Fast Concurrent Queues for x86 Processors

Adam Morrison Yehuda Afek

Blavatnik School of Computer Science, Tel Aviv University

Abstract

Conventional wisdom in designing concurrent data structures is to use the most powerful synchronization primitive, namely `compare-and-swap` (CAS), and to avoid contended hot spots. In building concurrent FIFO queues, this reasoning has led researchers to propose *combining-based* concurrent queues.

This paper takes a different approach, showing how to rely on `fetch-and-add` (F&A), a less powerful primitive that is available on x86 processors, to construct a *nonblocking (lock-free) linearizable concurrent FIFO queue* which, despite the F&A being a contended hot spot, outperforms combining-based implementations by $1.5\times$ to $2.5\times$ in all concurrency levels on an x86 server with four multicore processors, in both single-processor and multi-processor executions.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; E.1 [*Data Structures*]: Lists, stacks, and queues

Keywords concurrent queue, nonblocking algorithm, fetch-and-add

1. Introduction

Avoiding *contended hot spots* is a fundamental principle in the design of concurrent algorithms [13]. The concurrent FIFO queue, a fundamental and commonly used data structure, is a prime example of this principle in action: Both of Michael and Scott's classic algorithms [19], one lock-based and one nonblocking, do not scale past a small amount of concurrency because threads *contend* on the queue's tail and head [11, 13]. To get around this seemingly inherent bottleneck, researchers have recently applied *combining* approaches in which one thread gathers pending operations of other threads and executes them on their behalf [7, 8, 11].

Most non-combining concurrent algorithms synchronize using `compare-and-swap` (CAS) loops: a thread observes the shared state, performs a computation, and uses CAS to update the shared state. If the CAS succeeds, this read-compute-update sequence appears to be atomic; otherwise the thread must retry. Essentially, the idea behind combining is that the synchronization cost of a contended CAS hot spot (due to cache coherency traffic on the contended location) is so large that performing all the work *serially*, to save synchronization, performs better [11]. In this paper we show that the truth is more nuanced: it is *work wasted due to CAS failures* that largely causes the poor performance of algorithms with a CAS hot spot, not just the synchronization cost.

PPoPP'13, February 23–27, 2013, Shenzhen, China.

	`compare-and-swap`	`swap`	`test-and-set`	`fetch-and-add`
ARM	LL/SC	deprecated	no	no
POWER	LL/SC	no	no	no
SPARC	yes	deprecated	yes	no
x86	**yes**	**yes**	**yes**	**yes**

Table 1: Synchronization primitives supported as machine instructions on dominant multicore architectures.

Observing this distinction, let alone exploiting it, is not possible on most commercial multicore architectures which only support the *universal* primitives CAS or `load-linked/store-conditional` (LL/SC). While in theory these can implement weaker primitives in a wait-free manner [12], such implementations are heavyweight and in practice vendors direct programmers to use CAS loops [1]. However, there is an interesting exception: the (64 bit) x86 architecture, which dominates the server and desktop markets, directly supports various theoretically weaker primitives whose crucial property for our purpose is that *they always succeed* (see Table 1).

Consider, for example, the `fetch-and-add` (F&A) primitive. Figure 1 shows the difference in the time it takes a thread to increment a contended counter on a modern x86 system when using F&A vs. a CAS loop. Avoiding the retries inherent and paying only the synchronization price leads to a $4\times-6\times$ performance improvement. In this paper we transfer this insight to the domain of FIFO queues, henceforth simply queues.

Our contribution We present **LCRQ**, a *linearizable nonblocking FIFO queue* that uses contended F&A objects to spread threads among items in the queue, allowing them to enqueue and dequeue quickly and in parallel, in contrast the inherently serial behavior of combining-based approaches. As a result, LCRQ outperforms prior queue implementations by $1.5\times$ to $2.5\times$ on a system with four Intel Xeon E7-4870 multicore processors, both on single-processor and on multi-processor executions. Because LCRQ is nonblocking, it maintains its performance in oversubscribed scenarios in which there are more threads than available hardware threads. In such workloads it outperforms by more than $20\times$ lock-based combining queues, which cannot make progress if a combiner gets scheduled out.

Our LCRQ algorithm is essentially a Michael and Scott linked list queue [19] in which a node is a concurrent ring (cyclic array) queue, CRQ for short. A CRQ that fills up becomes *closed* to further enqueues, who instead append a new CRQ to the list and begin working in it. Most of the activity in the LCRQ occurs in the individual CRQs, making contention on the list's head and tail a non-issue. Within a CRQ, the head and tail are F&A objects which are used to spread threads around the slots of the ring, where they synchronize using (uncontended in the common case) CAS.

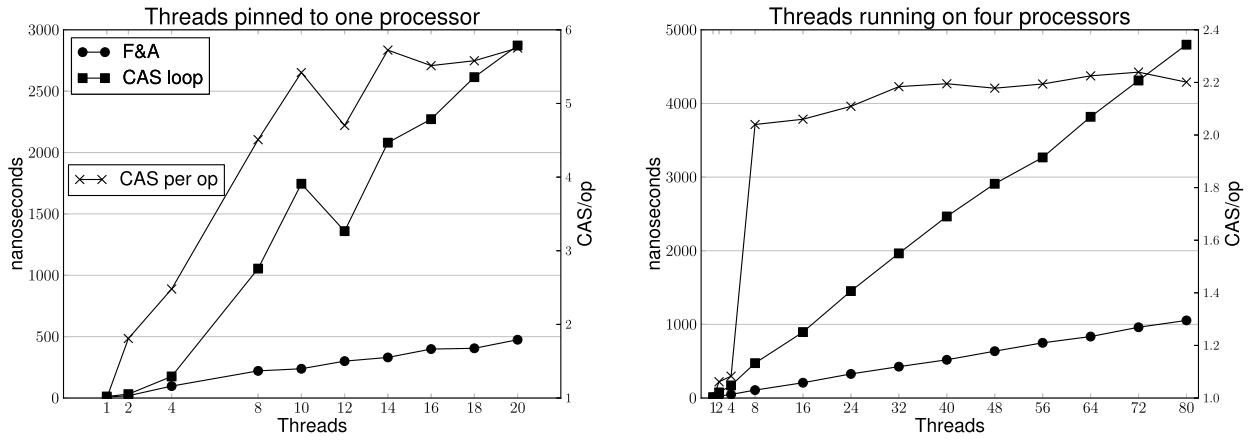

Figure 1: Time to increment a contended counter on a system with four Intel Xeon E7-4870 (Westmere EX) processors, each of which has 10 2.40 GHz cores that multiplex 2 hardware threads. The right vertical axis shows the number of CAS it takes to complete an increment.

One of the CRQ's distinctive properties compared to prior concurrent circular array queues [2–4, 9, 10, 22, 23] is that in the *common case* an operation accesses only the CRQ's head or tail but not both. This reduces the CRQ's synchronization cost by a factor of two, since the contended head and tail are the algorithm's bottleneck.

2. Related work

We refer the reader to Michael and Scott's extensive survey [19] for discussion of additional work that predates theirs.

List based queues Michael and Scott present two linked list queues, one nonblocking (henceforth MS queue) and one lock-based [19]. However, due to contention on the queue's head and tail, their algorithms do not scale past a low level of concurrency [7, 11]. Kogan and Petrank introduce a *wait-free* variant of the MS queue with similar performance characteristics [16]. Several works attempt to improve the MS queue's scalability, however all these still suffer from the CAS retry problem [15, 17, 20].

Cyclic array queues Prior concurrent cyclic array queues are *bounded* and can contain a fixed number of items. One of the challenges in these algorithms is correctly determining when the queue is full and empty. The queues of Gottlieb et al. [10] and of Freudenthal and Gottlieb [9] maintain a *size* counter that is updated using F&A. Such a F&A might bring the queue into an inconsistent state (e.g., $size < 0$) and the algorithm then tries to recover using a compensating F&A. Still, the inconsistent states make these queues non-linearizable [1]. Blelloch et al. [3] use *room synchronization*, which prevents enqueues from running concurrently to dequeues, to construct a queue that is linearizable despite temporarily entering inconsistent states when its head/tail are updated using F&A. Another queue by Blelloch et al. [2] avoids inconsistent states of the head and tail by updating these indices using hardware *memory block transactions* which are not supported by commercial hardware. Tsigas and Zhang [23], Colvin and Groves [4] and Shafiei [22] present cyclic array queues that avoid inconsistent head and tail states by performing the updates using CAS, but are therefore prone to the CAS failure effect.

In contrast to these prior designs, LCRQ is an unbounded queue formed by linking CRQs (array queues) in a list, with a new CRQ added when an enqueue operation fails to make progress. The ability to *close* a CRQ, forcing enqueues to move to the next CRQ in the list, makes LCRQ nonblocking whereas prior F&A-based designs [2, 3, 9, 10] are blocking. In addition, since we do not need to determine when the queue is full in a linearizable way, we can recover from inconsistent states that result from using F&A for head/tail updates without compromising linearizability. Performance-wise, a CRQ operation accesses only one end of the queue in the common case, whereas the operations in the previous designs access both the head and tail indices.

Combining Researchers have recently shown that *combining-based* queues scale better than CAS-based list queues [7, 8, 11]. A combining algorithm is essentially a *universal construction* [12] that can implement any shared object. The idea is that a single thread scans a list of pending operations and applies them to the object. Such algorithms greatly reduce the synchronization cost of accessing the object, at the cost of executing work serially.

Hendler et al. describe a linked list queue based on *flat combining*, a lock-based combining construction [11]. Fatourou and Kallimanis present SimQueue [8], a queue based on a wait-free combining construction, and CC-Queue, a queue based on a blocking combining algorithm [7]. Section 5 details these algorithms.

Both of Fatourou and Kallimanis' algorithms use weak synchronization primitives (F&A and SWAP). However, they do so to reduce the synchronization cost of the combining algorithm, which still needs to perform serial work that is linear in the number of threads. In contrast, we use F&A to enable parallelism in the seemingly inherently sequential FIFO queue.

3. Preliminaries

System model Most concurrent algorithms work assumes a *sequentially consistent* shared memory system, particularly for correctness proofs, as this allows modeling the execution as a sequence of interleaved memory operations performed by the threads. While the x86 memory model is not sequentially consistent, the only difference is that on the x86 a write gets buffered in a *write buffer* before reaching the memory, allowing a read to be satisfied from memory before a write preceding it becomes globally visible [21].

However, in our algorithms threads write to shared data only with atomic operations, such as CAS and F&A. Atomic operations flush the write buffer and are globally ordered [21], allowing us to treat the system as sequentially consistent. Formally, we have a set

[1] Blelloch et al. [3] show a non-linearizable execution for Gottlieb et al.'s queue. A similar scenario applies to Freudenthal and Gottlieb's queue.

of T sequential *threads* that communicate by performing operations on the shared memory, as described below.

Memory operations The memory is an array of locations, each holding a 64-bit value. We use the notation $m[a]$ for the value stored in address a of the memory. Our algorithms use the following primitives supported by the x86 architecture: (1) read(a) which returns $m[a]$, (2) `fetch-and-add`, denoted F&A(a,x), which returns $v = m[a]$ and changes $m[a]$'s value to $v+x$, (3) `swap`, denoted SWAP(a,x), which returns $v = m[a]$ and changes $m[a]$'s value to x, (4) `test-and-set`, denoted T&S(a), which returns $v = m[a]$ and changes $m[a]$'s value to 1, (5) `compare-and-swap`, denoted CAS(a,o,n), which changes $m[a]$'s value to n if $m[a] = o$ and returns TRUE, or returns FALSE otherwise, (6) `compare-and-swap2`, denoted CAS2($a, \langle o_0, o_1 \rangle, \langle n_0, n_1 \rangle$), which changes $m[a]$'s value to n_0 and $m[a+1]$'s value to n_1 if $m[a] = o_0$ and $m[a+1] = o_1$ before returning TRUE, or else returns FALSE [2].

Concurrent objects The threads implement a high-level object defined by a *sequential specification*, a state machine specifying the object's *states* and the *operations* used to transition between the states. Here we are concerned with the *FIFO queue*, an object whose state, Q, is a (possibly empty) sequence of items. It supports an `enqueue(`x`)` operation that appends x to Q and returns OK, and a `dequeue()` operation which removes the first item x from Q and returns x, or returns EMPTY if Q is the empty sequence.

Implementations, executions and linearizability We use the standard definitions of a high-level object implementation and its execution [14]. Our correctness condition is linearizability [14], which (informally) requires that a high-level operation appears to take place at one point in time during its execution interval.

Progress According to Herlihy's now standard definition [12], an implementation is *nonblocking* if it guarantees that some thread completes an operation in a finite number of steps. In other words, an individual operation may starve, but some operation always makes progress. This guarantee still allows some undesirable scenarios for queues, e.g., an execution in which enqueuers are starved by dequeuers returning EMPTY. Nonblocking queues in the literature [4, 19, 22, 23] actually provide a stronger guarantee, which we call *op-wise nonblocking* [3]: some `enqueue()` completes in a finite number of steps by enqueuing threads, and some `dequeue()` completes in a finite number of steps by dequeuing threads.

4. The LCRQ algorithm

LCRQ can be viewed as a practical realization of the following simple but unrealistic queue algorithm (Figure 2). The algorithm represents the queue using an *infinite* array, Q, with (unbounded) *head* and *tail* indices that identify the part of Q which may contain items. Initially, each cell $Q[i]$ is empty and contains a reserved value $\perp$ that may not be enqueued.

An enqueue(x) operation obtains a cell index t via a F&A on *tail*. The enqueue then atomically swaps the value in $Q[t]$ with x. If the swap returns $\perp$, the enqueue operation completes; otherwise, it repeats this process.

A dequeue, D, obtains a cell index h using F&A on *head* and atomically swaps the value in $Q[h]$ with another reserved value $\top$. If $Q[h]$ contained some $x \neq \perp$, D returns x. If D finds $\perp$ in $Q[h]$, the fact that D stored $\top$ in the cell guarantees that an enqueue operation which later stores an item in $Q[h]$ will not complete. D then returns EMPTY if $tail \leq h+1$ ($h+1$ is the value of *head* following D's F&A). If D cannot return EMPTY, it repeats this process.

[2] On the x86 these atomic primitives are known as `LOCK XADD`, `XCHG`, `LOCK BTS`, `LOCK CMPXCHG` and `LOCK CMPXCHG16B`.

[3] We are not aware of this property being explicitly pointed out before.

```
1  enqueue(x : Object) {
2    while (true) {
3      t := F&A(&tail, 1)
4      if ( SWAP(&Q[t], x) = ⊥ ) return OK
5  } }
6  dequeue() {
7    while (true) {
8      h := F&A(&head, 1)
9      x := SWAP(&Q[h], ⊤)
10     if x ≠ ⊥ return x
11     if ( tail ≤ h+1) return EMPTY
12 } }
```

Figure 2: Infinite array queue.

While this algorithm is a linearizable FIFO queue [4] it has two major flaws that prevent it from being relevant in practice: using an infinite array and susceptibility to livelock (a dequeuer continuously swaps $\top$ into the cell an enqueuer is about to access). We obtain the practical LCRQ algorithm by solving these problems.

Our array queue, CRQ, transforms the infinite array to a cyclic array (*ring*) of R nodes. The *head* and *tail* indices still strictly increase, but now the value of an index modulo R specifies the ring node it points to. Since now more than one enqueuer and dequeuer can concurrently access a node, we replace the infinite array queue's SWAP-based exchange with a CAS2-based protocol. This protocol is unique in that, unlike prior work [2, 10], an operation does not have to *wait* for the completion of operations whose F&A returns smaller indices that also point to the same ring node.

The CRQ's crucial performance property is that in the common *fast path*, an operation accesses only *one* F&A. We use the additional synchronization in the ring nodes to detect corner cases, such as an empty queue. Since *head* and *tail* are heavily contended, our approach halves an operation's synchronization cost in the common case. We detail the CRQ algorithm in Section 4.1.

The LCRQ algorithm (Section 4.2) builds on CRQ to prevent the livelock problem. We represent the queue as a *linked list* of CRQs. An enqueue(x) operation failing to make progress in the tail CRQ *closes* it to further enqueues. Upon noticing the tail CRQ is closed, each enqueuer tries to append a new CRQ, initialized to contain its item, to the list. One enqueuer succeeds and completes; the rest move into the new tail CRQ, leaving the old tail CRQ with only dequeuers inside it, which allows the dequeuers to complete. The LCRQ is thus op-wise nonblocking.

4.1 The CRQ algorithm

The pseudocode of the basic CRQ algorithm appears in Figure 3. The CRQ represents the queue as a *ring* (cyclic array) of R nodes, with 64-bit *head* and *tail* indices (Figure 3a). An index with value i points to node i mod R, which we denote by $node(i)$. We reserve the most significant bit of *tail* to denote the CRQ's CLOSED state. We thus make the realistic assumption that both *head* and *tail* do not exceed 2^{63}.

The synchronization protocol in a CRQ ring node needs to handle more cases than the infinite array queue, which only needs to distinguish whether an enqueue or dequeue arrives first at the node. We proceed to describe this protocol and how it handles these cases.

Node structure (Figure 3a) Physically, a ring node contains two 64-bit words. Logically, a ring node is a 3-tuple (s,i,v) consisting of (1) a *safe* bit s (used by a dequeuer to notify the matching enqueuer that storing an item in the node is unsafe as the dequeuer will not be around to dequeue it; we explain the details below), (2) an *index* i, and (3) a *value* v. Initially, node u's state is $(1,u,\perp)$ for every $0 \leq u < R$.

[4] We omit the full proof, which is similar to the proof in Section 4.1.2.

```
13 struct Node {
14   safe : 1 bit (boolean)
15   idx : 63 bits (int)
16   val : 64 bits (int or pointer)
17   // padded to cache line size
18 }
19 struct CRQ { // fields are on distinct cache lines
20   head : 64 bit int
21   tail : struct { closed : 1 bit, t : 63 bits }
22   next : pointer to CRQ, initially null
23   ring : array of R Nodes, initially node u = <1,u,⊥>
24 }
```

(a) Globals

```
25 dequeue(crq : pointer to CRQ) {
26   // local variables
27   val, idx : 64 bit int
28   h, t : 64 bit int
29   node : pointer to Node
30   closed : boolean
31   safe : boolean
32
33   while (true) {
34     h := F&A(&crq.head, 1)
35     node := &crq.array[h mod R]
36     while (true) {
37       val := node.val
38       <safe, idx> := <node.safe, node.idx> // one 64−bit read
39       if (idx > h) goto Line 52
40       if (val ≠⊥) {
41         if (idx = h) { // try dequeue transition
42           if (CAS2(node, <safe, h, val>, <safe, h+R, ⊥>))
43             return val
44         } else { // mark node unsafe to prevent future enqueue
45           if (CAS2(node, <safe, h, val>, <0, h, val>)) goto Line 52
46         }
47       } else { // idx≤ h and val =⊥: try empty transition
48         if (CAS2(node, <safe, idx, ⊥>, <safe, h+R, ⊥>))
49           goto Line 52
50       }
51     } // end of while loop, go back to Line 36
52     // failed to dequeue, check for empty
53     <closed, t> := crq.tail
54     if (t ≤ h+1) {
55       fixState(crq)
56       return EMPTY
57 } } }
```

(b) Dequeue

```
58 void fixState(crq : pointer to CRQ) {
59   // local variables
60   h, t : 64 bit int
61
62   while (true) {
63     h = F&A(&crq.head, 0)
64     t = F&A(&crq.tail, 0)
65
66     if (crq.tail ≠ t)
67       continue // continue loop at Line 62
68
69     if (h ≤ t)
70       return // nothing to do
71
72     if (CAS(&crq.tail, t, h))
73       return
74 } }
```

(c) fixState()

```
75 enqueue(crq : pointer to CRQ, arg : Object) {
76   // local variables
77   val, idx : 64 bit int
78   h, t : 64 bit int
79   node : pointer to Node
80   closed : boolean
81   safe : boolean
82
83   while (true) {
84     <closed, t> := F&A(&crq.tail, 1) // F&A on all 64 bits of tail
85     if (closed)
86       return CLOSED
87     node = &crq.array[t mod R]
88     val := node.val
89     <safe, idx> := <node.safe, node.idx> // one 64−bit read
90     if (val =⊥) {
91       if ((idx ≤ t) and
92           (safe = 1 or crq.head ≤ t) and
93           CAS2(node, <safe, idx, ⊥>, <1, t, arg>)) {
94         return OK
95       }
96     }
97     h := crq.head
98     if (t − h ≥ R or starving()) {
99       T&S(&crq.tail.closed) // atomically set tail.closed to true
100      return CLOSED
101    }
102 } }
```

(d) Enqueue

Figure 3: Pseudocode of CRQ algorithm.

A node can be in one of two states: if its value is $\perp$ the node is *empty*; otherwise the node is *occupied*. A CRQ operation attempts to *transition* a node from empty to occupied or vice versa using CAS2. We say that an operation, *op*, *accesses node u using index i* if *op*'s F&A returns i and $u = i \bmod R$, and refer to the operation as enq_i or deq_i as appropriate. An operation accessing a node uses the value of the node's safe bit and index, as described next, to determine whether it can attempt a transition or should obtain a new index and try accessing another node.

Dequeuing an item When a node is in an occupied state, (s,i,x), it holds the item x that has been stored by $enq_i(x)$. In this case, only deq_i, the dequeue operation accessing the node using index i – *exactly* i and not just equal to i modulo R – can return x. Such a dequeue attempts to remove x by performing the transition $(s,i,x) \mapsto (s,i+R,\perp)$ using CAS2 (Figure 3b, Line 42). We refer to this as a *dequeue transition*.

Dequeue arrives before enqueuer while node is empty This case occurs when an empty node whose state is $(s,i,\perp)$ is accessed by deq_j with $j = i+kR$, i.e., before the matching enqueue enq_j completes. Similarly to the infinite array queue, deq_j prevents enq_j from storing its item in the node by performing an *empty transition* $(s,i,\perp) \mapsto (s,j+R,\perp)$ (Line 48). In fact, the empty transition prevents *any* operation using some index $j-kR$ from performing a transition on the node. This stronger property was not needed in the infinite array queue, where only one enqueuer and one dequeuer ever access a node.

Dequeue arrives before enqueuer while node is occupied This case has no analog in the infinite array queue. It occurs when an occupied node (s,i,x) is accessed by deq_j with $j = i+kR$ ($k > 0$), i.e., before deq_i which is the dequeuer supposed to dequeue x. While deq_j cannot remove x, it must still mark the node somehow before moving on, so that enq_j knows not to enqueue an item in the node. deq_j uses the safe bit for this purpose, making an *unsafe* transition $(s,i,x) \mapsto (0,i,x)$ (Line 45). Once a node is unsafe, all dequeuer transitions keep the safe bit at 0. This prevents any enqueuer from storing its item in the node unless it first verifies that the corresponding dequeuer has not yet started, as explained next.

Enqueuing an item When a node is in an empty state $(s,i,\perp)$, any $enq_j(x)$ operation with $j = i+kR$ may attempt an *enqueue*

transition to store x in the node. If $s = 1$, enq_j simply performs the transition $(1,i,\perp) \mapsto (1,j,x)$ (Figure 3d, Line 93). However, if $s = 0$, enq_j needs to make sure that deq_j is not the dequeuer that set s to 0, since then deq_j will not dequeue from the node. enq_j determines this by checking if $head \leq j$ (Line 92), which means that deq_j has not yet started. If so, then enq_j makes the transition $(0,i,\perp) \mapsto (1,j,x)$ (Line 93) which undoes the previous unsafe transition. If deq_j then starts after enq_j's check that $head \leq j$ and performs a transition on the node before enq_j's transition, then deq_j performs an empty transition (Line 49) which changes the node's index and causes enq_j's CAS2 to fail.

We now turn to a walk-through of the algorithm's pseudocode. For simplicity, we describe optimizations in Section 4.1.1, omitting them from the pseudocode.

Enqueue (Figure 3d) An enqueue *enq* repeats the following. It obtains an index to a ring node with a F&A on *tail* (Line 84). If the CRQ is closed, *enq* returns CLOSED (Lines 85-86). Otherwise, *enq* attempts an enqueue transition (Lines 87-96). If this fails (because the node is occupied or the CAS2 fails), *enq* decides to give up and closes the queue in one of two cases: (1) *enq*'s index has passed *head* by R places, indicating a possibly full queue, or (2) *enq* is failing to make progress for a long time (checked by `starving()`) (Lines 97-101).

Dequeue (Figure 3b) A dequeue *deq* repeats the following. It obtains an index, h, to a ring node using F&A on *head* (Lines 34-35). It then enters a loop in which it attempts to read a consistent state of the node and perform a transition. If $h < i$, where i is the node's index, then *deq* has been overtaken between its F&A and reading the node, and so it exits the loop (Line 39). If the node is occupied, *deq* attempts a dequeue transition (Lines 40-47). If the node is empty, *deq* attempts an empty transition (Line 48) and exits the loop if successful. Throughout this process, if *deq*'s CAS2 fails (implying the node's state changes) then *deq* restarts the loop of reading the node and performing a transition. Whenever *deq* exits the loop without successfully dequeuing an item, it verifies that the queue is not empty before trying to dequeue with a new index (Line 53-54). If the queue is empty, *deq fixes* (see below) the queue's state so that $head \leq tail$ before returning EMPTY (Lines 55-56).

Fixing the queue state (Figure 3c) A dequeuer F&A may bring the queue into an invalid state in which $head > tail$. In such a case, the dequeuer can just perform an empty transition and return EMPTY. However, doing so prevents the enqueuer with the same index from using the node, forcing it to F&A *tail* again and increasing contention. To avoid this problem, a dequeuer always verifies that $head \leq tail$ before returning EMPTY (Lines 62-74).

4.1.1 Optimizations

Bounded waiting for matching enqueues When an enqueue and dequeue operation using the same index are active concurrently, the dequeue may arrive at the node before the matching enqueuer. Performing an empty transition in such a case just leads to both operations restarting and accessing the F&A again, needlessly increasing contention on *head* and *tail*.

To avoid this, before performing an empty transition (Line 48), a dequeue operation checks whether $tail \geq h+1$, where h is the dequeue's index. If so, then the matching enqueuer is active and the dequeuer spins for a short while, waiting for the enqueue transition to take place. Only after timing out on this spin loop does the dequeue perform an empty transition.

Hierarchy awareness Large servers are typically built hierarchically, with *clusters* of cores such that inter-core communication inside a cluster is cheap, but cross-cluster communication is expensive. For example, in a multiprocessor system a cluster consists of all the cores on a (multicore) processor. In these hierarchical machines, creating batches of operations that complete on the same cluster without interference from remote clusters reduces synchronization cost.

To achieve this, we add a `cluster` field to the CRQ, which identifies the current cluster from which most operations should complete. Before starting a CRQ operation, a thread checks if it is running on `cluster`. If not, the thread waits for a while, and then CASes `cluster` to be its cluster and enters the algorithm (even if the CAS fails). Similarly to prior NUMA-aware lock-based algorithms [5, 7], this divides the execution into segments such that in each segment most operations in the CRQ are from the same cluster. However, our optimization does not rely on locks nor does it introduce blocking, as every operation eventually enters the CRQ.

4.1.2 CRQ linearizability proof

The CRQ is not a standard FIFO queue because an enqueue can return CLOSED. To deal with this we give the CRQ the semantics of a *tantrum queue*: a queue in which an enqueue can nondeterministically refuse to enqueue its item, returning CLOSED instead and moving the queue to a CLOSED state. When a tantrum queue is in the CLOSED state, every enqueue operation returns CLOSED without enqueuing its item.

In the following, we prove that CRQ is a linearizable tantrum queue. Let $E = e_1, e_2, \ldots$ be a possibly infinite execution of CRQ. We assume every thread whose next local step is to complete does indeed complete in E. We denote an operation $op \in \{deq, enq\}$ that returns *ret* in E by $\langle op : ret \rangle$. We now describe a procedure P (Figure 4) to assign linearization points to the operations in E.

Essentially, the linearization order of $\langle enq(x) : \text{OK} \rangle$ operations is by the index the enqueuer uses when successfully enqueuing its item, and similarly $\langle deq() : x \rangle$ operations ($x \neq$ EMPTY) are linearized in the order of the index used to dequeue. The trick is to order enqueues and dequeues consistently, since for example a dequeuer's F&A returning index i can occur before the corresponding enqueuer's F&A.

To do this, we track the CRQ's state at each point in E using an auxiliary sequential queue. The auxiliary queue consists of an

```
103 Input: CRQ execution, E = e1, e2, ...
104
105 for j = 1 ...
106   if e_j is a T&S by ⟨enq(x) : CLOSED⟩ that sets tail's closed bit
107       (Figure 3d, Line 99) then
108     Linearize ⟨enq(x) : CLOSED⟩ at e_j
109   elseif e_j = ⟨t := F&A(tail, 1)⟩ by ⟨enq(x) : CLOSED⟩ which
110           returns a closed value (Figure 3d, Line 84) then
111     Linearize ⟨enq(x) : CLOSED⟩ at e_j
112   elseif e_j = ⟨t := F&A(tail, 1)⟩ is the last F&A in E by ⟨enq(x) : OK⟩
113           (Figure 3d, Line 84) then
114     Q[t] := x
115     tail(Q) := t + 1
116     Linearize ⟨enq(x) : OK⟩ at e_j
117   elseif e_j = ⟨t := tail⟩, is a read returning t ≤ head by ⟨deq : EMPTY⟩
118           (Figure 3b, Line 53) then
119     Linearize ⟨deq : EMPTY⟩ at e_j
120   endif
121   while head(Q) < tail(Q)
122     h := min{i : Q[i] ≠ ⊥}
123     if dequeue whose F&A (Line 34) returns h not active in e1, ..., e_j then
124       goto Line 130
125     endif
126     x := Q[h]
127     Q[h] := ⊥
128     head(Q) := h + 1
129     Linearize ⟨deq : x⟩ at e_j
130   endwhile
```

Figure 4: CRQ linearization procedure P. Operations linearized at the same event are ordered based on the order of P's steps.

infinite array, Q, coupled with indices $head(Q)$ and $tail(Q)$ representing Q's head and tail. (Note that Q is *not* cyclic.) Initially, $tail(Q) = head(Q) = 0$ and $Q[i] = \perp$ for all i.

We process the execution one event at a time, in order of execution, but using information about future events to decide when to linearize an operation. When we linearize an operation we also apply it to the auxiliary queue. We linearize an $\langle enq(x) : \text{OK} \rangle$ on its final F&A, the one returning index t such that the operation enqueues x in $node(t)$. At this point we also set $tail(Q)$ to $t+1$. We linearize the dequeue of item $x = Q[h]$ as soon as the dequeue becomes active and h is the lowest indexed non-$\perp$ cell in Q, and set $head(Q)$ to $h+1$ at this point. We linearize a $\langle deq : \text{EMPTY} \rangle$ on its read of $tail$ that returns a value $\geq head$ (we later show that $head(Q) = tail(Q)$ at this point). The full pseudocode of P in Figure 4 also includes the straightforward cases of linearizing $\langle enq(x) : \text{CLOSED} \rangle$ operations.

By construction, the linearization point of an operation is within its execution interval, and all completed enqueues and all dequeues that return EMPTY are linearized. We now show that completed dequeues which do not return EMPTY are also linearized. Here we denote by $enq_i(x)$ the $\langle enq(x) : \text{OK} \rangle$ operation whose last F&A on $tail$ in E returns i, causing P to set $Q[i] := x$ and linearize it. Similarly, we denote a dequeue operation whose last F&A on $head$ in E returns i by deq_i.

Lemma 1. *Suppose P linearizes $enq_i(x)$. If there exists a dequeue operation deq that performs a F&A on head in E which returns i, then: (1) $deq = deq_i$ (i.e., deq performs no further F&As in E), (2) deq_i returns x if it completes, and (3) P linearizes $\langle deq_i : x \rangle$.*

Proof. Let $(s, j, \perp) \mapsto (1, i, x)$ be $enq_i(x)$'s enqueue transition storing x into $u = node(i)$ (Figure 3d, Line 93). Notice that $j \leq i$. If deq takes sufficiently many steps after obtaining i from its F&A on $head$, it performs a transition on u using index i. To see this, notice that deq moves on from u without performing a transition only if it reads an index $> i$ from u (Figure 3b, Line 39). Because enq_i's transition succeeds, deq is the only operation that can move u's index beyond i, so this is impossible.

Now, consider deq's transition. It cannot be $(\cdot, k, \perp) \mapsto (\cdot, i+R, \perp)$ (Line 48) since that implies enq_i's transition fails. deq's transition also cannot be of the form $(\cdot, k, v) \mapsto (0, k, v)$ (Line 45) because then, enq_i's transition succeeding implies that some enqueue (possibly enq_i) subsequently obtains index $t \leq i$ and then observes $head \leq t$, which is impossible since $head > i$. Thus, deq's transition can only be a dequeue of x. Hence (1) and (2) hold.

We prove (3) using induction on k, the number of linearized enqueue operations. For $k = 0$ the claim is vacuously true. Suppose now that the k-th enqueue operation linearized is $enq_i(x)$. If deq_i exists in E, then it does not complete before $enq_i(x)$'s F&A which returns i, since otherwise deq_i does not return x, contradicting (2). Therefore, there exists a first event e in which $Q[i] = x$ and deq_i is active. Thus at some event e', at or after e, $Q[i] = x$ and deq_i has performed the F&A on $head$ which returns i. Let $idx = \{j : j < i, Q[j] \neq \perp \text{ at } e'\}$. For all $j \in idx$, deq_j starts by e' (because deq_i's F&A has returned i) and does not complete before e' (as that implies it is not linearized before completing, contradicting the induction hypothesis). Therefore, at e' P linearizes deq_j for all $j \in idx$ and subsequently linearizes deq_i. □

To complete the linearizability proof, we must show that our linearization order meets the tantrum queue specification. Because we enqueue to Q's tail, dequeue from Q's head, and following the first enqueue to return CLOSED all enqueues do so, this amounts to showing that the auxiliary queue is empty when we linearize a $\langle deq : \text{EMPTY} \rangle$ operation. Lemma 2 below implies this, because we linearize a $\langle deq : \text{EMPTY} \rangle$ when it reads a value t from $tail$ (Figure 3b, Line 53) such that $t \leq h+1$, where $h < head$ is the value that the deq's prior F&A returns (Line 34).

```
131 // shared variables on distinct cache lines :
132 tail : pointer to CRQ
133 head : pointer to CRQ
134 // initially :
135 tail = head = empty CRQ
```

(a) Globals

```
136 dequeue() {
137   // local variables
138   crq : pointer to CRQ
139   v : 64 bit value
140
141   while (true) {
142     crq := head
143     v := dequeue(crq)
144     if (v ≠ EMPTY) return v
145     if (crq.next = null) return EMPTY
146     CAS(&head, crq, crq.next)
147 } }
```

(b) Dequeue

```
148 enqueue(x : Object) {
149   // local variables
150   crq, newcrq : pointer to CRQ
151
152   while (true) {
153     crq := tail
154     if (crq.next ≠ null) {
155       CAS(&tail, crq, crq.next)
156       continue // next iteration at Line 153
157     }
158     if (enqueue(crq, x) ≠ CLOSED)
159       return OK
160     newcrq := a new CRQ initialized to contain x
161     if (CAS(&crq.next, null, newcrq)) {
162       CAS(&tail, crq, newcrq)
163       return OK
164     }
165   }
166 }
```

(c) Enqueue

Figure 5: Pseudocode of the LCRQ algorithm, using a linearizable CRQ black box.

Lemma 2. *If at event e, $head \geq tail$, then $head(Q) = tail(Q)$.*

Proof. Suppose towards a contradiction that $head(Q) < tail(Q)$ at e. Then there exists a minimal i such that $Q[i] \neq \perp$ at e. Because we update $tail(Q)$ following the order of F&As on $tail$, $i < tail(Q) \leq tail \leq head$ at e. Thus, deq_i is active before e and should have been linearized by P, a contradiction. □

In conclusion, we have shown the following.

Theorem 1. *CRQ is a linearizable implementation of a tantrum queue.*

4.2 The LCRQ algorithm

We now present LCRQ using the CRQ as a black box. The LCRQ is simply a linked list of CRQs in which dequeuing threads access the head CRQ and enqueuing threads access the tail CRQ (Figure 5a). An enqueue(x) operation that receives a CLOSED response from the tail CRQ creates a new CRQ, initialized to contain x, and links it after the current tail, thereby making it the new tail (Figure 5c). If the head CRQ becomes EMPTY and there is a node linked after it, dequeues move to the next node, after installing it as the new head (Figure 5b).

Memory reclamation A dequeue that successfully changes the head pointer cannot reclaim the memory used by the old CRQ because there may be concurrent operations about to access it (i.e., stalled just before Line 143 or Line 158). We address this problem by using *hazard pointers* [18] to protect an operation's reference to the CRQ it is about to access. We omit the details, which are standard.

Linearizability Assuming that the CRQ is a linearizable tantrum queue, proving that LCRQ is a linearizable queue implementation is straightforward:

Theorem 2. *If CRQ is a linearizable* tantrum queue *implementation, then LCRQ is a linearizable* queue *implementation.*

Proof. (Sketch) We linearize an enqueue that completes after appending a new CRQ to the list at the CAS which links the new CRQ (Figure 5c, Line 161). We linearize any other completed operation at the point in which its final CRQ operation takes place. The `next` pointer of a CRQ q changes from `null` only after q becomes CLOSED, and conversely, after a CRQ q becomes CLOSED no new enqueue completes until a new CRQ is linked after q. Thus, if q_0 precedes q_1 in the list, any q_1 enqueue is linearized after any q_0 enqueue. Similarly, any q_0 dequeue is linearized before any q_1 dequeue. Linearizability follows. □

4.2.1 LCRQ nonblocking proof

In this section, we sketch the proof of the following theorem:

Theorem 3. *LCRQ is op-wise nonblocking.*

An enqueuer that does not complete within a finite number of steps in the tail CRQ closes it. Once the CRQ is closed, every enqueuer taking enough steps tries to append a new CRQ to the LCRQ. The first one to CAS the CRQ's next pointer (Figure 5c, Line 161) succeeds and completes. Thus, an enqueue operation completes within a finite number of steps by enqueuing threads.

Now, consider a dequeuer *deq* taking an infinite number of steps without completing. Suppose first that *deq* remains in one LCRQ node, q. If enqueuers take infinitely many steps in q, then q does not close and so, because q's size is finite, dequeuers remove items from q. If enqueuers take only finitely many steps in q, then from some point only dequeuers take steps in q and so eventually q's *head* exceeds its *tail*. Then *deq* finds that q is empty (Lines 53-54), enters `fixState()` but never leaves. Thus, new dequeuers continue to enter q and increment *head*. Since the number of dequeuers is finite, this implies some dequeuer completes.

The other possibility is that *deq* returns EMPTY in each CRQ node q_i it enters but never reaches the LCRQ's tail. Each node q_i contains at least one item, and so there is a dequeuer d_i that holds the index to this item. After traversing through T nodes, where T is the number of threads in the system, it must be that $d_i = d_j$ for some $j > i$. This means d_i completes and returns. Overall, we have shown that a dequeue must complete within a finite number of steps by dequeuing threads.

5. Evaluation

Evaluated algorithms We compare LCRQ to the best performing queues reported in the recent literature, all of which are based on the *combining* principle: Hendler et al.'s **FC queue** [11] and Fatourou and Kallimanis' **CC-Queue** and **H-Queue** [7]. We also test Michael and Scott's classic nonblocking **MS queue** [19].

The FC queue is based on *flat combining*, in which a thread becomes a *combiner* by acquiring a global lock, and then applies the operations of the non-combining threads. The queue we test is a linked list of cyclic arrays, with a new tail array allocated when the old tail fills.

The CC-Queue replaces each of the two locks in Michael and Scott's *two-lock queue* [19], which serialize accesses to the queue's head and tail, with an instance of the *CC-Synch* universal construction [7]. The CC-Synch universal construction maintains a linked list to which threads add themselves using SWAP. The thread at the head of the list traverses the list and performs the requests of waiting threads. Since the enqueue and dequeue CC-Synch instances work in parallel, the CC-Queue outperforms the FC queue [7].

The H-Queue is a hierarchical version of the CC-Queue. It uses an instance of the *H-Synch* universal construction [7] to replace the two-lock queue's locks. The H-Synch construction consists of one instance of CC-Synch per cluster and a lock that synchronizes the CC-Synch instances. Each CC-Synch combiner acquires the lock and performs the operations of the threads on its cluster.

To obtain the most meaningful results, we use the queue implementations from Fatourou and Kallimanis' benchmark framework [7, 8], all of which are in C [5]. We incorporate the FC queue implementation into this framework.

LCRQ implementation We use CRQs whose ring size, R, is 2^{17}. (We include a sensitivity study of LCRQ to the ring size below.) In addition to baseline LCRQ, we also evaluate LCRQ+H, in which we enable our hierarchical optimization (with a timeout of 100 μs). To explore the impact of CAS failures, we test LCRQ-CAS, a version of LCRQ in which we implement the F&As using a CAS loop. All LCRQ variants include the overhead of pointing a hazard pointer at the CRQ before accessing it [6].

Methodology We follow the testing methodology of prior work [7, 19]. We measure the time it takes for every thread to execute 10^7 pairs of enqueue and dequeue operations, averaged over 10 runs.

As in prior work, in every test we avoid artificial *long run* scenarios [19], in which a thread zooms through many consecutive operations, by having each thread wait for a random number of nanoseconds (up to 100) between operations. Each thread is *pinned* to a specific hardware thread, to avoid interference from the operating system scheduler. Our tests use the `jemalloc` [6] memory allocator to prevent memory allocation from being a bottleneck. Results' variance is negligible (we use a dedicated test machine).

Platform We use a Fujitsu PRIMERGY RX600 S6 server with four Intel Xeon E7-4870 (Westmere EX) processors, which were launched by Intel in early 2011. Each processor has 10 2.40 GHz cores, each of which multiplexes 2 hardware threads, so in total our system supports 80 hardware threads. Each core has private write-back L1 and L2 caches; an inclusive L3 cache is shared by all cores.

Single processor executions (Figure 6a) Here we restrict threads to run on one of the server's processors. This evaluates the queues in a modern multicore environment in which all synchronization is handled on-chip and thus has low cost. We omit results of LCRQ+H and H-Queue, since they are relevant only for multi-processor executions.

LCRQ outperforms all other queues beyond 2 threads. From 10 threads onwards, LCRQ outperforms CC-Queue by 1.5×, the FC queue by $> 2.5\times$, and the MS queue by $> 3\times$. LCRQ-CAS matches LCRQ's performance up to 4 threads, but at that point its performance levels off. Subsequently, LCRQ-CAS exhibits the throughput "meltdown" associated with highly contended hot spots. Its throughput at maximum concurrency is 33% lower than its peak performance at 8 threads. Similarly, MS queue's performance peaks at 2 threads and degrades as concurrency increases.

Table 2 explains the above results. LCRQ, LCRQ-CAS and the MS queue all complete in a few instructions, but some of these

[5] We fixed a memory leak bug in the CC and H-Queue implementations, thereby improving their performance.

[6] This consists of a writing the CRQ's address to a thread-private location, issuing a memory fence, and rereading the LCRQ's *head*/*tail*.

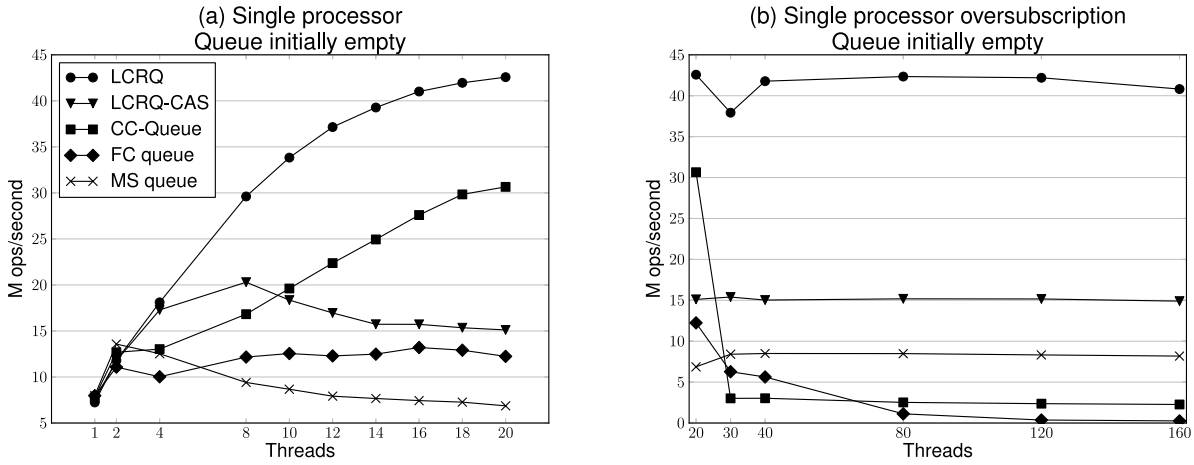

Figure 6: Enqueue/dequeue throughput on a single processor. The right plot shows throughput with more threads than available hardware threads; the first point, showing the throughput at maximal hardware concurrency, is included for reference.

Single processor execution (queue initially empty)										
	1 thread					20 threads				
	LCRQ	**LCRQ-CAS**	**CC-Queue**	**FC queue**	**MS queue**	**LCRQ**	**LCRQ-CAS**	**CC-Queue**	**FC queue**	**MS queue**
Latency	0.13 μs	0.96 ×	0.95 ×	0.91 ×	0.88 ×	0.44 μs	2.70 ×	1.45 ×	3.51 ×	5.95 ×
Instructions	278.46	284.96	294.96	284.96	228.77	280.27	302.08	867.27	3846.29	321.05
Atomic operations	2	2	1	1	1.5	2	3.04	1	0.21	4.3
L1 misses	0.51	0.51	0	0.03	0	3.85	7.64	9.62	5.46	13.19
L2 misses	0.05	0.06	0	0	0	3.86	7.23	6.91	3.81	13.15

Table 2: Single processor average per-operation statistics. Latency numbers are relative to LCRQ. There are no L3 misses because the workload fits into the shared L3 cache.

are expensive atomic operations on contended locations. LCRQ-CAS and the MS queue suffer from CAS failures, which also lead to more cache misses as the algorithm wastes work. In the combining algorithms, communication between combiners and waiting threads causes more cache misses compared to LCRQ.

Oversubscribed workloads (Figure 6b) Problems related to blocking usually occur in *oversubscribed* scenarios, in which the number of software threads exceeds the hardware supported level and forces the operating system to context switch between threads. If a thread holding a lock is scheduled out, the algorithm cannot make progress until it runs again. We show this by increasing the number of threads in a single processor execution beyond 20. The throughput of the lock-based combining algorithms plummets, with FC queue dropping by 40× and CC-Queue by 15×, whereas both LCRQ and the MS queue maintain their peak throughput. As a result, LCRQ outperforms the CC-Queue by 20×.

Four processor executions (Figure 7) To measure the effect of the increased synchronization cost between processors, we pin the threads across the processors in a round-robin manner, so that the cross-processor cache coherency cost always exists. One can see how, in contrast to the single processor experiment, when going from 1 to 2 threads the throughput of all algorithms drops due to cross-processor synchronization, except for LCRQ and LCRQ+H.

Figure 7a shows the results when the queue is initially filled with 2^{16} elements, thus keeping the queue's head and tail apart [7]. This causes the throughput of CC-Queue to degrade by $\approx$ 10% compared to the initially empty case (Figure 7b), due to reduced *locality*: in an initially empty queue, the queue's state keeps hovering around empty and so there is a 1 in 4 chance that dequeued items will have just been enqueued on the same processor by the enqueuing combiner. In contrast, switching to an initially filled queue *improves* LCRQ's throughput by $\approx$ 5%. The reason is that when the queue is not empty an LCRQ dequeuer does not wait for an enqueuer to arrive at its ring node. (Table 3 shows that LCRQ operations take less instructions to complete.) The reduced locality does not hurt LCRQ because dequeued items are fetched *in parallel* by all operations, and not sequentially by a single combiner. Overall, using an initially filled queue increases LCRQ's advantage over CC-Queue from $\approx$ 1.5× to $\approx$ 1.8×.

Heavy synchronization cost due to lack of locality also explains why only the hierarchical LCRQ+H and H-Queue scale past 16 threads. These algorithms amortize the synchronization cost by running batches of operations on a single processor while operations on other processors wait. However, H-Queue suffers much more from the reduced locality caused when switching to an initially filled queue: it triples the number of L3 misses (Table 3), which must be satisfied from off-chip resources and so its throughput drops by $\approx$ 40%. In contrast, LCRQ+H maintains its performance, increasing its advantage over H-Queue from 1.5× to 2.5×.

Latency of operations (Figure 8) Examining the latency distribution of queue operations at maximum concurrency provides more refined insight on the performance of the algorithms. For instance, while the average latency of an LCRQ+H operation is 2.19 μs (Table 3), 96% of the operations complete in $\leq$ 0.5 μs and 98% in $\leq$ 1 μs. The remaining operations are those that complete only

[7] On a single processor this test yields similar results to an initially empty queue and so we did not discuss it earlier.

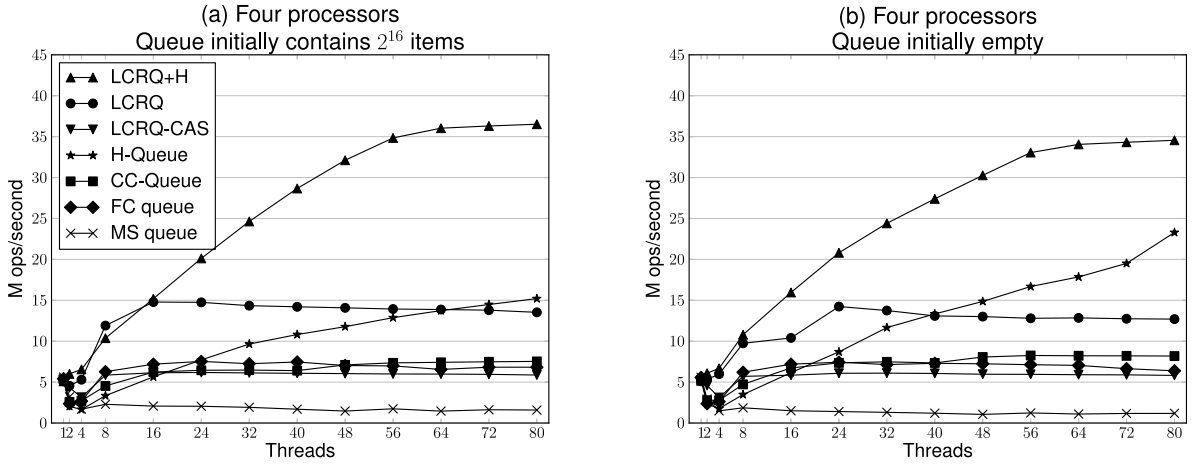

Figure 7: Enqueue/dequeue throughput on four processors (threads run on all processors from the start).

	Four processor execution (80 threads)									
	Queue initially empty					Queue initially full				
	LCRQ+H	**LCRQ**	**LCRQ-CAS**	**H-Queue**	**CC-Queue**	**LCRQ+H**	**LCRQ**	**LCRQ-CAS**	**H-Queue**	**CC-Queue**
Latency	2.19 μs	6.20 μs	13.50 μs	3.28 μs	9.70 μs	2.05 μs	5.81 μs	13.45 μs	5.19 μs	10.55 μs
Instructions	1456.65	307.15	338.98	5670.17	16249.94	1515.60	278.62	293.86	9173.94	18224.62
Atomic operations	2	2	2.88	1.05	1	2	2	2.95	1.05	1
L1 misses	4.12	2.91	4.15	9.99	10.70	3.43	3.01	4.31	10.60	11.33
L2 misses	4.15	2.83	4.01	7.10	8.65	3.54	2.90	4.17	7.74	9.07
L3 misses	0.51	1.47	2.23	0.34	5.90	0.81	1.43	2.22	0.95	6.19

Table 3: Four processor average per-operation statistics.

after the timeout expires. The spinning these operations do while waiting accounts for the increased average instruction count of LCRQ+H compared to LCRQ shown in Table 3. In general, LCRQ operations have better latency than combining-based operations, which spend time either servicing other threads or waiting for the combiner. On a single processor, 42% of LCRQ operations finish in ≤ 0.1 μs while none of the combining operations do. On four processors, 80% of LCRQ operations finish in ≤ 4 μs compared to 50% of CC-Queue operations. Similarly, 80% of LCRQ+H operation finish in ≤ 0.2 μs compared to 30% of H-Queue operations.

Ring size sensitivity study (Figure 9) The ring size plays an important role in the performance of LCRQ. Intuitively, as the ring size decreases an LCRQ operation needs more tries before it succeeds in performing an enqueue/dequeue transition.

To quantify this effect, we test LCRQ on an initially empty queue at maximum concurrency with various ring sizes. On a single processor, taking $R \geq 32$ is enough for LCRQ to outperform the CC-Queue by 1.33×. As R increases LCRQ's throughput increases up to $\approx 1.5\times$ that of the CC-Queue. In other words, as long as an individual CRQ has room for all running threads, LCRQ obtains excellent performance.

On the four processor benchmark the results are similar, but due to the higher concurrency level, LCRQ outperforms CC-Queue starting with $R = 128$ and the advantage becomes $\approx 1.5\times$ starting with $R = 1024$. LCRQ+H requires $R = 512$ to match H-Queue and $R = 4096$ to outperform H-Queue by 1.5×.

6. Conclusion

We have presented **LCRQ**, a concurrent nonblocking linearizable FIFO queue that outperforms prior combining-based queue implementations by 1.5× to more than 2× in all concurrency levels on an x86 server with four multicore processors. LCRQ uses contended F&A objects to spread threads around items in the queue, allowing them to complete in parallel. Because the hardware guarantees that every F&A succeeds, we avoid the costly failures that plague CAS-based algorithms.

Our results show a couple of ways in which modern x86 multicore architecture requires reevaluating conventional wisdom about concurrent programming. First, LCRQ shows that on modern hardware an algorithm with a contended hot spot can scale quite well. Instead, it is CAS retries that are often the cause for notorious "contention meltdowns." Second, the conventional wisdom in the literature, of avoiding F&A or CAS2 since they are not widely supported, is outdated. We believe these principles can guide the design of future concurrent algorithms.

More practically, the LCRQ algorithm is simple to implement and offers excellent and robust performance on one of today's dominant multicore architecture. We therefore hope it gets adopted and used in practice.

Acknowledgments

This work was supported by the Israel Science Foundation (grant 1386/11), by the Israeli Centers of Research Excellence (I-CORE) program (Center 4/11), and by Intel's lab support program.

References

[1] Power ISA Version 2.06. `http://www.power.org/resources/downloads/PowerISA_V2.06B_V2_PUBLIC.pdf`, January 2009.

[2] G. E. Blelloch, P. B. Gibbons, and S. H. Vardhan. Combinable memory-block transactions. In *SPAA 2008*.

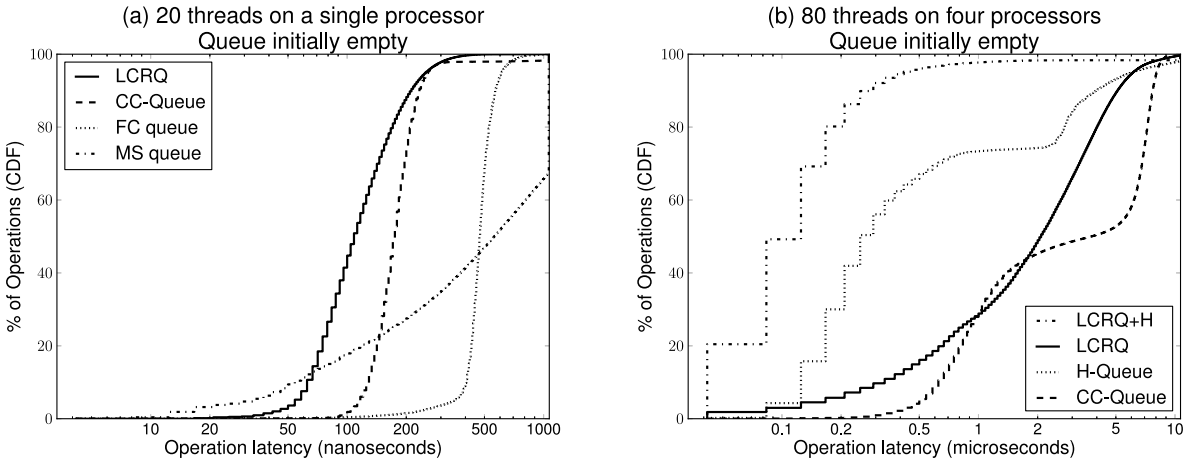

Figure 8: Cumulative distribution of queue operation latency at maximum concurrency.

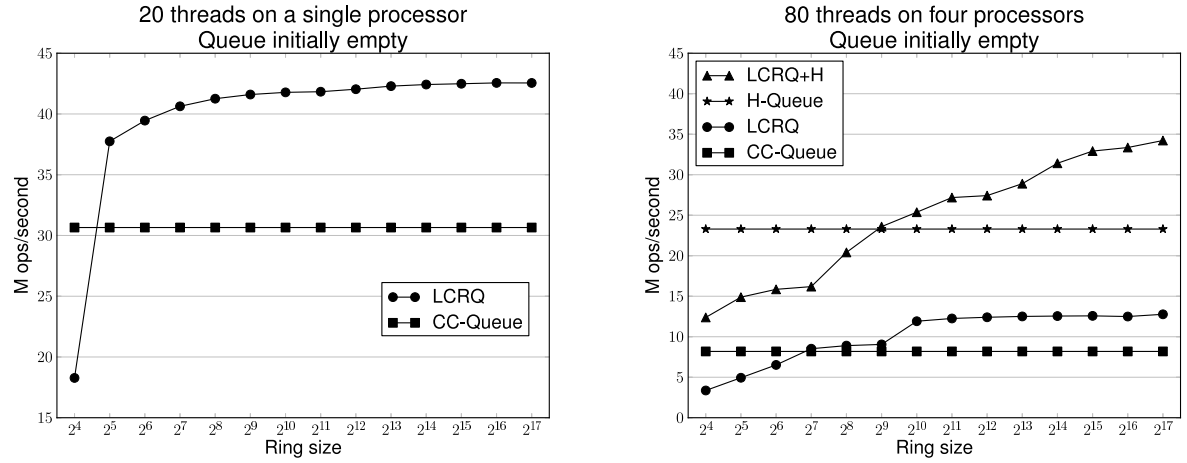

Figure 9: Impact of ring size on LCRQ throughput (CC-Queue and H-Queue results are shown for reference).

[3] G. E. Blelloch, P. Cheng, and P. B. Gibbons. Scalable room synchronizations. *Theory of Computing Systems*, 36, 2003.

[4] R. Colvin and L. Groves. Formal verification of an array-based nonblocking queue. In *ICECCS 2005*.

[5] D. Dice, V. J. Marathe, and N. Shavit. Lock cohorting: a general technique for designing numa locks. In *PPoPP 2012*.

[6] J. Evans. Scalable memory allocation using jemalloc. `http://www.facebook.com/notes/facebook-engineering/scalable-memory-allocation-using-jemalloc/480222803919`, 2011.

[7] P. Fatourou and N. D. Kallimanis. Revisiting the combining synchronization technique. In *PPoPP 2012*.

[8] P. Fatourou and N. D. Kallimanis. A highly-efficient wait-free universal construction. In *SPAA 2011*.

[9] E. Freudenthal and A. Gottlieb. Process coordination with fetch-and-increment. In *ASPLOS 1991*.

[10] A. Gottlieb, B. D. Lubachevsky, and L. Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *TOPLAS*, 5(2), Apr. 1983.

[11] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *SPAA 2010*.

[12] M. Herlihy. Wait-free synchronization. *TOPLAS*, 13:124–149, January 1991.

[13] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[14] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *TOPLAS*, 12:463–492, July 1990.

[15] M. Hoffman, O. Shalev, and N. Shavit. The baskets queue. In *OPODIS 2007*.

[16] A. Kogan and E. Petrank. Wait-free queues with multiple enqueuers and dequeuers. In *PPoPP 2011*.

[17] E. Ladan-Mozes and N. Shavit. An optimistic approach to lock-free FIFO queues. In *DISC 2004*.

[18] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE TPDS*, 15(6):491–504, June 2004.

[19] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC 1996*.

[20] M. Moir, D. Nussbaum, O. Shalev, and N. Shavit. Using elimination to implement scalable and lock-free FIFO queues. In *SPAA 2005*.

[21] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97, July 2010.

[22] N. Shafiei. Non-blocking array-based algorithms for stacks and queues. In *ICDCN 2009*.

[23] P. Tsigas and Y. Zhang. A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems. In *SPAA 2001*.