SCALABLE COMPUTING: PRACTICE AND EXPERIENCE
Volume 9, Number 3, pp. 197–206. http://www.scpe.org

ISSN 1895-1767
© 2008 SCPE

LATENCY IMPACT ON SPIN-LOCK ALGORITHMS FOR MODERN SHARED MEMORY MULTIPROCESSORS

JAN CHRISTIAN MEYER* AND ANNE C. ELSTER*

Abstract. In 2006, John Mellor-Crummey and Michael Scott received the Dijkstra Prize in Distributed Computing. This prize was 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). Their spin-lock algorithm distributes spin locations in memory to lessen the impact of bandwidth limitations. Their empirical work and architectural suggestions have since 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. Several testcases and a tabular overview of our results are included.

1. Introduction. Shared memory distributed computing is becoming more and more important as multi-core architectures are used to overcome the 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, "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. The design of a suitable data structure for spin-locks requires consideration of both scalable performance and fairness. As pointed out by Mellor-Crummey and Scott [1], interconnects may be saturated by a large number of requests for a shared data structure, and a poorly designed lock may cause starvation. This is further complicated by the multiple hierarchical memory levels of today's multi-core processors.

This study arose from looking at the cache coherency protocols of the Stanford Dash [8] architecture as adopted on SGI platforms with ccNUMA interconnects [7], and the outline of the Power5 cache system given by Kumar et. al. [9]. Both designs share the property that write requests are buffered enroute to their destination. This provides an opportunity to arbitrate memory access by transmitting negative acknowledgements to requests in transit before they are able to saturate the interconnect. Since 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 [1] 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. It also introduced novel algorithms for both barriers and spin-locks, most notably the aforementioned *MCS* spin-lock which exploits this bandwidth limiting effect. Part of their contribution was to show 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. Their work and architectural suggestions have 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 [6] through detailed program analysis and simulation [3, 4] to empirical studies [5], 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. [3] 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.

*Norwegian University of Science and Technology, Department of Computer and Information Science, Sem Sælands v. 7–9, NO-7491 Trondheim, Norway, {janchris, elster@idi.ntnu.no}

Michael and Scott [4] 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 [6] 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 [6] and Yang and Anderson [5] present asymptotic time complexities measuring time in terms of number of remote memory references.

3. Background. Given the amount of attention devoted to the concept of local spins in the literature published since the Mellor-Crummey and Scott article [1] 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.

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 reported by Mellor-Crummey and Scott [1].

Based on these results, Michael and Scott [4] recommended using LL/CS to create atomic primitives for future architectures. Such instructions are hence included on modern shared-memory architectures such as the MIPS 14000 and IBM Power5. Our work empirically tests all of the spin-lock algorithms tested by Mellor-Crummey and Scott [1] on recent two recent larger shared-memory systems.

Our results question the conventional cost models which limits itself to local and remote accesses. Today's shared memory systems typically have several levels of memory hierarchy and is shown to be much more sensitive to latency rather than purely bandwith bound, as we will show in Section 6.

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 System p575+ (up to 8 dual cores tested). While these systems are not highly parallel in the multi-core sense, the authors believe that the communication issues faced by the larger scale interconnects of contemporary supercomputers will soon become relevant for emerging multi-core architectures. These architectures are rapidly approaching the same degree of parallelism, and are likely to confront the same "memory wall" which has precipitated the hierarchical memory structure of our test platforms, albeit on a chip-level scale.

The SGI Origin 3800 consists of "bricks" of 4 MIPS 14000 processors clocked at 600 MHz, which share a memory module via a crossbar. These 4-way units are interconnected in a fat tree. The system features a ccNUMA interconnect, which provides programs with a shared address space potentially spanning several physical racks of compute nodes. Transmissions of data across the interconnect is transparently handled via address translation, so that the system may be programmed as a symmetric multiprocessor. Memory traffic is handled by the same routing system regardless of the proximity of the communicating nodes, to provide the best approximation to uniform memory access the system can offer. Cache coherency in ccNUMA systems is maintained using a variant of the protocol in the Stanford Dash multiprocessor [8], using a directory approach where a snooping bus connects a small set of processors with their local part of the distributed memory. This bus is also connected with the local share of the directory, which maintains the caching status of the local lines, and forms an interface to the other parts of the directory [7].

The IBM 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.

Detailed information regarding IBM System p575+ coherency protocol is difficult to find, but some information can be gleaned from Kumar et. al. [9], who give an explicit description of Shared Bus Fabric (SBF) design, stating that "Details of the modeled design [...] are based heavily on the shared bus fabric in the Power 5 multi-core architecture" [9]. Bearing in mind that the source is not an official technical document, it will still be used here as a high-level description of the Power5 coherence mechanism. Cache coherency is maintained using a "[...]MESI-like snoopy write-invalidate protocol[...]" [9], which is implemented with a set of unidirectional, pipelined buses and a hierarchy of dedicated units which arbitrate bus usage and queue requests.

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.

Although LL/SC can be used for all the semantic properties of fetch_and_ϕ operations, the SGI 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 [4], these depend on the use of special-purpose uncached memory (called *atomic reservoir memory* in the SGI Origin series). A similar mechanism is mentioned in the description of the Stanford Dash [8].

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. The empirical results in this study are divided into a set of general experiments which form an overview of lock performances on both target architectures, and a more specific set which validates the implications of the general experiments by isolating predictable effects. This reflects the timeline of the underlying research effort, as the general experiments form a preliminary evaluation of relative performances and tentative explanations, while later experiments address some of the questions raised in the initial phase.

Although the result material is reiterated below for the benefit of the discussion, the interested reader may trace this development, as the general experiments alone formed the basis of our previous conference paper [10].

The implementations used in the general experiments are based on pseudocode given by Mellor-Crummey and Scott [1]. The test suite includes implementations of a plain *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.

Modified versions of the *test_and_test_and_set* and *ticket* locks are tested on the IBM System p575+ only. The extensions made to these locks primarily address properties of the experimental setting, and are not intended as generally useful lock designs; their purpose is to further isolate the communication properties of locks which are prone to starvation.

In the interest of producing comparable results, the same program code was used in the general experiments on both platforms, except for a few conditionally compiled macros to handle the minor idiosyncrasies of each platform. All locks were implemented using *compare_and_swap* operations available in system libraries on the respective systems. Since the IBM System 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. To reduce system induced variances, each presented result is the median of 75 runs.

Experiments with the modified *test_and_test_and_set* lock also monitored the fairness of tested algorithms, using the assumption that a fair lock will distribute the locks in all tests uniformly, awarding $\frac{N}{p}$ locks to each processor. The fairness results given are the median of 75 tests each reporting the maximal deviation from this expected value in a 5000 lock run.

5.1. Tested Locks. This section briefly describes each of the tested algorithms. We include all the locks described by Mellor-Crummey and Scott[1], as well as two versions we modified to isolate specific communication properties.

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 modified *test_and_test_and_set* lock exploits the fact that the original lock reserves an entire memory location to store one out of two states. Using integers smaller than 0 to represent the open state, and positive integers to lock, it is possible to retain memory of which contestant won the last lock without significantly altering the amount of interconnect traffic. This is done by using an identifier for the acquiring process(or) as the lock value, and releasing it by negating this number. It is thereby possible to prevent the lock from being acquired twice by the same contestant, at the cost of a single local comparison operation.

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 modified *ticket* lock uses the same mechanism, but only forces an adjustable-length tail of the queue to back off and wait. The remaining pool of processors at the head of the queue participate in a contest for a *test_and_test_and_set* lock. This modification makes the lock prone to starvation, as a single processor can enter the pool and be bypassed any number of times. The purpose of testing this lock is that the permitted length of the queue effectively puts a limit on how many lock acquisitions must take place between successive acquisitions by a single processor. It should be noted that this technique subsumes the modified *test_and_test_and_set* lock as a special case (length 1). The extra overhead of managing two structures, however, incurs an overhead which means that it is not competitive with other tested locks; results are included mainly to provide a test environment for comparing the results of variable queue length.

The *Anderson* lock uses a queue like the *ticket* lock, 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. When implemented using coherent caches as processor-local memory, its main difference from the *Anderson* lock is that the *MCS* lock requires each contesting processor to enter the linked list by performing a remote write operation to its predecessor in the list, to communicate its own spin location.

6. General Experiments. The results of the general experiments are presented in Figures 6.1 through 6.5.

Fig. 6.3 presents the same material as Fig. 6.2, 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 6.1, 6.2 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 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.

The most striking feature of the presented results can be found in the difference between Figures 6.1 and 6.2, where the *test_and_set* lock with exponential backoff goes from displaying favorable acquisition times with no

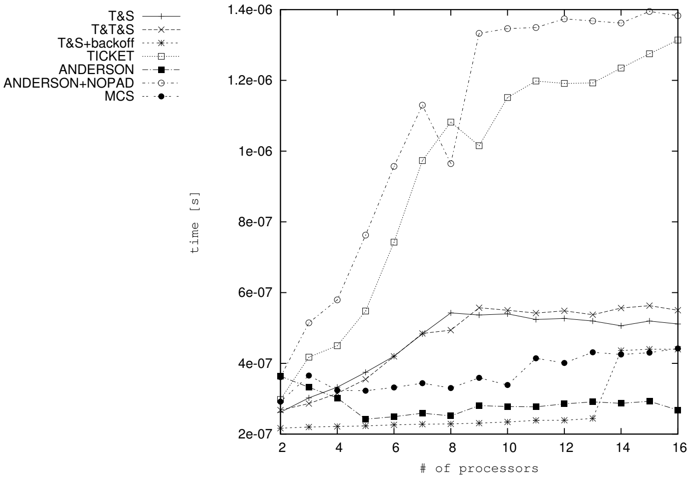

Fig. 6.1. *Performance of all locks on IBM System p575+, no critical section.*

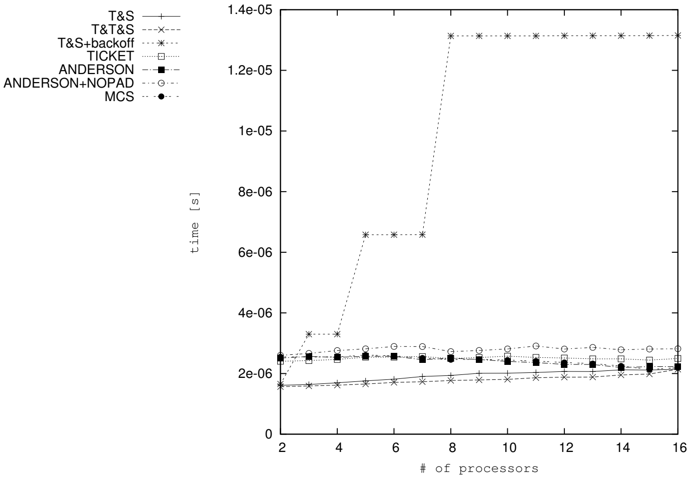

Fig. 6.2. *Performance of all locks on IBM System p575+, small critical section.*

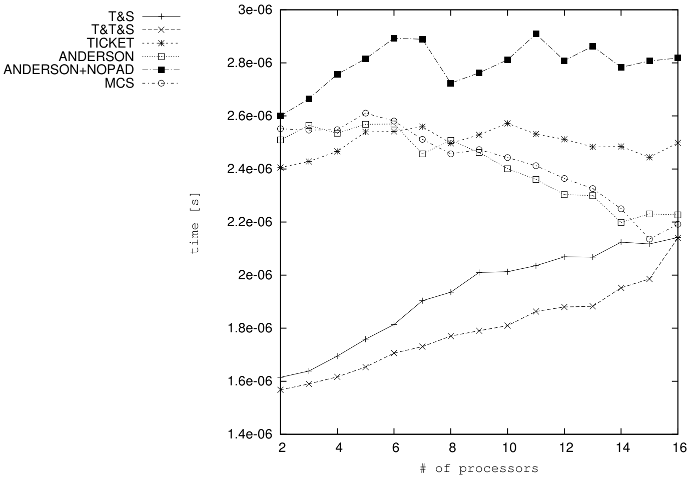

Fig. 6.3. *Performance of selected locks on IBM System p575+, small critical section.*

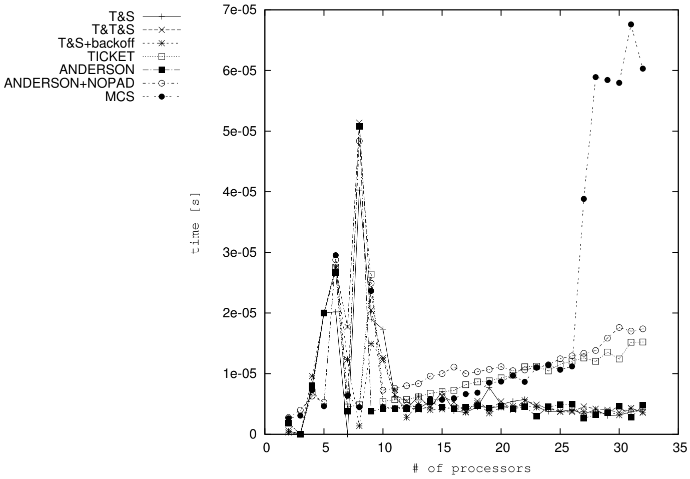

FIG. 6.4. *Performance of all locks on SGI Origin 3800, no critical section.*

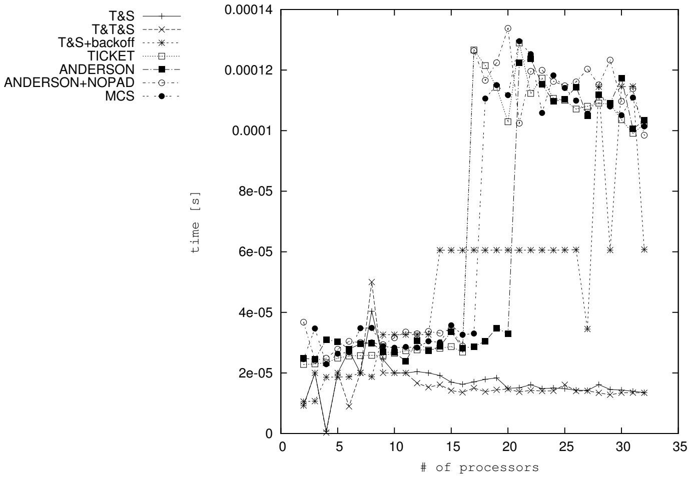

FIG. 6.5. *Performance of all locks on SGI Origin 3800, small critical section.*

critical section, to showing drastic increases when there is a critical section. In Fig. 6.2, 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 SGI Origin 3800 in Fig. 6.4, 6.5 display the same behavior, although slightly more erratically.

Note that Fig. 6.5 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 are evidence of this to this in two ways:

1. The variations on the *test_and_set* lock display scalable characteristics in all the general 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.
2. 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

Table 6.1
Summary of results with critical section, IBM System p575+

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

Table 6.2
Summary of results with critical section, SGI Origin 3800

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

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. 6.3), while the SGI Origin 3800 shows no observable advantage (Fig. 6.5).

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 question. 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 SGI Origin 3800 are inconclusive with respect to which lock has the superior acquisition time, particularly for the cases with immediate release in Fig. 6.4, but also for the *test_and_set* and *test_and_test_and_set* locks with critical sections in Fig. 6.5. 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.

7. Experiments with the Modified Test_and_Test_and_Set Lock. Figure 7.1 plots acquisition times in a repetition of the critical section experiment on IBM System p575+, with the addition of the modified *test_and_test_and_set* lock.

The two simple *test_and_set*-type locks appear to be superior to all other tested algorithms in Figures 6.3, 6.5. The common trait shared by these locks is that they contain the possibility of starvation. The observation that communication latency dominates lock acquisition times predicts low acquisition times for these locks when acquiring multiple locks in a tight loop, as a processor which has already cached the lock structure will have a significant advantage over others in the next acquistion.

The modified *test_and_test_and_set* lock is designed to eliminate the possibility for a single processor to monopolize the lock, effectively enforcing that each acquisition will pass the lock structure far enough down the memory hierarchy to reach a level accessible by multiple cores. It is still prone to starvation by permitting a small set of processors to pass the lock between them.

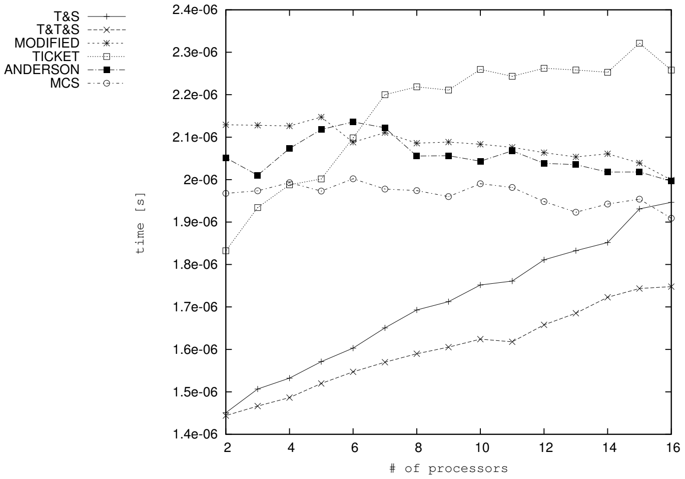

Fig. 7.1. *Performance with modified Test_and_Test_and_Set lock and critical section*

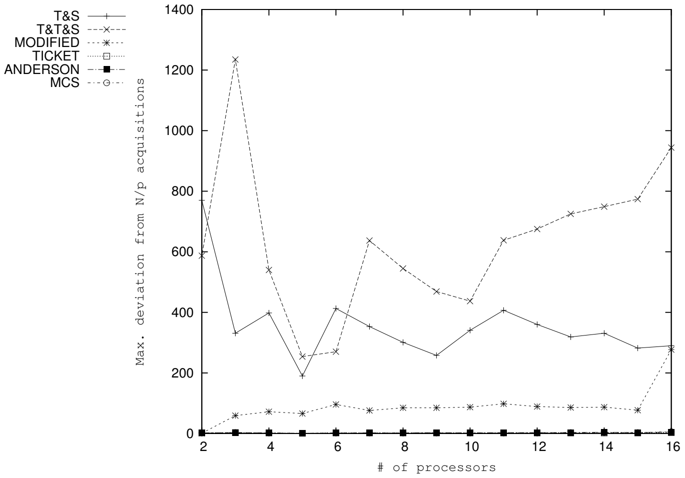

Fig. 7.2. *Fairness with modified Test_and_Test_and_Set lock and critical section*

The impact of this is observable in Figure 7.2, which plots deviations from the assumption that a fair algorithm will distribute locks uniformly among participating processors. The validity of this measure of fairness is supported by the fact that all the starvation-free locks display near perfect fairness. It is evident that the observed run times from both of the *test_and_set*-type locks are heavily biased by the increased speed of a single processor re-aquiring a recently released lock, given the marked improvement in fairness which stems from explicitly disallowing this behavior.

The other interesting feature of this experiment is that the resulting acquisition times for the modified lock closely follow those of the *Anderson* lock in Figure 7.1. Both lock structures force a remote access for each acquisition and release. This agrees with the observation that increased contention for a single memory location has an insignificant effect compared to the overhead of accessing any remote memory. Predicting acquisition time is nevertheless more complicated than counting the number of remote accesses. This is shown by the *MCS* lock outperforming both the modified *test_and_test_and_set* and *Anderson* locks in spite of requiring at least as many remote accesses for a typical acquire/release cycle, and more in the worst case.

Given the otherwise similar structure of the *MCS* and *Anderson* locks when implemented using cache coherency and padding, the main difference which separates their operation is that the *MCS* lock requires each new member of the ordered queue to register by its predecessor. This invalidates the cache line which holds its spin location, while the *Anderson* lock only invalidates the spin locations of successors, waking them when they acquire the lock. When a successor in the *MCS* algorithm causes the lock of its predecessor to be re-fetched

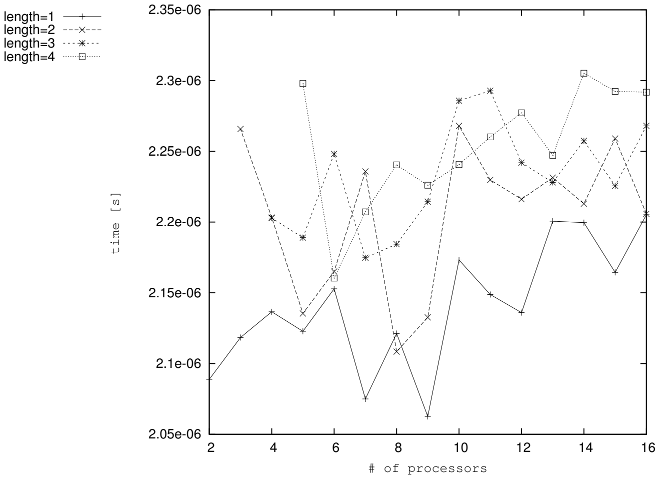

FIG. 8.1. *Comparison of modified ticket locks with variable queue length*

while it is being released, the resulting memory transfer will have some of its latency masked as a side-effect of the former invalidation. This predicts that the *MCS* lock's advantage is related to the fact that a short critical section may release the lock before a line has formed, while a long critical section will complete the establishment of a longer queue before the lock is released to its head.

Figure 6.1 shows that the absence of any critical section gives an advantage to the *Anderson* lock. We found that repeated testing involving only the *Anderson* and *MCS* locks shows that their run-time characteristics become indistinguishable when adjusting the length of the critical section by factors $\frac{1}{5}$ and 10, but distinguishable between these points. This is a curious effect to observe on an architecture which otherwise should suit the cost model of local vs. remote access (featuring private L2 caches and a crossbar interconnect [9]). As consistent performance advantages are still tied to reduced latency, and increasingly nonuniform interconnect performance can be expected from future architectures, this further supports the argument that practical communication cost models must account for processor locality to enable the differentiation of faster and slower remote accesses.

8. Experiments with the Modified Ticket Lock. The design of the modified *ticket* lock is intended to elicit similar effects to the modified *test_and_test_and_set* lock, by forcing each successive lock acquisition by a single processor to be separated by an adjustable number of other acquistions. The reason for this experiment is that the 16 cores on the test system are distributed as 8 dual cores, and the shared bus fabric of the Power5 design of the IBM System p575+ implements its snoopy cache coherency protocol by a dedicated bus which shortcuts the interconnect to update cache lines between cores on the same chip. Such updates would give an advantage to neighboring cores when acquiring a lock which does not enforce ordering. The behavior of the modified lock can hence be expected to alter when the queue length exceeds the number of cores per chip, yielding an estimate of the magnitude of the advantage lost.

Figure 8.1 shows that for the IBM System p575+, the expected change in behavior is observable between queue lengths 1 and 2. While the modified lock design naturally prevents a full set of measurements to be obtained for $p < length$, there is still a clear indication that queue length 1 gives a measurable performance advantage, while no systematic differences were found for longer queues.

This observation adds weight to the argument that future communication cost models will need locality details. While the significance of the observable difference between closely and loosely coupled interconnect on the test platform is small, its significance can be expected to increase with growing numbers of cores on a chip.

9. Conclusions and Future Work. The larger scale interconnects of recent supercomputers such as IBM Power systems and SGI's Origin series may soon become relevant for large multi-core processor systems. This article hence examines the performance of several spin-lock algorithms on such systems.

In contrast to observations from older architectures [1], this work shows that neither the *test_and_set* nor the *test_and_test_and_set* lock suffers degraded acquisition times with upscaling. Modern interconnects are not saturated by the bandwidth requirements of the tested algorithms. Unlike the older test systems which had severe bandwidth restrictions, modern interconnects feature highly connected topologies such as crossbars and fat trees. Increasing the connectivity of the interconnect greatly improves the aggregate bandwidth of the system, but has a more moderate impact on latency. Modern systems also feature snoopy cache coherency protocols between on-chip cores. Our experiments showed that locks performing well on such systems favored processes on processors near the one which last held the lock. Unfortunately, this impacts fairness.

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. However, in the practically useful cases, where the lock protects a critical section, such an effect was visible to only a moderate extent on the IBM System p575+, and not at all on the SGI Origin 3800. Our conclusion is that the interconnect bandwidth of the test platforms is sufficient to reasonably handle the greater demands of the *ticket* lock, because of its similar performance to the other starvation-free locks.

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 SGI Origin 3800. This showed that the latency of these remote writes had a more significant effect on acquisition time than saturation of the interconnect due to the greater amount of traffic generated by the *ticket* lock. This conclusion was supported by further experiments on the IBM System p575+. There it was found that starvation-prone locks achieve favorable run time when awarding locks to a single processor many times in succession. Forcing these locks to perform remote writes resulted in improved fairness, but also in run times comparable to those of the poorer performing starvation-free locks. This demonstrated that the same latency penalty applied when locks were transferred across the interconnect.

Finally, it was shown that lock exchanges between neighboring cores permitted a small performance improvement on the IBM System p575+.

Our tests only include dual-core processor nodes, but the lower latency exhibited between the cores indicates that models of future many-core architectures should distinguish between multiple levels of interconnect. Locking algorithms with a detailed measure of distances 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. M. Mellor-Crummey and M. L. Scott, *Algorithms for Scalable Synchronization on Shared-Memory Architectures*, in ACM Transactions on Computer Systems, Vol. 9, No. 1, 1991, pp. 21–65.

[2] E. W. Dijkstra, *Solution of a problem in concurrent programming control*, in Communications of the ACM, Vol. 8, No. 9, 1965, pp. 569.

[3] P. Magnusson, A. Landin and E. Hagersten, *Queue Locks on Cache Coherent Multiprocessors*, in Proceedings of the Eighth International Parallel Processing Symposium, 1994, pp. 26–29

[4] M. M. Michael and M. L. Scott, *Scalability of Atomic Primitives on Distributed Shared Memory Multiprocessors*, University of Rochester Computer Science Department, Tech. rep. #528, July 1994.

[5] J-H. Yang and J. H. Anderson, *A Fast, Scalable Mutual Exclusion Algorithm*, in Distributed Computing, Vol. 9, No. 1, 1995, pp. 51–60.

[6] J. H. Anderson and Y-K. Kim, *A Generic Local-spin Fetch-and-ϕ-based Mutual Exclusion Algorithm*, in Journal of Parallel and Distributed Computing, Vol. 67, Issue 5, 2007, pp. 551–580.

[7] J. Laudon and D. Lenoski, *The SGI Origin: A ccNUMA Highly Scalable Server*, in ACM SIGARCH Computer Architecture News, Vol. 25, No. 2, 1997, pp. 241–251.

[8] D. E. Lenoski, J. Laudon, K. Gharachorloo, W-D. Weber, A. Gupta, J. Hennessy, M. Horowitz and M. S. Lam, *The Stanford Dash multiprocessor*, in IEEE Computer, Vol. 25, No. 3, 1992, pp. 63–79.

[9] R. Kumar, V. Zyuban and D. M. Tullsen, *Interconnections in Multi-core Architectures: Understanding Mechanisms, Overheads and Scaling*, in ACM SIGARCH Computer Architecture News, Vol. 33, No. 2, 2005, pp. 408–419.

[10] J. C. Meyer, A. C. Elster, *Latency Impact on Spin-Lock Algorithms for Modern Shared Memory Multiprocessors*, in Proceedings of CISIS2008, IEEE Computer Society CPS, March 2008, pp. 786–791.

Edited by: Sabri Pllana, Siegfried Benkner
Received: June 16, 2008
Accepted: July 28, 2008