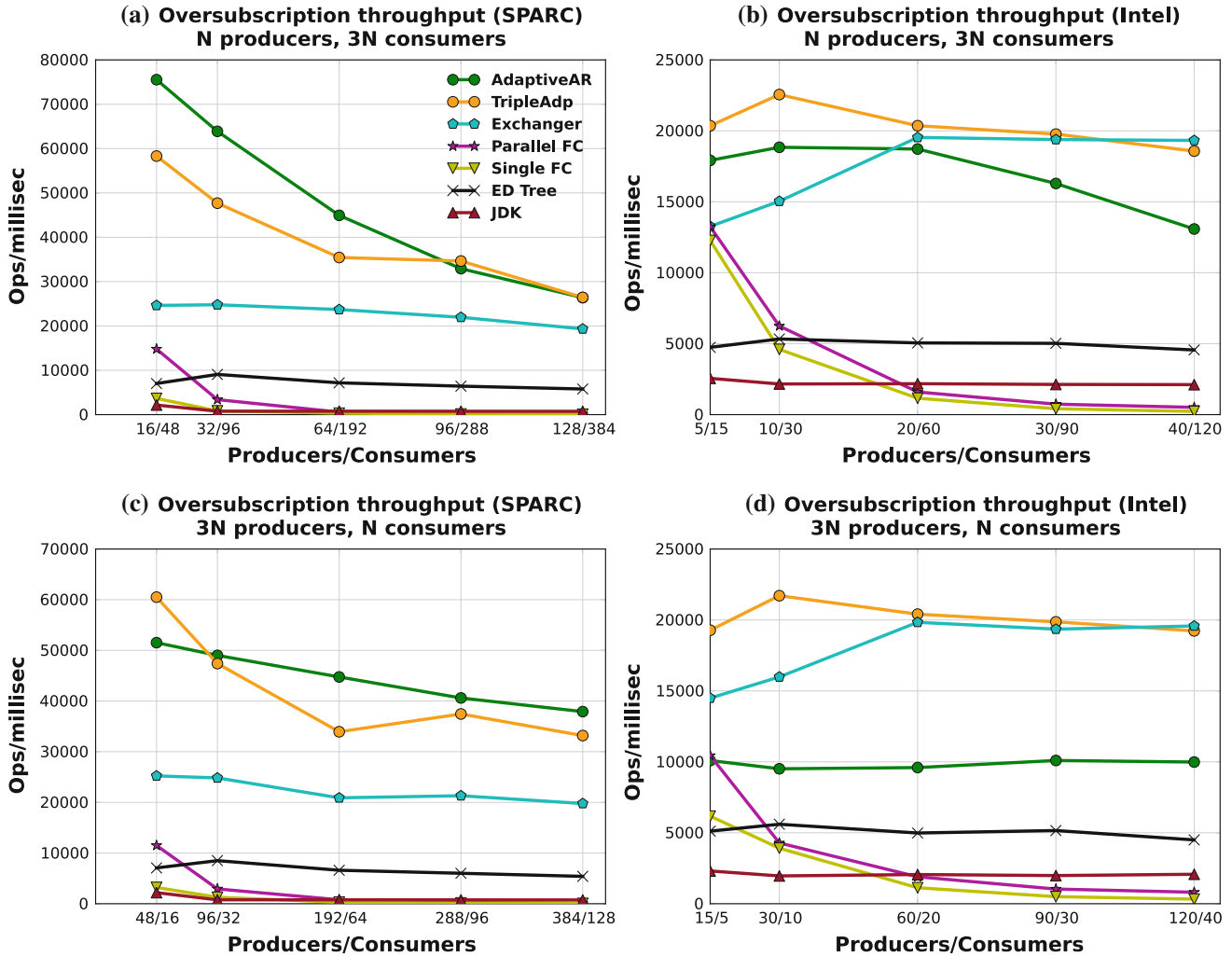


**Fig. 20** Oversubscribed (more threads than available hardware threads) asymmetric worklads. Total number of threads grows, starting from the number of available threads in the hardware

*Abort semantics:* We change the specification of the hand-off object as follows. Both `put()` and `get()` operations can nondeterministically return a reserved value $\perp$ indicating an aborted operation, in which case the state of the object is not changed. Note that since there is no notion of time in our model, we do not define when it is considered legal to time-out and abort a rendezvous operation. This allows for trivial rendezvous implementations that always return $\perp$, though they would be of little use in practice.

*Abort implementation:* We add a parameter to the algorithm specifying the desired timeout. Timeout expiry is then checked in each iteration of the main loop in `put()`, `get()` and `findFreeNode()` (Fig. 2c). If the timeout expires, a producer or a consumer in `findFreeNode()` aborts by returning $\perp$. A consumer that has captured a ring node cannot abort before freeing the node by CASing its `item` 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 $\perp$.

We consider two variants of an elimination stack, where we abstract the elimination task to an abortable rendezvous object. The first variant (Fig. 21a) uses elimination as a back-off scheme: threads try to eliminate upon detecting contention on the main stack. (This is essentially Hendler et al.'s elimination stack [9]). The second variant (Fig. 21b) is *optimistic*: a thread enters the elimination layer *first*, accessing the main stack only if it fails to find a match. If it then encounters contention on the main stack it goes back to try the rendezvous, and so on.

We implemented these variants in C++ within the benchmark framework of Hendler et al. [7]. We tested each variant with two algorithms for performing elimination: AdaptiveAR and a collision array. Since code for the original collision array implementation [9] is not available we chose to port Java's optimized Exchanger to C++ for use as the collision

```
shared objects:
  rendezvous : either our algorithm or a collision array

push(item x) {
  while (true) {
    // Perform Treiber stack code first
    if ( successfully CAS x to head of linked list )
      return
    // Fall back to elimination
    if (rendezvous.put(x) != ⊥)
      return
  }
}

pop() {
  while (true) {
    // Perform Treiber stack code first
    if (head of list is null)
      return EMPTY
    if ( successfully remove x, the head of list , using CAS)
      return x
    // Fall back to elimination
    x := rendezvous.get()
    if (x != ⊥)
      return x
  }
}
```

**(a)** Backoff elimination stack [9]

```
shared objects:
  rendezvous : either our algorithm or a collision array

push(item x) {
  while (true) {
    // Try to eliminate first
    if (rendezvous.put(x) != ⊥)
      return
    // Fall back to Treiber stack
    if ( successfully CAS x to head of linked list )
      return
  }
}

pop() {
  while (true) {
    // Try to eliminate first
    x := rendezvous.get()
    if (x != ⊥)
      return x
    // Fall back to Treiber stack
    if (head of list is null)
      return EMPTY
    if ( successfully remove x, the head of list , using CAS)
      return x
  }
}
```

**(b)** Optimistic elimination stack

**Fig. 21** Elimination stack variants. We omit the linked list manipulation details of the Treiber lock-free stack [24]

array. We set AdaptiveAR's wait threshold to a value smaller than the abort timeout, for otherwise ring size decreases never occur as the thread always aborts first.

We compared these implementations (referred to as **Backoff elim (AdaptiveAR/Exchanger)** and **Optimistic elim (AdaptiveAR/Exchanger)** in the plots) to the implementations of Treiber's lock-free stack (**Lock-free**) and the FC based stack [7] (**Flat combining**) in the framework.

We report results of a benchmark measuring the throughput of an even `push`/`pop` operation mix. Each test consists of threads repeatedly invoking stack operations over a period of 10 s. Each thread performs both `push()` and `pop()` operations; this is the same benchmark used in [7].[6] In the Intel test each thread performs a small random number ($\leq 64$) of empty loop iterations between completing one stack operation and invoking the next one. We found this delay necessary to avoid *long runs* [15] where a thread holding the top of the stack in its cache quickly performs a long sequence of operations, leading to unrealistically high throughput. On the SPARC the long run pathology does not occur because all `CAS`es are performed in the shared L2 cache, so we did not add delays in the SPARC test.

Figure 22 shows the results of the benchmark. The backoff variants fail to scale in all workloads and machine combinations, illustrating the price paid for accessing the main stack: both the overhead of allocating a node to push on the stack as well as trying (and failing) to `CAS` the stack's top to point to the node. However, AdaptiveAR's backoff stack achieves 3.5× the throughput of the Exchanger backoff stack on the SPARC by avoiding false matches.

The lock-free stack has no elimination to fall back on and does not scale at all. Consequentially, the optimistic AdaptiveAR elimination stack outperforms it by 33× on the SPARC and by 4.8× on the Intel. However, the optimistic algorithms achieve this scalability at the expense of lower throughput under low thread counts. The backoff AdaptiveAR elimination stack achieves 3× the throughput of the optimistic AdaptiveAR variant with 2 threads on the Intel, and 1.65× on the SPARC.

The performance trends of the optimistic elimination stacks match those observed in Sect. 6 on the SPARC. The optimistic AdaptiveAR elimination stack obtains the highest throughput. It outperforms the backoff Exchanger elimination stack by 7.5×, the optimistic Exchanger elimination stack by 2.1×, and the FC stack by 3.6×. On the Intel, the optimistic Exchanger and AdaptiveAR elimination stacks show different scaling trends. The optimistic Exchanger stack is the best performer at 8 to 16 threads, however its throughput flattens beyond 12 threads. In contrast, the optimistic AdaptiveAR stack scales up to 20 threads, where it outperforms the Exchanger stack by 1.25×.

This behavior occurs because at low concurrency levels, entering the collision array and failing to eliminate provides sufficient backoff to allow threads to complete their

[6] We reduced the overhead due to memory allocation in the original implementations [7] by caching objects popped from the stack and using them in future push operations.

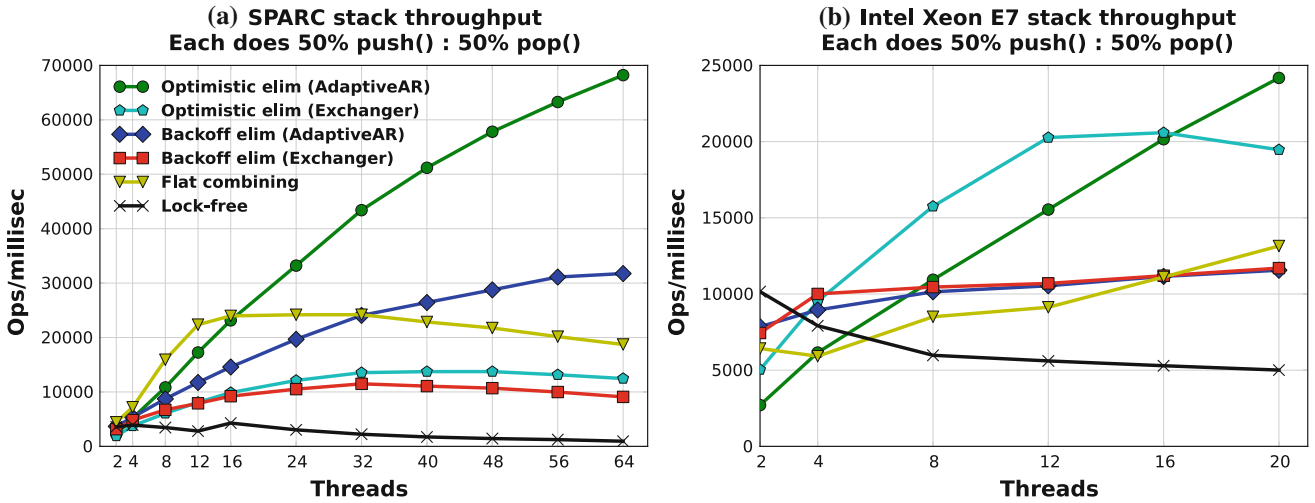


**Fig. 22** Stack throughput. Each thread performs both `push` and `pop` operations with probability 1/2 for each operation type

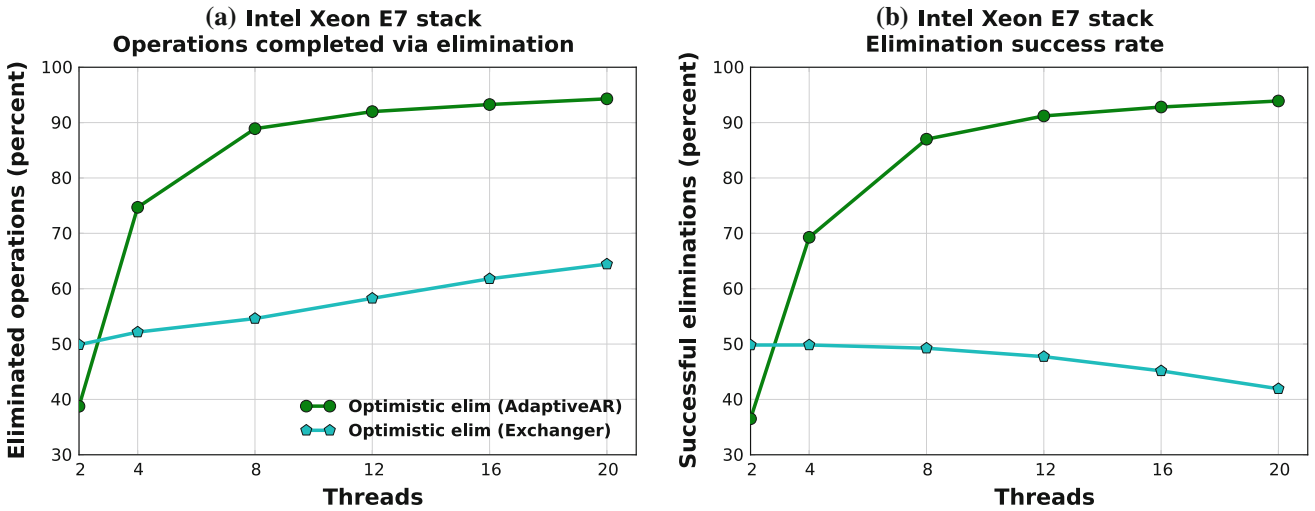


**Fig. 23** Analysis of stack elimination on Intel Xeon E7. *Left* Percent of stack operations that complete through elimination. *Right* a stack operation may complete through elimination only after several rounds of failing to eliminate and going back to the main stack. Thus, we show the percent of successful elimination attempts (resulting in completion of the stack operation) out of all elimination attempts, which outnumber the stack operations

operations quickly. We show this by plotting the number of optimistic stack operations that complete through elimination in this benchmark (Fig. 23a). Even as concurrency increases, the Exchanger elimination stack completes about 40–50 % of the operations on the main stack. The reason, as Fig. 23b shows, is that half of the Exchanger matches are false. Threads that fail to eliminate go back to the stack and—because concurrency is low—often manage to quickly complete their operation there. In contrast, AdaptiveAR obtains almost 100 % elimination success rate. It pays a price in the low concurrency levels, where threads wait longer until a partner comes along, leading to lower throughput.

## 8 Conclusion

This paper focused on the rendezvous problem, an abstract problem of high throughput concurrent matching, in which consumers and producers show up and are matched each with a unique thread of the other type. We have presented adaptive, nonblocking, high throughput asymmetric rendezvous systems that scale well under symmetric workloads and maintains peak throughput in asymmetric workloads. This is achieved by a careful marriage of new algorithmic ideas and attention to implementation details, to squeeze all available cycles out of the processors.

Our work shows the degree of impact that hardware architecture specific details can have on a concurrent algorithm's performance. For instance, the Intel processor's write-back cache architecture leads to significant cost differences between CASing a location in a core's cache compared to one in a remote core's cache, whereas on the SPARC with its write-through cache architecture the CAS cost is more uniform. This suggests that algorithms must be fully aware of their underlying hardware to maximize their performance. Yet much of the programming world is moving toward managed languages, such as Java, whose goal is to abstract away such architectural details and avoid exposing them to the program. Reconciling these conflicting directions is a very interesting research problem.

Finally, our results raise a question about the flat combining paradigm. Flat combining has a clear advantage in inherently sequential data structures, such as a FIFO or priority queue, whose concurrent implementations have central hot spots. But as we have shown FC may lose its advantage in problems with inherent potential for parallelism. It is therefore interesting whether the FC technique can be improved to match the performance of our asymmetric rendezvous systems.

## Availability

Our implementation is available from Tel Aviv University's Multicore Computing Group web site at http://mcg.cs.tau.ac.il/.

**Acknowledgments** We are grateful to Hillel Avni, Nir Shavit and the anonymous reviewers, whose comments and suggestions helped to considerably improve this paper.

## References

1. Afek, Y., Korland, G., Natanzon, M., Shavit, N.: Scalable producer-consumer pools based on elimination-diffraction trees. In: Euro-Par 2010—Parallel Processing, vol. 6272 of LNCS, pp. 151–162. Springer, Berlin, Heidelberg (2010)
2. Andrews, G.R.: Concurrent Programming: Principles and Practice. Benjamin-Cummings Publishing Co, Redwood City (1991)
3. Berger, E.D., McKinley, K.S., Blumofe, R.D., Wilson, P.R.: Hoard: a scalable memory allocator for multithreaded applications. SIGARCH Comput. Archit. News **28**(5), 117–128 (2000)
4. Fatourou, P., Kallimanis, N.D.: A highly-efficient wait-free universal construction. In: Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2011, pp. 325–334. ACM, New York, NY, USA (2011)
5. Fatourou, P., Kallimanis, N.D.: Revisiting the combining synchronization technique. In: Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '12, pp. 257–266. ACM, New York, NY, USA, (2012)
6. Hanson, D.R.: C Interfaces and Implementations: Techniques for Creating Reusable Software. Addison-Wesley Longman Publishing, Boston (1996)
7. Hendler, D., Incze, I., Shavit, N., Tzafrir, M.: Flat combining and the synchronization-parallelism tradeoff. In: Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2010, pp. 355–364. ACM, New York, NY, USA (2010)
8. Hendler, D., Incze, I., Shavit, N., Tzafrir, M.: Scalable flat-combining based synchronous queues. In: Proceedings of the 24th International Symposium on Distributed Computing (DISC 2010), vol. 6343 of LNCS, pp. 79–93. Springer, Berlin, Heidelberg (2010)
9. Hendler, D., Shavit, N., Yerushalmi, L.: A scalable lock-free stack algorithm. J. Parallel Distrib. Comput. **70**(1), 1–12 (2010). doi:10.1016/j.jpdc.2009.08.011
10. Herlihy, M.: Wait-free synchronization. ACM Trans. Program. Lang. Syst. (TOPLAS) **13**, 124–149 (1991)
11. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. (TOPLAS) **12**, 463–492 (1990)
12. Lea, D., Scherer, W.N. III, Scott, M.L.: 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. Merritt, M., Taubenfeld, G.: Computing with infinitely many processes. In: Proceedings of the 14th International Symposium on Distributed Computing (DISC 2000), vol. 1914 of LNCS, pp. 164–178. Springer, Berlin, Heidelberg (2000)
14. Michael, M.M.: Hazard pointers: safe memory reclamation for lock-free objects. IEEE Trans. Parallel Distrib. Syst. **15**(6), 491–504 (2004)
15. Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing, PODC '96, pp. 267–275. ACM, New York, NY, USA (1996)
16. Moir, M., Nussbaum, D., Shalev, O., Shavit, N.: Using elimination to implement scalable and lock-free fifo queues. In Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2005, pp. 253–262. ACM, New York, NY, USA (2005)
17. Scherer, W.N., III, Lea, D., Scott, M.L.: Scalable synchronous queues. In Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2006, pp. 147–156. ACM, New York, NY, USA (2006)
18. Scherer, W.N. III, Lea, D., Scott, M.L.: A scalable elimination-based exchange channel. In: Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL 2005) October (2005)
19. Scherer, W.N. III, Scott, M.L.: Nonblocking concurrent data structures with condition synchronization. In: Proceedings of the 18th International Symposium on Distributed Computing (DISC 2004), vol. 3274 of LNCS, pp. 174–187. Springer, Berlin/Heidelberg (2004)
20. Shavit, N., Touitou, D.: Elimination trees and the construction of pools and stacks. Theory Comput. Syst. **30**(6), 645–670 (1997). doi:10.1007/s002240000072
21. Shavit, N., Zemach, A.: Diffracting trees. ACM Trans. Comput. Syst. (TOCS) **14**, 385–428 (1996)
22. Shavit, N., Zemach, A.: Combining funnels: a dynamic approach to software combining. J. Parallel Distrib. Comput. **60**(11), 1355–1387 (2000)

23. Tang, L., Mars, J., Vachharajani, N., Hundt, R., Soffa, M.L.: The impact of memory subsystem resource sharing on datacenter applications. In: Proceedings of the 37th Annual International Symposium on Computer Architecture, ISCA '11, ACM, New York, NY, USA (2011)

24. Treiber, R.K.: Systems programming: coping with parallelism. Technical Report RJ5118, IBM Almaden Research Center (2006)