Queue Locks on Cache Coherent Multiprocessors

Peter Magnusson Anders Landin[*] Erik Hagersten[†]
Swedish Institute of Computer Science Sun Microsystems

Large-scale shared-memory multiprocessors typically have long latencies for remote data accesses. A key issue for execution performance of many common applications is the synchronization cost. The communication scalability of synchronization has been improved by the introduction of queue-based spin-locks instead of Test&(Test&Set). For architectures with long access latencies for global data, attention should also be paid to the number of global accesses that are involved in synchronization.

We present a method to characterize the performance of proposed queue lock algorithms, and apply it to previously published algorithms. We also present two new queue locks, the LH lock and the M lock. We compare the locks in terms of performance, memory requirements, code size, and required hardware support. The LH lock is the simplest of all the locks, yet requires only an atomic swap operation. The M lock is superior in terms of global accesses needed to perform synchronization and still competitive in all other criteria. We conclude that the M lock is the best overall queue lock for the class of architectures studied.

1 Introduction

A major issue in designing and programing multiprocessors is the cost of synchronization. Traditionally, this area of research has focused on hardware primitives and their implementation. In recent years, however, several researchers have demonstrated that what were perceived as hardware problems could be solved in software using simple synchronization primitives as building blocks.

The difficulties involved in designing synchronization primitives vary greatly with the nature of the underlying hardware. The earliest work in the area assumed that the most powerful atomic primitive was a read or a write [7, 9]. For various reasons, we are not interested in such a restriction: to begin with, such an algorithm can never be bounded [1]. The inefficiency of these algorithms prompted development of more powerful hardware primitives.

[*]psm@sics.se, landin@sics.se. Surface mail: SICS, Box 1263, 164 28 KISTA, Sweden.

[†]Erik.Hagersten@eng.sun.com. Work done while at SICS.

These primitives, such as atomic swap, were sufficiently powerful to implement trivial locking algorithms. The common problem with these locks, such as the Test&(Test&Set) lock [15], is that they required $O(N^2)$ network transactions to service N processors attempting to enter a critical section simultaneously. In some cases these locks were a major contributor to network contention, including the problem of so-called "hot-spots" [14].

To remedy this, Goodman suggested the queue-on-sync bits [5], which involves local spinning on a synchronization flag, thereby eliminating the $O(N^2)$ network operations. This solution was meant to be implemented in hardware, but Mellor-Crummey and Scott [11] and Andersson [2] implemented similar concepts in software.

The queueing principle addresses the scalability of a lock in terms of number of contending processors. Another scalability issue is performance—large shared memory architectures are today designed with some form of memory hierarchy. This implies a large cost difference between accessing local first-level caches and memory on remote nodes. This problem will be exasperated by the continuous performance improvements of microprocessors.

This paper has two main contributions. First, we analyze queueing locks from the perspective of performance on a large cache-coherent shared-memory multiprocessor. Second, we present a method of analysis that is more analytical than previous approaches, allowing us to draw more general conclusions about the performance of a synchronization algorithm. Also, we demonstrate the usefulness of this approach in algorithm design by using it to guide the formulation of two new queueing spin lock algorithms.

The paper is organized as follows. In section 2 we present our approach to analyzing queue lock performance, and, in section 3, we apply this method to previously published queue locks, the LH lock, and the M lock. Finally, we summarize the characteristics of all the locks in section 4 and conclude in section 5.

A longer version of this paper contains details of the study that we could not present here [10]. Wherever material is described as omitted, the reader is referred to the full paper.

2 Lock characterization

Comparing locks has previously been done either by comparing the complexity, such as determining whether acquiring a lock is $O(P)$ or $O(logP)$, by running some form of test on real hardware, or by simulation. These approaches have several limitations. A general $O()$ statement is useful mostly for theoretical scalability. In reality, this is seldom an issue—the constants involved are more important. Comparing locks on real hardware is made difficult by the absence of some primitives on suitable platforms. In some published studies, the implementation of the locks being compared bears little resemblance to the published pseudo-code.

Running a simulation is a useful approach, but as a method, simulation should be used with care. The way the simulation is designed and implemented becomes an additional set of assumptions in the analysis.

Our primary concern is the number of global accesses of the algorithms, which is the dominant performance impediment on a large multiprocessor. At the same time we want to keep memory requirements at a minimum, as well as the overhead of the locks when there is no contention (for instance when they are superfluous).

By hand-assembling the algorithms into a generic RISC-like assembler, and counting the number and types of operations each execution path requires, we can describe the cost of each critical path for every lock. Comparing the critical paths then allows us to make general statements about the strengths and weaknesses of the different algorithms.

2.1 Architectural assumptions

We assume a cache-coherent shared memory architecture, i.e., memory addresses can be read from and written to by any processor with the hardware maintaining consistency. The cache coherency protocol is presumed be of the write invalidate type, i.e., all other copies are invalidated upon writes and data is replicated in the caches upon multiple reads. Finally, we assume our memory to be sequentially consistent [8]. We expect most of the results to hold for weaker memory models as well, but we do not explore the issue in this paper.

Since we're interested in large machines, we assume that the principal distinction in memory access times is that between cache hits and misses. Therefore, it is the number and type of global memory accesses that will decide an algorithm's performance in relation to the size of the machine.

The interesting case for lock performance is when the locks are used frequently. This does not necessarily imply contention. Therefore, we assume that any (small) state that was copied into the cache upon the previous lock access is presumed to still reside in the cache unless another processor has written to that specific memory address (cache line). This is a *local* access. Any other access is global.

2.2 Generic processor

Each algorithm has been hand translated into "pseudo assembler", assuming a RISC-like processor. The time taken for each segment of code is divided up into generic operations: read, write, swap, execute simple instruction, and branches. A given architecture is characterized by atomic time units for individual operations. The time to execute a read is r, a write w, a swap s, a fetch-and-increment a, a clear-if-equal c, and a compare-and-swap x. Whether the operation is in local or global (remote) memory is indicated by suffixes. For instance, s_l is local swap and x_g a global compare-and-swap.

The compare-and-swap operation compares the old value with a parameter, and overwrites it if they are equal, returning true. Otherwise, the old value is left unchanged and the operation returns false. Clear-if-equal is similar, but always writes a zero.

The branch prediction is presumed to be statically given, i.e., all branch instructions contain a bit indicating whether the compiler or library code predicts the branch will be taken. Correctly predicted branches take time b_c, and incorrectly predicted take $\overline{b}_c$.

If any instructions are executed that is not a read, write, swap, or branch then this is counted as i.

The pseudo-assembler code for the locks, with timing, is omitted in this paper.

2.3 Timing analysis

We divide the timing elements of a lock into the different execution paths. Consider figure 1. Processor 2 acquires the lock for the first time (1), spending time t_l in the lock algorithm (*lock*). It then enters its critical section. Next, processor 1 attempts to take the lock and enqueues (2). The time for this operation is not critical, since the processor has to wait anyway. Next,

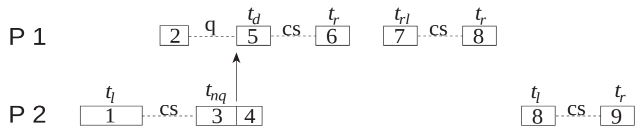

Figure 1: Timing components of a queue lock

processor 2 completes its critical section and signals to the next processor in the queue (if any) that the lock is free (3). This takes time t_{nq} (*notice-queue*). Processor 1 now proceeds to come off the queue (5), taking time t_d to do so (*detect*). This once more is on the critical path. Processor 2, meanwhile, does any clean-up required (4) prior to proceeding with non-critical code. The time spent here is not part of the sequential component, and is in any case reflected in the following two cases.

Processor 1 is now in its critical section. If more processors were to enter the queue, steps (3) and (5) would be repeated every time the lock was passed down the queue. We call this cost the *hand-off* cost of the queuing algorithm, $t_{ho} = t_{nq} + t_d$. When there is contention for the lock, the hand-off cost will be the part of the sequential execution that will be spent inside the lock algorithm.

In the example, processor 1 now releases the lock (6), taking time t_r (*release*). Later on, processor 1 retakes the lock (7), with no other processor having taken the lock in-between, spending time t_{rl} (*re-lock*). After its critical section, it releases the lock (8), t_r. This is the *optimistic* case, taking $t_{opt} = t_{rl} + t_r$. Since no component can be done in parallel, all the time spent in the lock is critical. The optimistic case is in a sense the cost of unnecessary locking.

Finally, processor 2 also takes (8) and releases the lock (9), but must this time spend time t_l to take the lock. This is the *pessimistic* case, $t_{pes} = t_l + t_r$. It reflects the cost of a lock that is only sporadically used.

There are two cases that we do not distinguish: taking a lock that, when we last had it, a queue formed, t_{lq}. In the figure, this would be (8). Also, the time to release the lock when there is a queue, case (4). These distinctions are not important in pratice. In [10] the distinction is made.

2.4 Memory requirements

We classify memory requirements into two types: pointers and flags. Space requirements are proportional to the total number of locks (L), the total number of processors (P), the number of processors currently in a queue (P_q), the maximum number of processors that conceivably could try for a lock simultaneously (P_{max}), the maximum number of locks a single processor might hold at the same time (n), and the current average number of locks held by a processor (n_{avg}). Pointers are not necessarily full address space pointers—typically an offset in a data structure will be sufficient. A flag has only two states, so requires only a single bit.[1] Typically, whether or not to allocate a full cache line for a flag or a pointer (in order to reduce false sharing) will depend on their number and use.

3 Lock analysis

In this section we describe and compare some previously published queue locks. Also, we present two new locks, the LH lock and the M lock.

For lack of space, pseudo code for previously published algorithms are not given in this paper.

3.1 Anderson's queue lock

Anderson describes an array-based lock for shared-memory multiprocessors [2]. A similar approach was independently developed by Graunke and Thakkar (see section 3.2). An array of flags is used to allocate a unique flag for each processor that might attempt to take the lock. This flag is cached and the processor can thus read-spin locally, relieving the network. The previous processor in the queue selectively releases the lock for the next processor. A side effect of this approach is guaranteed fairness, i.e., there is no possibility of starvation. Anderson's lock requires an atomic fetch-and-increment operation.

The critical paths of Anderson's lock are:

$$\begin{aligned} t_{ho} &= 2r_l + r_g + 2w_g + 3i + b_c + \overline{b}_c/2 && (1)\\ t_{opt} &= 5r_l + 3w_l + a_l + 5i + b_c && (2)\\ t_{pes} &= 4r_l + 2w_l + r_g + w_g + a_g + 5i + b_c && (3)\end{aligned}$$

The optimistic case is particular in this lock. Since elements in the lock array are used sequentially regardless of whether there is a queue or not, the lock must have been taken P_{max} times by the local processor for the entire state to be in the cache. Consequently, for mixed usage t_{opt} will be almost as bad as t_{pes}.

3.2 Graunke and Thakkar's queue lock

Graunke and Thakkar [6], independently from Anderson, developed a queue lock with similar characteristics.

The lock requires only atomic swap. Whereas Anderson's lock requires a fixed allocation of lock flags equal to the maximum number of processors that will ever contend for the lock, Graunke and Thakkar's lock requires a fixed allocation of lock flags equal to the number of processors regardless of their contention patterns.

[1] In practice its smallest size is dictated by the atomic operations available. A minimal size close to 1 bit is achievable by locking a vector of flags.

The critical paths for the GT lock are:

$$t_{ho} = 3r_l + r_g + w_g + 3\frac{1}{2}i + b_c + \overline{b}_c/2 \quad (4)$$

$$t_{opt} = 7r_l + w_l + s_l + 8i + b_c \quad (5)$$

$$t_{pes} = 6r_l + w_l + r_g + s_g + 8i + b_c \quad (6)$$

3.3 The MCS lock

Mellor-Crummey and Scott have described a queue-based spin lock that spins on local data, requires $O(N+P)$ space, and guarantees FIFO [11, 12]. A global pointer is maintained for each lock. If there is a queue, this pointer points to a record allocated by the enqueued processor. Subsequent processors that wish to queue on the lock update the pointer to point to a new record, and subsequently update the record previously pointed to to complete the link. The lock requires an atomic clear-if-equal primitive.

Mellor-Crummey and Scott also describe a version of the MCS lock that only requires an atomic swap. This version has the disadvantage that fairness is no longer guaranteed, and it is slightly more complex.

Critical paths for the MCS lock are:

$$t_{ho} = r_l + 2r_g + w_g + 2b_c + \overline{b}_c/2 \quad (7)$$

$$t_{opt} = 5r_l + w_l + c_l + s_l + 2b_c + \overline{b}_c \quad (8)$$

$$t_{pes} = 5r_l + w_l + c_l + s_g + 2b_c + \overline{b}_c \quad (9)$$

Notice that the clear-if-equal instruction is done only locally in any critical path.

The version of the MCS lock that uses only the swap primitive, though more complex, has very similar critical paths. In equations 8 and 9 above, c_l is replaced with $s_l + i$. Equation 7 is unaffected. The complicated cases occur when a processor that releases a lock erroneously detects that the queue is empty. In other words, no additional global operations are induced in the critical paths.

3.4 The LH lock

The main objective for the LH lock is to minimize hand-off cost. The releasing processor should not need to make any global memory accesses in order to determine where to write for the next processor to be notified. This implies that the state of the queue upon acquiring the lock must be allowed to change without affecting the hand-off code.

Once a flag has been set to indicate that the lock is free, the processor has two choices. Either synchronize (in some manner) with the next processor, or just release the lock and go away. The LH lock selects the latter route. By choosing not to synchronize, the LH

```
lh_acquire(int **L, **I, **P)
{  **I = 1;
   atomic {  /* swap */
       *P = *L;
       *L = *I; }
   while (**P != 0) ; /* spin */ }

lh_release(int **I, **P)
{  **I = 0;
   *I = *P; }
```

Figure 2: The LH lock

algorithm can release a lock by marking a flag free and discarding the flag. This means that the release code ignores whether or not there is a queue (we obviously cannot do the same on acquire).

If there is no queue, ownership of the flag must be picked up by some other processor. In the meantime, the queue data structure must refer to the flag since it is the only item that indicates that the lock is free.

From this reasoning, the LH flag falls neatly in place. We acquire a lock by switching a flag with the lock. If the flag we receive is set, we spin on it—it will eventually be released by the processor before us in the queue. If it is not set, we're free to enter the critical section. The flag we gave to the lock is obviously set, preventing other processors from entering. When we release the lock, we clear the flag we initially put in the lock and then discard any reference to it. If a processor has tried to take the lock after us, it will be spinning on this flag. We keep the flag previously owned by the lock and can reuse it the next time we try to take a lock.

In other words, the processors and locks share a name-space of $(L+P)$ flags, where L is the number of locks and P is the number or processors. Locks each own a (free) flag initially. Processors can arrive and leave dynamically, as long as they contribute/dispose of flags in an orderly manner.

Figure 2 shows the pseudo-C code for the LH-lock. The same lock has been developed independently by Craig [3].

Each lock requires an initial global pointer, `L`, that points to a global memory space for a flag. The flag is initially set to free.

The location of the pointer is statically known by all processors. In addition, each processor requires two pointers, `I` and `P`, as well as a space for a flag, for each lock that will be held. If a processor will require holding at most `n` locks at once, it will need `n` sets of `I`, `P`, and a flag. These cannot be allocated dynamically, but must be permanent.

Consider figure 3, where processor 1 has the lock.

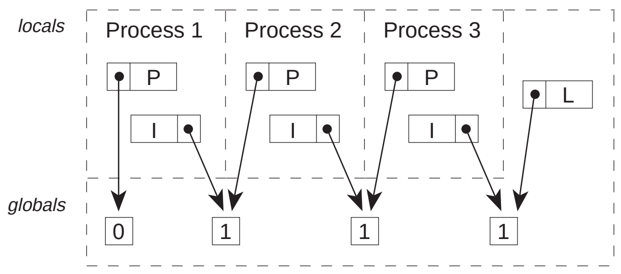

Figure 3: Enqueueing on the simple LH-lock

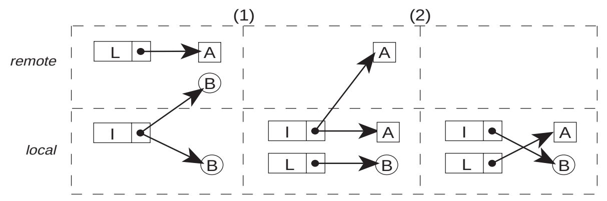

Figure 4: Cache behaviour of the simple LH lock

Here, `I` and `P` refer to different sets of pointers in each context. The flags they point to are globally accessible. Each processor spins on the flag pointed to by its local `P`, and releases the lock by clearing the flag pointed to by its local `I`. `L` points to the last flag in the queue.

The cache behaviour of the LH lock is curious. Upon entry to a lock, the locations of the flags owned by the processor and the lock will depend upon their usage in the previous two locks. Consider figure 4. The initial state is the worst case, i.e., there has been previous contention for the lock. The flag owned by the lock is located remotely, and the flag owned by the processor is replicated remotely (at the processor that held the lock immediately before we last held it).

We now assume no further contention. After the first acquire/release of the lock, the flag previously owned by the lock is now replicated locally, and the flag owned by the lock resides locally. After the second acquire/release, both flags reside locally.

If, after (1) in figure 4, the lock is acquired by another processor, both `L` and `B` will migrate. Thus, the next time around the processor will be back in the worst-case initial state.

The critical paths are:

$$t_{ho} = r_l + w_g + r_g + b_c + \overline{b}_c/2 \quad (10)$$
$$t_{opt} = 5r_l + 4w_l + s_l + b_c \quad (11)$$
$$t_{pes} = 4r_l + 3w_l + r_g + w_g + s_g + b_c \quad (12)$$

3.5 The M lock

To improve the LH lock we need to understand why it requires an additional two global accesses in the pessimistic case. The answer has already been given in figure 4. The aggressive method of enqueueing spoils the cache behaviour. The processors exchange ownership of flags on each acquire/release, therefore forcing an unnecessary global read/write on each exchange.

If the processor could know upon acquire that there is no queue, then a global read could be avoided. Similarly, if the processor knew that there was no queue upon release, it would not be necessary to switch flags with the lock. This would eliminate the the global write when `*I` is written to in the next use of the lock.

The idea is that if, upon releasing the lock, no queue has formed, then we do not need to exchange a flag with the lock. If we can avoid the switch, we will retain an already localized flag for the next acquire. However, this manipulation must be done off the critical path. In particular, it must not delay the hand-off time. The reason is that if there's a queue, `L` will no longer be local, and hence we will be inducing a global access in the critical path. This is in fact one of the problems with the MCS lock, where an "optimization" is to read the local pointer that, if a queue has formed, will no longer be local. Therefore, we wish to keep the core advantage of the LH lock, namely, that only a single global write is needed to notify that the lock is free.

In order to ascertain that the queue has indeed not changed, it is not sufficient to compare `*L` with `*I`. The reason is that once we've released the flag pointed to

```
m_acquire(struct lock *Q, int **I, **K)
{  int old_id;
    **I = 1;
    atomic {            /* swap */
        *P = Q->L;
        old_id = Q->id;
        Q->L = *I;
        Q->id = myid; }
    if (old_id > 0)
        if (*K)
            free_flag(*K);
        while (**P != 0) /* spin */ ;  }

m_release(struct lock *Q, int **I, **K)
{  int failed = 0;
    **I = 0;
    atomic {            /* compare-and-clear */
        if (Q->id == myid) {
            Q->id = 0;
        } else
            failed = 1; }
    if (failed)
        if (*K) {
            *I = *K;
            *K = 0;
        } else
            *I = alloc_flag();  }
```

Figure 5: `m_acquire()` and `m_release()`

	Hand-off ($t_{nq} + t_d$)	Optimistic ($t_{rl} + t_r$)	Pessimistic ($t_l + t_r$)
Anderson	$r_l + w_g + 3i$	$2w_l + r_l + a_l + 5i$	$r_g + w_g + a_g + 2w_l + 5i$
GT	$2r_l + 3\frac{1}{2}i$	$3r_l + s_l + 8i$	$2r_l + w_l + r_g + s_g + 8i + b_c$
MCS	$r_g + b_c$	$r_l + s_l + c_l + b_c + \bar{b}_c$	$r_l + w_l + s_g + c_l + 2b_c + \bar{b}_c$
LH	0	$r_l + 3w_l + s_l$	$3w_l + w_g + s_g + r_g + b_c$
M	i	$w_l + s_l + c_l + 2i + b_c$	$2w_l + c_l + s_g + 2i + 2b_c$
Baseline	$r_l + r_g + w_g + b_c + \bar{b}_c/2$	$4r_l + w_l + b_c$	$4r_l$

Table 1: Summary of queue lock performance for critical paths

	Anderson	MCS	GT	LH	M
Hand-off ($t_{nq} + t_d$)	$r_g + 2w_g$	$2r_g + w_g$	$r_g + w_g$	$r_g + w_g$	$r_g + w_g$
Pessimistic ($t_l + t_r$)	$r_g + w_g + a_g$	s_g	$r_g + s_g$	$r_g + w_g + s_g$	s_g
Optimistic ($t_{rl} + t_r$)	16	14	19	12	14
Lines of code	15	18	18	11	24+
Atomic operations	fetch_and_inc	c_&_c (swap)	swap	swap	swap + c_&_c
Number of flags	LP_{max}	$P_q + n_{avg}P$	$L(P+1) + P_q$	$L + nP$	$(n+1)P$
Number of pointers	$L + P_q + n_{avg}P$	$L + P_q + n_{avg}P$	$L(P+1) + P_q$	$L + nP$	$(n+1)P + P_q + L$

Table 2: Queue lock comparison

by `*I`, we're creating a racing condition for the value of `*L`.

Therefore, we must uniquely identify the processor that last modified `*L`. We do this by merging two values into `*L`—the pointer to a flag, and the id of the processor that last changed the value. Only if this value has changed do we write back the old pointer value. This is easily arranged with 64-bit or even 32-bit integers.

The algorithm for the M lock is shown in figure 5. The trailing `*K` maintains an old `*P` value. Notice that `free_flag()` is called when the processor is about to spin anyway, and `alloc_flag()` is called only when a queue has formed. This way, both segments of code are outside the critical paths. As with the previous version, myid is numbered 1 and up.

If a queue forms, the processor that began the queue "volunteers" a flag. Upon leaving the queue, it may or may not still have a flag in `*K`. If not, it needs to allocate one. Processors that join an existing queue, however, simply replace `*I` with an old `*P` *after* leaving the queue.

The critical paths of the final M lock are:

$$t_{ho} = r_l + r_g + w_g + i + b_c + \bar{b}_c/2 \qquad (13)$$

$$t_{opt} = 4r_l + 2w_l + s_l + c_l + 2i + 2b_c \qquad (14)$$

$$t_{pes} = 4r_l + 2w_l + c_l + s_g + 2i + 2b_c \qquad (15)$$

4 Comparison of locks

Table 1 summarizes the critical paths for all the locks we've looked at in this paper. The "baseline" entry contains the minimum of all the studied locks, and has been subtracted from all the locks to facilitate comparison. In table 2, we include only the global accesses from table 1, together with other important characteristics of each lock. As the tables indicate, we have six criteria: performance for the three critical paths, size of code, atomic operation requirements, and memory requirements. In this discussion, we consider only the version of the MCS lock that is fair: this is reasonable, since we are searching for a general lock (i.e., library implementation).

Two of the critical paths, hand off and pessimistic, are easy to draw conclusions from. For large multiprocessors, global memory accesses will be considerably more expensive than any of the other operations, and so we need only consult the first two rows of table 2. Here, the M lock is superior or equivalent to all the alternatives.

The third critical path, the optimistic case, is more dependent on architectural details. If, as an example, we assume the following relative costs:

$$\bar{b}_c = 2b_c = 2i = 2w_l = 2r_l = s_l = c_l = a_l \qquad (16)$$

then we arrive at the values in the third row in table 2. The simple design of the LH lock pays off, but the MCS and M locks are only 16% slower.

The code size is clearly in favour of the LH lock. The 24+ entry for the M lock means that, in addition to 24 lines of code, there are 2 procedure calls. This is more than twice as much as the LH lock, with the MCS and GT locks in-between. The smaller the code, the less of an impact for in-lining the locks.

The compare-and-clear can be just as easily implemented as atomic swap on current architectures, such as the R4000 and Alpha [13, 4]. Hence we are not too concerned with the atomic operation requirements. Nevertheless, if only an atomic swap is available, only the LH and GT locks qualify.

Memory requirements fall into two categories: $O(LP)$ vs $O(L+P)$. One of the motivations for developing the MCS lock was indeed that it is $O(L+P)$, and the LH and M locks also achieve this.

Anderson's lock is clearly not competitive any more, requiring $O(LP)$ memory, a fetch-and-increment primitive, and more global operations than the other locks. The GT lock also requires $O(LP)$ memory, and it's advantage of requiring only an atomic swap primitive is equaled by the LH lock.

The MCS lock is competitive, but requires two global reads for hand-off compared to one read for the LH and M locks. With weaker consistency models, writes will typically be handled by a write buffer, thus global reads will dominate in large machines with fast processors.

This discussion makes evident that the M lock is either clearly superior or acceptable. It's drawbacks are more complex code and its requirement of a compare-and-clear primitive. It requires only $O(L+P)$ memory, and is either clearly faster or competitive in all critical paths of execution.

5 Conclusion

We have presented a detailed discussion of the performance, memory requirements, code size, and minimal hardware support for previously published queue locks, including two new locks. The performance of the locks was studied by deducing the important critical paths and comparing the time required for all locks along these paths in terms of the primitive operations required.

We conclude that for a class of architectures, large cache-coherent shared-memory multiprocessors, the M lock is the overall preferred choice. If code requirements are tight and we do not wish the overhead of procedure calls, the LH lock provides good performance for smaller architectures and requires only an atomic swap.

Acknowledgments

We'd like to thank the SICS sponsors for their continued support: ABB, Swedish Defense Material Administration, IBM Sweden, CelsiusTech, Ericsson, and Telia.

References

[1] R. Alur and G. Taubenfeld. Results about fast mutual exclusion. In *Proceedings of the 1992 Real-Time Symposium, Phoenix, Arizona, December 2-4*, pages 12–21. IEEE, 1992.

[2] T. E. Anderson. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[3] T. S. Craig. Building FIFO and Priority-Queueing Spin Locks from Atomic Swap. Technical Report 93-02-02, Department of Computer Science and Engineering, FR-35, University of Washington, Feb. 1993. Available via anonymous ftp from "cs.washington.edu" as "tr/1993/02/UW-CSE-93-02-02.PS.Z".

[4] Digital Equipment Corp. *Alpha Architecture Handbook*, 1992.

[5] J. R. Goodman, M. K. Vernon, and P. Woest. Efficient Synchronization Primitives for Large-scale Cache-Coherent Multiprocessors. In *ASPLOS*, pages 64–75, 1989.

[6] G. Graunke and S. Thakkar. Synchronization Algorithms for Shared-Memory Multiprocessors. *IEEE Computer*, 23(6):60–69, June 1990.

[7] L. Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

[8] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.

[9] G. Le Lann. Distributed systems: towards a formal approach. In *IFIP Congress, North Holland*, pages 155–160, 1977.

[10] P. Magnusson, A. Landin, and E. Hagersten. Efficient software synchronization on large cache coherent multiprocessors. Technical report, Swedish Institute of Computer Science, February 1994.

[11] J. Mellor-Crummey and M. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, pages 21–65, 1991.

[12] J. Mellor-Crummey and M. Scott. Synchronization Without Contention. In *ASPLOS*, pages 269–278, 1991.

[13] *MIPS R4000 Microprocessor User's Manual*, 1991.

[14] G. F. Pfister and A. Norton. "Hot Spot" Contention and Combining in Multistage Interconenction Networks. *IEEE Trans. Comput.*, October 1985.

[15] L. Rudolph and Z. Segall. Dynamic Decentralized Cache Schemes for MIMD Parallel Processors. In *ISCA*, pages 340–347, June 1984.