Chapter 7

Spin Locks and Contention Management

7.1 Introduction

We now turn our attention to the performance of mutual exclusion protocols on realistic architectures. Any mutual exclusion protocol poses the question: what do you do if you can't get the lock? There are two alternatives. If you keep trying, the lock is called a *spin lock*, and this activity is sometimes called *spinning* or *busy-waiting*. Spinning is sensible when you expect the lock delay to be short. For obvious reasons, spinning makes sense only on multiprocessors. You might also inform the operating system's scheduler that the time is right to schedule some other thread on your processor, an activity that is sometimes called *blocking*. Because switching from one thread to another is expensive, this alternative makes sense if you expect the lock delay to be long. Blocking always makes sense on a uniprocessor. In this chapter, we focus on spin locks.

We will perform mutual exclusion using the simple `Lock` interface illustrated in Figure 7.1. It provides two methods: `acquire` and `release`. Figure 7.2 shows how to use such a lock to implement a `fetch-and-inc` register from a read/write register.

Many architectures provide an atomic *test-and-set* instruction (method `TAS`) that atomically returns the register's current value and sets that value to 1. A schematic definition of this instruction appears in Figure 7.3. (The definition is schematic in the sense that the `TAS` instruction is implemented directly in hardware, not really as a Java class as shown.) A `TAS` register seems like the ideal way to implement a spin lock. A value of 0 indicates that the lock is free. The first thread to call the `TAS` method atomically sets the register value to 1 (indicating that the lock is taken) and returns value 0, indicating that the

[0] *This chapter is part of the Manuscript* MULTIPROCESSOR SYNCHRONIZATION *by Maurice Herlihy and Nir Shavit copyright © 2003, all rights reserved. This document is not for distribution and should only to be used by students in this class.*

```
public interface Lock {
  void acquire();
  void release();
}
```

Figure 7.1: The Lock interface

```
public class FAIRegister {
  private int value;
  private Lock lock;
  public FAIRegister() {
  }
  int read() {
    return value;
  }
  void write(int x) {
    value = x;
  }
  int fetchAndInc() {
    int oldValue;
    lock.acquire();
    oldValue = value++;
    lock.release();
    return oldValue;
  }
}
```

Figure 7.2: Fetch-and-Increment Register

```
public class TASRegister implements Register {
  private int value;
  public synchronized int TAS() {
    int oldValue = value;
    this.value = 1;
    return oldValue;
  }
  public int read() {
    return value;
  }
  public void write(int x) {
    value = x;
  }
}z,
```

Figure 7.3: Test-and-set register

```
public class TASLock implements Lock {
  private TASRegister value = new TASRegister(0);
  public void acquire() {
    while (value.TAS() == 1) {}
  }
  public void release() {
    value.write(0);
  }
}
```

Figure 7.4: Test-and-set lock

```
public class TTASLock implements Lock {
  private TASRegister value = new TASRegister(0);
  public void acquire() {
    while (true) {
      while (value.read() == 1) {}
      if (value.TAS() == 0)
        return;
    }
  }
  public void release() {
    value.write(0);
  }
}
```

Figure 7.5: Test-and-test-and-set lock

thread can enter the critical section. The thread releases the lock by resetting it back to 0. Figure 7.4 shows such an implementation.

Now consider the following alternative to the `TASLock` implementation, illustrated in Figure 7.5. Instead of performing the `TAS` directly, the thread repeatedly reads the locks waiting until it "looks" free. Only after the lock appears to be free does the thread apply the `TAS`.

Clearly, the `TASLock` and `TTASLock` implementations are equivalent from the point of view of correctness: each one guarantees no-lockout mutual exclusion. Under the simple model we have been using so far, there should be no difference between these two implementations.

How do they compare on a real multiprocessor? In a classic 1989 experiment, Anderson measured the time needed to execute a simple test program on several contemporary multiprocessors. These experiments measure the elapsed time for n threads to increment a shared counter one million times, normalized by the time it takes one thread to increment the shared counter one million times.

Figure 7.6 shows the elapsed times for each of these experiments. The top curve marked "spin test&set" is the `TASLock`, the middle curve marked "spin on read" is the `TTASLock`, and the bottom curve marked "ideal" shows the time that would be needed if the threads did not interfere at all. The difference is dramatic: the `TASLock` performance is so bad that no experiments were conducted for 13 or more processors, and the `TTASLock` performance, while substantially better, still falls far short of the idea. What's going on?

7.2 Multiprocessor Architectures

To understand the performance of the two spin-lock implementations, we need to take a closer look at multiprocessor architectures. Simplifying slightly, there are four kinds of basic architectures.

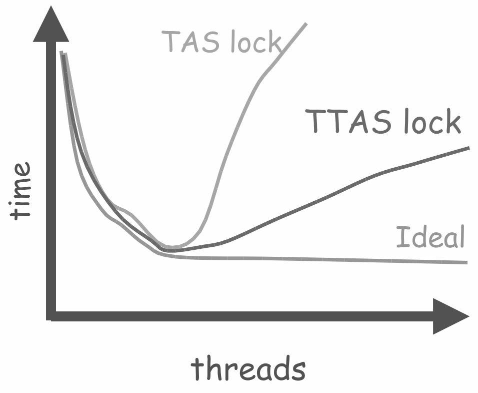

Figure 7.6: Performance of locking on a cache coherent bus based machine.

1. SISD – Single instruction stream, single data stream (uniprocessors).
2. SIMD – Single instruction stream, multiple data stream (vector machines).
3. MIMD – Multiple instruction stream, multiple data stream.

Here, we focus on MIMD architectures. There are two basic approaches to MIMD architecture. In a *shared bus* architecture, processors and memory are connected by a shared bus. A bus is a broadcast medium (like a tiny Ethernet). Both the processors and the memory controller can broadcast on the bus. Only one processor (or the memory) can write to the bus at a time, but all processors

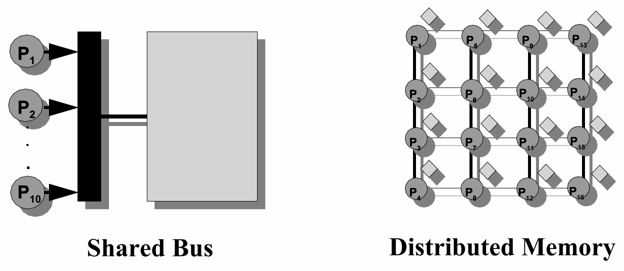

Figure 7.7: Shared Memory Architectures.

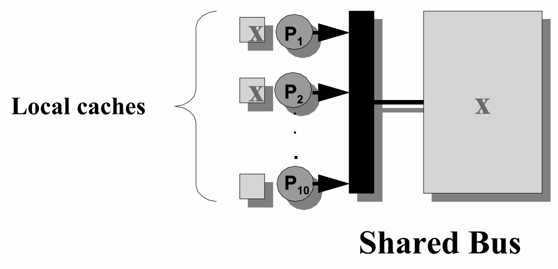

Figure 7.8: Shared bus architecture with local caches.

common today because they are easy to build. They scale poorly beyond, say, 32 processors, because the bus itself becomes a communication bottleneck.

An alternative to the shared-bus architecture is the *distributed memory* architecture illustrated on the right side of Figure 7.7. Each processor controls its local memory, and processors communicate by sending messages over a network. A number of distributed memory research machines have been built, including Alewife, CM5, TERA, and DASH. In principal distributed-memory machines are more scalable than shared-bus machines, but they are more difficult to build, and are rarer in practice. It is also possible to mix both kinds of architectures, say by having bus-based clusters linked by a fast network or switch, an approach at the core of the architecture of many modern commercial machines. Because memory in a processor's own cluster is faster to access then a thread in a cluster across the network, the time to access memory is not uniform. Such machines are often called non-uniform memory access machines, or NUMA for short.

7.3 Cache Memory & Consistency

In most shared-bus architectures, each processor has a *cache*, a small amount of high-speed memory in which the processor keeps data likely to be of interest (see Figure 7.8). A cache access typically requires one or two machine cycles, while a memory access typically requires tens of machine cycles. Technology trends are making this contrast more extreme: although both processor cycle times and memory access times are becoming faster, the cycle times are improving faster than the memory access times, so cache performance is critical to the overall performance of a multiprocessor architecture.

A cache stores the data itself, the address of the data, and various bookkeeping flags. When a processor reads from an address in memory, it first checks whether that address and its contents are present in its cache. If so, then the processor has a *cache hit*, and can load the value immediately. If not, then the processor has a *cache miss*, and must find the data either in the memory or in another processor's cache. The processor than broadcasts the address on

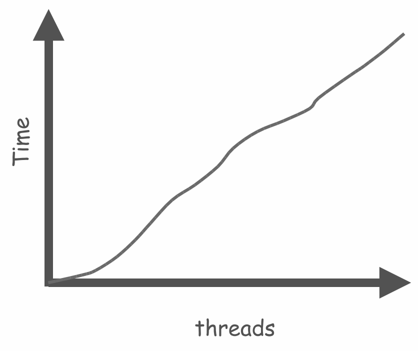

Figure 7.9: Time to quiesce on shared bus architecture with local caches.

the bus. The other processors "snoop" on the cache. If one processor has that address in it its cache, then it responds by broadcasting the address and value. If no processor has that address, then the memory itself will respond with the value at that address.

Now suppose the processor modifies the cached value. Here we have a problem: the value in the processor's is now different from the value in the memory, and from other copies of that address that might be stored in other processors' caches. The problem of keeping track of multiple copies of the same data is called the *cache coherence* problem, and ways to accomplish it are called *cache coherence* protocols.

Cache coherence protocols have two ways of dealing with modified data. A *write-through* coherence protocol immediately broadcasts the new value, so that both the memory and other processors' caches become aware of the new value. By contrast, Write-through protocols have the advantage that all cached copies of the data agree, but they have the disadvantage that every modification to a location requires bus traffic. Since the vast majority of all updates are to memory locations that are not shared among processors, write-through protocols are generally not used.

a *write-back* coherence protocol sends out an *invalidation* message message when the value is first modified, instructing the other processors to discard that value from their caches. Once the processor has invalidated the other cached values, it can make subsequent modifications without further bus traffic. A value that has been modified in the cache but not written back is called *dirty.* If the processor needs to use the cache for another value, however, it must remember to write back any dirty values.

Real cache coherence protocols can be very complex. For example, some protocols mix write-through and write-back strategies, some distinguish between exclusive and shared access, and almost all modern multiprocessors have multi-level caches, where each processor has an on-chip (L1) cache, and clusters of processors share an off-chip (L2) cache.

Cache architecture is a fascinating subject in its own right, but we already know most of what we need to understand the relative performance of the two

`TAS`-based lock implementations.

7.4 `TAS`-Based Spin Locks

In the simple `TASLock` implementation, each `TAS` operation goes over the bus. Since all of the waiting threads are continually using the bus, all threads, even those not waiting for the lock, end up having to wait to use the bus for their memory accesses. Even worse, the `TAS` operation invalidates all cached copies of the lock, so every spinning thread encounters a cache miss every time, and has to use the bus. When the thread holding the lock tries to release it, it may be delayed waiting to use the bus that is currently monopolized by the spinners. No wonder the `TASLock` performs so poorly.

Now consider the behavior of the `TTASLock` implementation while the lock is held by some thread. The first time a thread reads the lock it takes a cache miss and loads the value (which is 1) into its cache. As long as the lock is held, the thread repeatedly rereads the value, and each time it hits in its cache. While it is waiting, it produces no bus traffic, allowing other threads to get to memory unhindered by bus traffic, without hindering other processors, and allowing lock holder to release it without having to contest control of the bus from spinners.

Things get worse, however, when the lock is released. The lock holder releases the lock by writing 0 to the lock variable, which immediately invalidates the spinners' cached copies. They each take a cache miss, load the new value, and all (more-or-less simultaneously) call `TAS` to grab the lock. The first to succeed invalidates the others, who then sequentially reread the value, causing a storm of bus traffic until the processors settle down once again to local spinning.

This notion of *local spinning*, where threads repeatedly reread cached values instead of repeatedly using the bus, is an important principle critical to the design of efficient spin locks.

7.5 Introducing Delays

Note that spinning processors slow down other processors by consuming bus bandwidth. One way to reduce bus contention is by introducing delays into the `acquire` method. There are four ways of inserting delays, which can be defined by two dimensions:

1. Where the delay is inserted:
   (a) after the lock is released, or
   (b) after every separate access to the lock,
2. The way the delay duration is set:
   (a) statically, or
   (b) dynamically.

```
  public void acquire() {
    while (value.read() == 1 || value.TAS() == 1) {
      while(value.TAS() == 1) {}
      Thread.sleep(delay);
    }
  }
```

7.6 Delay After Release

Figure 7.6 shows an `acquire` implementation in which the thread inserts a delay between the time it notices that the lock is free and the time it calls `TAS`. Notice that the `TAS` is performed only if the lock is still free after the delay. This technique further reduces the number of unsuccessful `TAS` calls. When the lock is released all local copies are invalidated. Thus, the processor that updates its local copy and finishes waiting its delay time first is the one that acquires the lock. By the time the other processors finish their delays, their local copies of the lock should show that the lock has been reacquired.

7.6.1 Static Delays

We view the execution of the algorithm as a series of time units called *slots*. If, during each *slot*, only one processor uses the bus, there is no contention. We can achieve a similar effect by making each processor wait a different amount of (predetermined) time, so that if two processors started to wait simultaneously, one always finishes before the other. The amount of delay for each processor must be a unique multiple of the *slot* size, varying from 0 to $n - 1$ slots. This method of assigning delay times is *static*. Static delays ensure that at any instant at most one processor times out.

This algorithm performs well when there are many processors. It is likely that some spinning processor has a short delay. When the lock is released, some processor quickly reacquires it. With only one processor, however, the delay is unlikely to be short. Thus the lock remains unacquired and performance suffers.

With fewer processors, smaller delay times improve performance by reducing the time to pass control of the lock. With many processors, smaller delay times degrade performance, since multiple processors time out simultaneously. This observation motivates that the number of slots should be proportional to the number of participating processors. See Figures 7.10, 7.12, and 7.17.

7.6.2 Dynamic Delays

With *dynamic* delays, each processor initially assumes that no other processors are waiting and assigns itself a short delay time. Whenever it times out, sees that the lock is free, calls `TAS` and fails, then it detects contention, and doubles its delay time, up to a limit. (Empirical results show that the best upper limit is the static delay of the previous section.) Similarly, when a processor notices

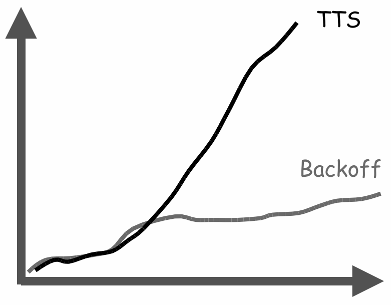

Figure 7.10: Performance of exponential backoff.

```
public void acquire() {
  while (value.read() == 1 || value.TAS() == 1) {
    Thread.sleep(delay);
  }
}
```

Figure 7.11: Delay after every lock reference

that it has a large number of successful `TAS` operations it assumes that there are fewer processors and halves its delay time. This method is called *exponential back-off*.

When processor activity is bursty, *back-off* performs poorly.

7.7 Delay After Every Lock Reference

In the algorithm in Figure 7.11, a thread inserts a delay after each reference to the lock. If a thread notices that another thread holds the lock, it spins on its locally cached copy and waits before trying to read again. Here to, the thread calls `TAS` only if if the lock is free after the delay. This technique also reduces the number of unsuccessful `TAS` operations and decreases bus activity.

The *static* and *dynamic* options for choosing delay times are also available here.

7.7.1 Exponential Backoff

7.8 Queue Locks

7.8.1 An Array Based Queue Lock

In Figure 7.14 we present the code of the A-lock. The idea is simple, keep an array and a `tail` counter which indexes into the array. The `tail` counter is

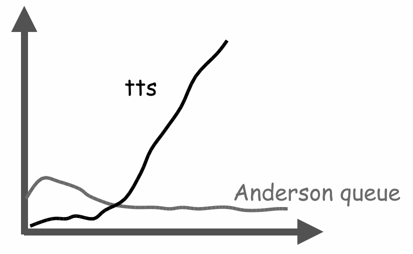

Figure 7.12: Performance of Anderson queue lock on cache coherent bus based machine.

accessed by a fetch-and-increment operation by every thread wishing to acquire the lock. Initially all entries in the queue except location 0 are set to `WAIT`. The tail counter is set to 0 and location 0 is set to `ENTER`. Each thread performs a fetch-and-increment on the tail, receiving the index of the slot it must wait on and incrementing the counter to point to the slot in which the next thread to arrive will spin on. It then spins on that slot in the array until a thread leaving the critical section sets it to `ENTER`. The lock improves on exponential backoff since it reduces invalidations to a minimum, provides FIFO ordering, and schedules access to the critical section tightly, minimizing the time from when it is freed by one thread to when it is reacquired by another. The fact that fetch-and-increment is not available on modern hardware is not a limitation since an efficient lock-free implementation of fetch-and-increment from compare-and-swap is easy to implement. The A-lock's main limitation is that it is not clear how to cache the array slots in memory. In any case we need to allocated the full array per lock even if only a subset of threads will ever access it.

7.8.2 The CLH queue lock

The CLH lock improves on the A-lock by allocating the slots on which thraeds spin dynamically. Figure 7.15 shows a simplified version of the *CLH Queue Lock*.

The lock is a virtual linked list of `Qnode` objects, each waiting to enter the critical section. We use the term "virtual" since unlike conventional linked lists, it cannot be passively traversed because the pointers in the list are the threads' private `pred` variables. We will not deal here with recycling of the `Qnode` objects though this can be done rather efficiently through simple modifications of the

```
public class backoff implements lock {

  private TASRegister value = new TASRegister(0);

 public void acquire() {
  int delay = MIN_DELAY;
  while (lock.read() == 1) {
   if (lock.TAS() == 0)
    return;
   sleep(random() % delay);
   if (delay < MAX_DELAY)
    delay = 2 * delay;
 }}}

  public void release() {
    value.write(0);
  }
}
```

Figure 7.13: Exponential backoff lock

above code.

To acquire the lock, a thread creates a `Qnode` structure, sets its `locked` variable to indicate that it has not yet released the critical section, and atomically places its own `Qnode` at the tail of the list while finding the `Qnode` object of its predecessor. It then spins on the `locked` variable of its predecessor (if there is one), waiting until the predecessor sets the `locked` field of its `Qnode` to *false*. To release the lock, a thread sets the `locked` field of its `Qnode` to *false*. The key point here is that each thread spins on a distinct location, so when one thread releases its lock, it does not invalidate every waiting thread's cache, only the cache of its immediate successor.

7.8.3 The MCS queue lock

The CLH algorithm relies on the fact that a thread spins on a locally cached copy of its prececessor's `Qnode`, since the `Qnode` itself was created by another thread, and on a distributed memory machine will be local to that thread. What do we do however if the machine is distributed, that is, is a NUMA machine, and is non-coherent, so the cost of spinning on a remote location is high? The answer is the MCS queue lock algorithm.

The *MCS Queue Lock* (shown in Figure 7.16) implements a queue of processors in a linked list waiting to enter the critical section. Like CLH the lock is linked-list of `Qnode` objects, where each `Qnode` represents either a lock holder or a thread waiting to acquire the lock. This is not a virtual list however. To

```
public class alock implements lock {

  private RMWRegister tail = new RMWRegister(0);

 public void acquire() {
 myslot[i] = tail.fetchInc();
 while (flags[myslot[i]] % n) == WAIT){};
 flags[myslot[i] % n] = WAIT;
}

public void release() {
 flags[myslot[i] + 1 % n] = ENTER;
} }
```

Figure 7.14: The A-lock

acquire the lock, a thread creates a `Qnode` structure, and places its own `Qnode` atomically at the head of the list. If it has a predecessor it directs a pointer from its predecessor to its `Qnode` so that its predecessor can locate the `Qnode`. It then spins on a *local* `locked` field in its own `Qnode`waiting until its predecessor sets this field to *false*. To release the lock, a thread locates its predecessor and sets the `locked` field of its predecessor's `Qnode` to *false*. The key point here is that each thread spins on a distinct local object, so the cost of spinning is low.

The relative performance of the queue lock on a cacheless NUMA machine are shown in Figure 7.17. This is an architecture from the 1980s. As can be seen, MCS outperforms all other techniques. However, the gap between exponential backoff and the MCS lock is small.

The relative performance of the queue lock on a state-of-art cache coherent NUMA machine are shown in Figure 7.17. This is an architecture from the 1998. As can be seen, exponential backoff on a modern NUMA machine is not a reasonable option since it does not scale, and MCS starts to outperform it from 20 processors and on.

7.9 Locking in Java

Reading material on Metalocks can be found at

`http:\\www.cs.tau.ac.il\~shanir\html\multiprocessor-synch2003\bibliography and related papers\metalock.ps.`

```
public class CLHLock {

  class Qnode {
    boolean locked = true;
  }

  /** Queue points to an unowned Qnode with locked == false */
  private RMWRegister queue = new RMWRegister(new Qnode());

  /** Creates a new instance of TASLock */
  public CLHLock() {
  }

  public void acquire(Qnode mynode) {
    /** mynode.locked == true */
    Qnode pred = (Qnode)queue.swap(mynode); /** find predecessor */
    while (pred.locked) {}
  }

  public void release(Qnode mynode) {
    mynode.locked = false;
    /** can use pred as Qnode for next acquire */
  }
}
```

Figure 7.15: CLH queue lock

```
public class MCSLock {

  class Qnode {
    boolean locked = false;
    Qnode   next   = null;
  }

  private RMWRegister queue = new RMWRegister(new Qnode());

  /** Creates a new instance of TASLock */
  public MCSLock() {
  }

  public void acquire(Qnode mynode) {
    Qnode pred = (Qnode)queue.swap(mynode);
    if (pred != null) {
      mynode.locked = true;
      pred.next = mynode;
      while (mynode.locked) {}
    }
  }
  public void release(Qnode mynode) {
    if (mynode.next == null) {
      if (queue.CAS(mynode, null))
        return;
      while (mynode.next == null) {}
    }
    mynode.next.locked = false;
  }

}
```

Figure 7.16: MCS queue lock

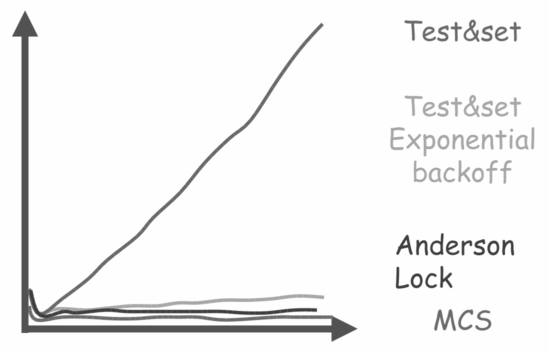

Figure 7.17: Performance of Spin-Locks on a cache-less NUMA machine.

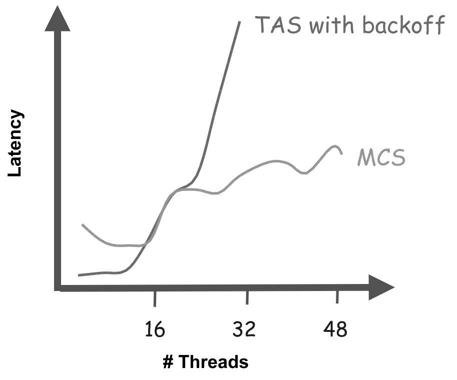

Figure 7.18: Performance of Spin-Locks on a modern (circa 1998) cache coherent NUMA machine.

7.10 Chapter Notes

The `TTASLock` is due to Kruskal and Rudolph and Snir. Exponential backoff is a well known technique used in Ethernet routing. Anderson was one of the first to empirically study contention in shared memory multiprocessors, introducing the A-lock, the first queue lock which was array based. Many of the graphs in this document are idealizations of his actual empirical results on the Sequent Symmetry. Mellor-Crummey and Scott provided the popular list based MCS Queue lock, later improved upon by the CLH lock of Craig and Landin and Hagersten. The NUMA machine graphs we show are idealizations of the experiments of Mellor-Crummey and Scott on the BBN butterfly machine. The modern machine graphs are due to Scott.

7.11 Bibliography

- T. E. Anderson. *The Performance Implications of Spin Lock Alternatives for Shared-Memory Multiprocessors.* IEEE Tran. on parallel and Distributed Systems, Vol. 1, No. 1, Jan. 1990.
- J. M. Mellor-Crummey and M. L. Scott. *Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors.* ACM Transactions on Computer Systems 9 (1): 21-65 (1991).
- Kruskal, L. Rudolph and M. Snir, Efficient synchronization of multiprocessors with shared memory, ACM Transactions on Programming Languages and Systems (TOPLAS), 10 (4), 1988, pp. 579–601.
- T. S. Craig. Building FIFO and priority-queueing spin locks from atomic swap. Technical Report TR 93-02-02, Dept. of Computer Science, Univ. of Washington, Feb. 1993.
- P. Magnussen, A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. In 8th IPPS, pages 165-171, Cancun, Mexico, Apr. 1994. Expanded version available as Efficient Software Synchronization on Large Cache Coherent Multiprocessors, SICS Research Report T94:07, Swedish Institute of Computer Science, Feb. 1994.