

Latency Impact on Spin-Lock Algorithms for Modern Shared Memory Multiprocessors

Jan Christian Meyer and Anne C. Elster
Norwegian University of Science and Technology
Department of Computer and Information Science
Sem Sælands v. 7–9, NO-7491 Trondheim, Norway
janchris@idi.ntnu.no and elster@idi.ntnu.no

Abstract

In 2006, John Mellor-Crummey and Michael Scott received the Dijkstra Prize in Distributed Computing for their 1991 paper on algorithms for scalable synchronization on shared memory multiprocessors, which included a novel spin-lock algorithm (a.k.a. MCS spin-lock) that carefully distributes spin locations in memory to lessen the impact of bandwidth limitations on spin algorithms. Their empirical work and architectural suggestions have had a major impact on how the field has viewed spin-locks.

Motivated by emerging architectures with an increasing number of cores, we present an empirical study on recent shared memory architectures, including IBM P5+ and SGI ccNUMA systems. Our results show that latency will have a much greater impact on performance than bandwidth on these and future architectures with many cores and private caches. Several testcases and a tabular overview of our results are included.

1. Introduction and Motivation

Shared memory distributed computing is becoming more and more important as multicore architectures are becoming the standard way of dealing with power and frequency "walls". With increasingly many cores fighting for shared resources, how resource allocation is handled will have a major impact on performance.

A process may request exclusive access to a resource by using a spin-lock which polls a shared data structure in a tight loop, "busy-waiting" or "spinning" until the state of the shared structure indicates that the lock is acquired by the requesting process.

Spin-locks are typically used to protect short critical sections where the time to suspend/resume a requesting process is greater than the time spent on polling the shared variable.

However, the design of a suitable data structure for spin-locks requires consideration of both scalable performance and fairness. A rapid growth in the number of requests for a shared data structure can potentially saturate the memory system, and a poorly designed lock may cause starvation.

The motivation for this study arose from the cache coherency protocols of the Stanford DASH[5] architecture as adopted on SGI platforms with ccNUMA interconnects[4], and the outline of the Power5 cache system given by Kumar et. al.[3]. Both of these designs share the property that write requests are buffered enroute to their destination, which provides an opportunity to arbitrate memory access by transmitting negative acknowledgements to requests in transit before they are able to saturate the interconnect. Acknowledging that this potentially alters the relationship between the actual performance of a shared spin location and the common preconception of their inefficiency, our study represents an attempt to quantify the significance of applying spin-lock algorithms which use a software approach to co-locate spin locations with their respective processors.

2. Historical Notes and Related Work

The use of a spin-lock algorithm for mutual exclusion was first suggested in a concise paper by Dijkstra[2], which proposed an algorithm which resolved conflicts due to concurrency by awarding the lock to the process which issued the last executed write request.

The acclaimed empirical work of Mellor-Crummey and Scott[7] from 1991 examined a range of spin-lock and barrier algorithms based on fetch-and- ϕ operations (a class of atomic read-modify-write instructions). Their work presented strong empirical evidence that significant performance improvements could be made by paying attention to how spin locations are distributed in memory. Their work introduced novel algorithms for both barriers and spin-locks, most notably the MCS spin-lock, which exploit this

bandwidth limiting effect.

Part of their contribution was to give evidence that the problems associated with contention for a shared location could be reduced by a sufficiently sophisticated software algorithm, although this was commonly thought to require special-purpose hardware at the time.

This work and the architectural suggestions in their paper has had a major impact on how the field has since viewed the importance of local spins. Consequently, in a literature which spans the range from theoretical correctness proofs[1] through detailed program analysis and simulation[6, 8] to empirical studies[9], great attention has been given to ensure that locking algorithms spin only on memory addresses which can be co-located with the spinning processor, either in a local share of global memory, or in cache memory.

Magnusson et. al.[6] compare algorithms purely on the basis of the number of global memory references in their run-time analyses. Their paper includes detailed run-time analysis of three queue-based locks including the MCS lock, and propose two new locks which reduce lock release costs.

Michael and Scott[8] compare the performances of various implementations of fetch-and- ϕ operations, compare-and-swap and load-linked/store-conditional (LL/CS) in a simulated MIPS R4000 64-node multiprocessor, and make architectural suggestions for future multiprocessors based on their results.

A recent paper by Anderson and Kim[1] use a time complexity measure based on the number of remote memory references, which is defined as the number of references which cause interactions over the interconnect. They generalize spin-locking algorithms with respect to the atomic operation employed, and introduce a ranking system for the relative power of various fetch-and- ϕ operations. Given atomic operations of a suitable rank, they use the ranking system to prove time bounds for correct and starvation-free mutual exclusion on cache-coherent and distributed shared memory architectures.

Both Anderson and Kim[1] and Yang and Anderson[9] present asymptotic time complexities measuring time in terms of number of remote memory references.

3. Background

Algorithms with explicit management of memory placement result in increased code complexity. In order to make an informed tradeoff between performance and code complexity, it is therefore useful to know the projected performance impact. Given the amount of attention devoted to the concept of local spins in the literature published since the Mellor-Crummey and Scott article[7] established their significance, it is interesting to note that already at that time,

cache coherent memory was observed to counteract the effect of contention in barrier algorithms. This is because the spinning associated with such algorithms mostly consists of extended strings of *read* operations in a wakeup phase, permitting a shared spin location to be cached (offloading the interconnect), and using the eventual cache invalidation as a broadcast wakeup signal.

Note that simple spin-lock algorithms do not enjoy this property, since their spins consist of lots of *write* requests that require testing against a shared memory location, introducing traffic which may saturate the interconnect.

However, many developments pertaining to the implementation of coherent cache memory have taken place since the saturation effect was first pointed out. In particular, current architectures come with notions of a memory hierarchy far more complex than the coherent cache/shared bus combination of the Sequent Symmetry which was used to establish the barrier results in [7].

4. Target Platforms

Motivated by emerging multicore architectures with an increasing number of cores, we selected the following two platforms, both with 16 or more processors in a shared memory system as our test beds: an SGI Origin 3800 (up to 32 processors tested), and an IBM p575+ system (up to 8 dual cores tested).

The Origin 3800 features a ccNUMA interconnect which transparently translates a shared address space across the machine. 4 MIPS 14000 processors clocked at 600 MHz share a memory module via a crossbar, and these 4-way units are interconnected in a fat tree.

The System p575+ restricts shared memory to a node, which consists of 8 dual-core Power 5 processors, clocked at 1.9GHz. Each pair of cores share a level 2 cache, and 4 pairs are completely interconnected in a multi-chip module.

Both platforms feature ‘load linked’ and ‘store conditional’ (LL/SC) instructions. These instructions work in pairs: LL is a read operation which marks a location as read, and a corresponding SC is allowed to write to the same location only if the value went unmodified in the meantime. This enables a processor to support the full range of fetch-and- ϕ operations without extending the instruction set, since they can be implemented in terms of short sequences of instructions with the final SC instruction failing if the entire operation was not carried out atomically.

We expect that the behaviors observed on these systems will be similar to future multicore architectures with private caches.

Although LL/SC can be used for all the semantic properties of fetch_and_ ϕ operations, the Origin 3800 also supports some fetch-and- ϕ operations natively, most notably `fetch_and_increment`, and `fetch_and_decrement`. Following

the recommendations made by Michael and Scott[8], these depend on the use of special-purpose uncached memory (called *atomic reservoir memory* in the Origin series). A similar mechanism is mentioned in the description of the Stanford DASH[5].

The properties of this feature are not examined here since atomic reservoir memory is not cached, making it less relevant with respect to examining effects of memory hierarchy on spin-locks.

5. Experimental Methodology

All our implementations are based on pseudocode given by Mellor-Crummey and Scott[7]. The test suite includes implementations of a straightforward *test_and_set* lock, a version with exponential backoff, a *test_and_test_and_set* lock, the ticket lock with proportional backoff, as well as Anderson's lock and the MCS lock.

The same program code was used in the experiments on both platforms, except for a few conditionally compiled macros to handle the minor idiosyncrasies of each platform. This was done in the interest of producing comparable results. All locks were implemented using *compare_and_swap* operations available in system libraries on the respective systems. Since the IBM p575+ *compare_and_swap* does not preserve the comparison value when the operation fails, we wrapped the call in a conditionally compiled macro, in order to provide identical semantics on both platforms without introducing the extra overhead of a function call. We verified that the generated assembly code of our lock implementations in fact used the respective LL/SC instructions.

Since neither of the target platforms provides directly addressable local memory per processor, the effect of locality had to be reproduced by manipulating the memory layout of the lock data structures to exploit cache memory. This was done by padding the data structures so that each value which should be locally accessible was separated by the length of a cache line.

As a control experiment, a modified version of the *Anderson* lock which was *not* padded was tested to see if it would result in a lock which consistently performed considerably worse than the original, padded *Anderson* lock. This predicted behavior is readily observable in the collected timings.

Each of the presented timing results represents the average lock acquisition time of 5000 locks, acquired and released in a tight loop, with or without a small critical section between acquisition and release. For each case, 75 such tests were run, and the presented result is the median from these tests, hence reducing system induced variances.

5.1. Tested Locks

The *test_and_set* lock maintains a single global variable for locking, and has each contesting processor waiting in a tight loop, attempting to set it atomically. The *test_and_test_and_set* lock lowers the bandwidth requirements of the *test_and_set* lock by reading the present lock value to determine whether it can be acquired before attempting an atomic update. The *test_and_set* lock with exponential backoff, on the other hand, responds to a failed atomic update by waiting for a basic time period before attempting another update. This waiting period is doubled with each successive failure.

The *ticket* lock maintains two global counters which track acquisition attempts and lock releases separately, effectively forming a FIFO queue (thus eliminating starvation). Each failed attempt to acquire the lock results in a waiting period which is proportional to the length of the queue at the time of the attempted acquisition.

The *Anderson* lock also uses a queue, but distributes the target locations of the waiting spins in an array. The array locations are chosen so that only the spinning processor and its predecessor in the queue are accessing them. This attempts to reduce contention to a greater extent than the global counters of the *ticket* lock, while preserving its starvation freedom.

The *MCS* lock uses a similar construct, but replaces the array in the *Anderson* lock with a linked list, moving the significant part of the lock data structure into processor-local memory.

For a more detailed description of the operation of the various locks, we refer to Mellor-Crummey and Scott's original article[7].

6. Results

Our results are presented in Figures 1 through 5.

Fig. 3 presents the same material as Fig. 2, except for the removal of the *test_and_set* lock with exponential backoff, to emphasize the differences between the remaining locks using a more appropriate scale.

Tables 1, 2 summarize the result material for the tests with a critical section, describing relative lock performance and scalability for each combination of target architecture and lock type. The results with immediate lock release are omitted from these summaries for the sake of brevity, since the tests with critical section both capture all behavioral differences, and are likely to be of greater practical importance. Acquisition time is categorized according to performance relative to other locks tested under the same conditions. Scaling properties are noted where a clear tendency is visible already by the limited number of processors

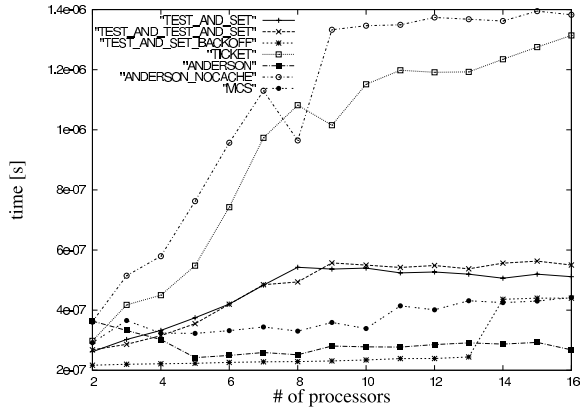


Figure 1. Performance of all locks on IBM System p575+, no critical section.

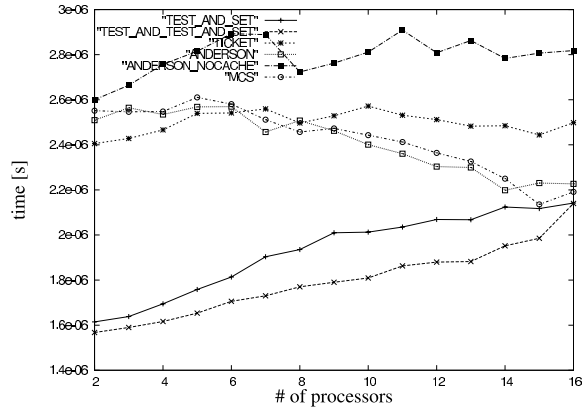


Figure 3. Performance of selected locks on IBM System p575+, small critical section.

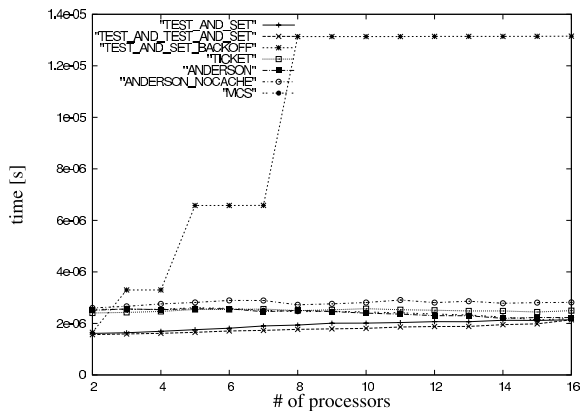


Figure 2. Performance of all locks on IBM System p575+, small critical section.

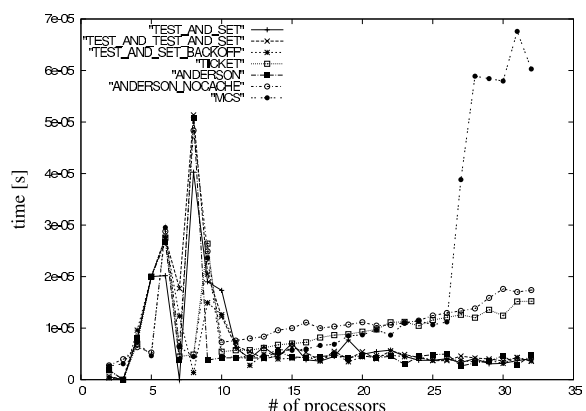


Figure 4. Performance of all locks on Origin 3800, no critical section.

in the results. Locks which exhibit favorable scaling properties are labelled with the highest number of processors for which this is observed. Entries of particular significance to our conclusions are highlighted in boldface.

6.1. Discussion

The most striking feature of the presented results can be found in the difference between Figures 1 and 2, where the *test_and_set* lock with exponential backoff goes from displaying favorable acquisition times with no critical section, to showing drastic increases when there is a critical section. In Fig. 2, the exponential backoff is visible in the shape of the graph. It is evident that the waiting periods dominate lock acquisition time, as a given number of processors implies that acquisition time becomes proportional to one of the power-of-two multiples of the basic delay. The slope

of this effect will depend on that basic waiting time as well as the degree of contention, but the characteristic behavior indicates that this lock needs to be tuned with a number of processors in mind, and thus is poorly scalable. The timings from the Origin 3800 in Fig. 4, 5 display the same behavior, albeit slightly more erratically.

Note that Fig. 5 in particular displays a sharp jump in acquisition times, corresponding with the need to traverse another level of switching for address translation to complete the memory access requests of the starvation-free locks. This consideration leads us to the central observation of this work, which is that when considering the performance implications of local spins, *interconnect latency has supplanted limited bandwidth as the primary reason for degradation*, at least on the architectures under examination. The presented results consistently bear witness to this in two ways:

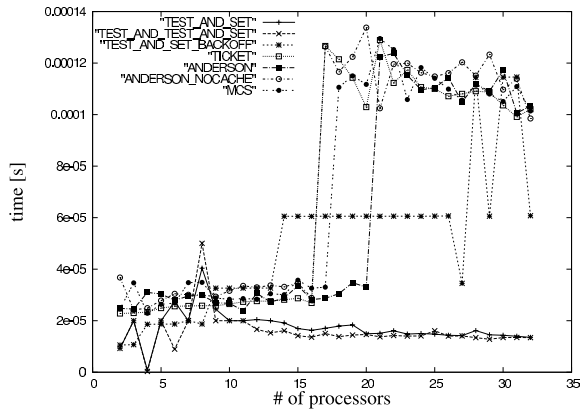


Figure 5. Performance of all locks on Origin 3800, small critical section.

Lock type	FIFO	Acq. time	Scaling
<i>Test&set</i> <i>Test&test&set</i>	No	Low	#cpus bound
<i>Test&set w.backoff</i>	No	High	#cpus bound
Ticket <i>Anderson</i> <i>MCS</i>	Yes	High	Scales for $p < 16$

Table 1. Summary of Results With Critical Section, IBM p575+

- The variations on the *test_and_set* lock display scalable characteristics in all the performed tests. This would not be true if acquisition time was dominated by the hardware struggling to serialize an increasing number of write requests for a single, shared location.
- The performance of the *ticket* lock appears similar to the *MCS* and *Anderson* locks, particularly in the cases where a critical section is present. The common property shared by these locks is that they maintain a FIFO ordering on lock acquisitions, forcing a modest number of remote write operations when the lock is handed to a remote process. The time taken to perform these remote writes visibly dominate the performance degradation due to contention for a shared location: on the IBM PSeries 575+ there is a small benefit from laying out spin locations in processor-local caches (Fig. 3), while the Origin 3800 shows no observable advantage (Fig. 5).

The implication for lock design is that the benefits from the effort of ensuring that spin locations are tightly bound to each single processor may reasonably be called into ques-

Lock type	FIFO	Acq. time	Scaling
<i>Test&set</i> <i>Test&test&set</i>	No	Low	Scales for $p < 32$
<i>Test&set w.backoff</i>	No	High	#cpus bound
Ticket <i>Anderson</i> <i>MCS</i>	Yes	High	Latency bound

Table 2. Summary of Results With Critical Section, SGI Origin 3800

tion. Instead of asking how many remote write operations are required for a given locking algorithm, the presented results indicate that it is equally relevant to ask exactly how remote these operations are likely to be.

The results for less than 10 processors on the Origin 3800 are inconclusive with respect to which lock has the superior acquisition time, particularly for the cases with immediate lock release in Fig. 4, but also for the *test_and_set* and *test_and_test_and_set* locks with critical sections in Fig. 5. A relevant comment to this is that during the experiments, it proved particularly tricky to get any measure of stability from timings collected on this machine. This is at least partly due to the fact that the transparent address translation of the ccNUMA architecture gives the operating system great freedom in the (re-)scheduling of processes at run time, effectively making it extremely difficult to predict the layout of processes. The noisy result material is included here for completeness, modest as its information value may be for comparisons.

Another interesting point is that in Figures 3 and 5 *test_and_set* and *test_and_test_and_set* locks appear to be superior to all other tested algorithms. This effect is most likely due to the lack of fairness in these locks. Processors located near the memory module of the shared spin location may have an advantage in competing for the lock. All the starvation-free locks have greater overhead than the simple *test_and_set* varieties.

Since this effect is most visible for the higher numbers of processors on the Origin 3800, some preliminary tests with logging of which lock is awarded to which processor were performed for the 32 node *test_and_set* case. Surprisingly, the tests revealed no significant bias for any subset of processors. This result is sufficiently unexpected to warrant a more systematic quantitative examination, but is beyond the scope of the present work, which is chiefly concerned with examining the impact of locality upon acquisition time.

7. Conclusions and Future Work

In contrast to previous observations on older architectures, this work showed that neither the *test_and_set* nor the *test_and_test_and_set* lock suffers great degradation with up-scaling on the tested architectures, the IBM System p575+ and SGI Origin 3800. This suggests that the interconnect bandwidth of today's architectures and future multi-core processors may be sufficient to avoid congestion. Since our tests did not track fairness, another possibility is that the interconnect traffic was kept local due to a set of processors causing starvation in the rest. The presented material provided insufficient information to determine whether these explanations are accurate, but the consequences of distributed shared memory for the fairness of a simple lock gives an interesting direction for future study on multicore architectures.

Secondarily, the scaling properties of the *ticket* lock was observed to be similar to those of the *Anderson* and *MCS* locks. The assumption that the *ticket* lock's use of a shared spin location would saturate the interconnect and cause performance degradation led to the prediction that the *ticket* lock should scale poorly compared to the *Anderson* and *MCS* locks, yet in the practically useful cases where the lock protects a critical section, such an effect was visible only to a moderate extent on the IBM p575+, and not at all on the Origin 3800. In these cases, the locking algorithms guaranteed freedom from starvation, so our conclusion is that the interconnect bandwidth of the test platforms is sufficient to reasonably handle the greater demands of the *ticket* lock.

Thirdly, the remote writes of both the *ticket*, *Anderson* and *MCS* locks visibly dominated acquisition time in the same way for all three locks when the distance to remote memory increased on the Origin 3800. This showed that the latency of these remote writes has a more significant effect on acquisition time than saturation of the interconnect due to the greater amount of traffic generated by the *ticket* lock. Locking algorithms which are aware of the distance to remote memory is hence an interesting topic for further study.

Acknowledgements

The authors thank Prof. Lasse Natvig at NTNU and Helge Rustad at SINTEF for their valuable feedback on early versions of this work. Magnus Jahre and Rune E. Jensen at NTNU provided enlightening discussions and meticulous scrutiny of the experimental code. The many useful comments of the referees are also appreciated.

References

- [1] J. H. Anderson and Y.-J. Kim. A generic local-spin fetch-and- ϕ -based mutual exclusion algorithm. *Journal of Parallel and Distributed Computing*, 67(5):551–580, May 2007.
- [2] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):21–65, September 1965.
- [3] R. Kumar, V. Zyuban, and D. M. Tullsen. Interconnections in multi-core architectures: Understanding mechanisms, overheads and scaling. *ACM SIGARCH Computer Architecture News*, 33(2):408–419, 2005.
- [4] J. Laudon and D. Lenoski. The SGI origin: A ccNUMA highly scalable server. *ACM SIGARCH Computer Architecture News*, 25(2):241–251, 1997.
- [5] D. E. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford Dash multiprocessor. *IEEE Computer*, 25(3):63–79, 1992.
- [6] P. Magnusson, A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. In *Proceedings of the Eighth International Parallel Processing Symposium*, pages 26–29, 1994.
- [7] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory architectures. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.
- [8] M. M. Michael and M. L. Scott. Scalability of atomic primitives on distributed shared memory multiprocessors. Technical Report 528, University of Rochester Computer Science Department, July 1994.
- [9] J.-H. Yang and J. H. Anderson. A fast, scalable mutual exclusion algorithm. *Distributed Computing*, 9(1):51–60, August 1995.