PRACTICAL PERFORMANCE ESTIMATION ON SHARED-MEMORY MULTIPROCESSORS

PAUL E. MCKENNEY
IBM Beaverton
pmckenne@us.ibm.com

1 Abstract

There has been much work done modeling, simulating, and measuring the performance of locking primitives under high levels of contention. However, an important key to producing high-performance parallel programs is to maintain extremely *low* levels of contention. Despite its importance, the low-contention regime has been largely neglected. In order to fill this gap, this paper analyzes the performance of several commonly used locking primitives under low levels of contention. The costs predicted by a number of analysis methodologies are compared to measurements taken on real hardware, thereby showing where each may be safely used.

Use of these methodologies is illustrated by an analytic comparison of different reader-writer spin-locks, and by a real-world case study.

Keywords: locking synchronization performance

2 Introduction

Maintaining low lock contention is essential to attaining high performance in parallel programs. However, even programs with negligible lock contention can suffer severe performance degradation due to memory latencies incurred when accessing shared data that is frequently modified. This is due to the high cost of memory latency compared to instruction execution overhead.

Increases in CPU-core instruction-execution rate are expected to continue to outstrip reductions in global latency for large-scale multiprocessors [1, 2, 3]. This trend will cause global lock and synchronization operations to continue becoming more costly relative to instructions that manipulate local data. Thus, the low-contention performance of locking primitives will continue to be governed by memory latency. This paper analyzes several commonly used classes of lock primitives from a memory-latency viewpoint.

Memory latencies are incurred for shared data structures in addition to the locks themselves. Therefore, simple empirical measurements of the locking primitives cannot give a complete picture of the performance of the program that uses the locks. The designer needs some way to account for the memory latencies incurred by the program's data as well as by its locks.

In short, the designer needs a technique that can provide reasonable estimates of performance based on information available at design time. This paper describes such a technique, and illustrates its use on a number of locking primitives, and in some real-world case studies.

3 Existing Solutions

The most straightforward way to measure the performance of an algorithm is to simply run it. Even so, modern microprocessor architectures complicate this seemingly simple task, so that special hardware support is often required [4]. Even with such support, this measurement approach produces results that are specific to a particular machine. Furthermore, both the algorithm and the hardware must be fully implemented and debugged, which is often infeasible during the early design phases of a project.

This work represents the view of the author, and does not necessarily represent the view of IBM.

Alternatively, an algorithm's performance may be evaluated via simulation. Although this approach can alleviate some of the measurement difficulties posed by modern microprocessor architectures, precise results require not only that the algorithm be fully coded and debugged, but that a simulator be available.

Traditional design-time methodologies for evaluating the performance of algorithms are based on operation counting [5]. This approach has been refined by many researchers over the decades. Recent work considers the properties of the underlying hardware, weighting the operations by their costs [6]. This hardware-centric approach requires detailed analysis of assembly code, and produces results that are specific to a particular machine. Magnusson's approach is nevertheless the technique of choice when exact analysis of existing short sequences of code is required.

In contrast, the techniques described in this paper are suitable for use during early design time, when the code is not yet written, let alone running. Although these techniques may be applied to both simulation and analytic operation-counting methodologies, this paper focuses on analytic methodologies.

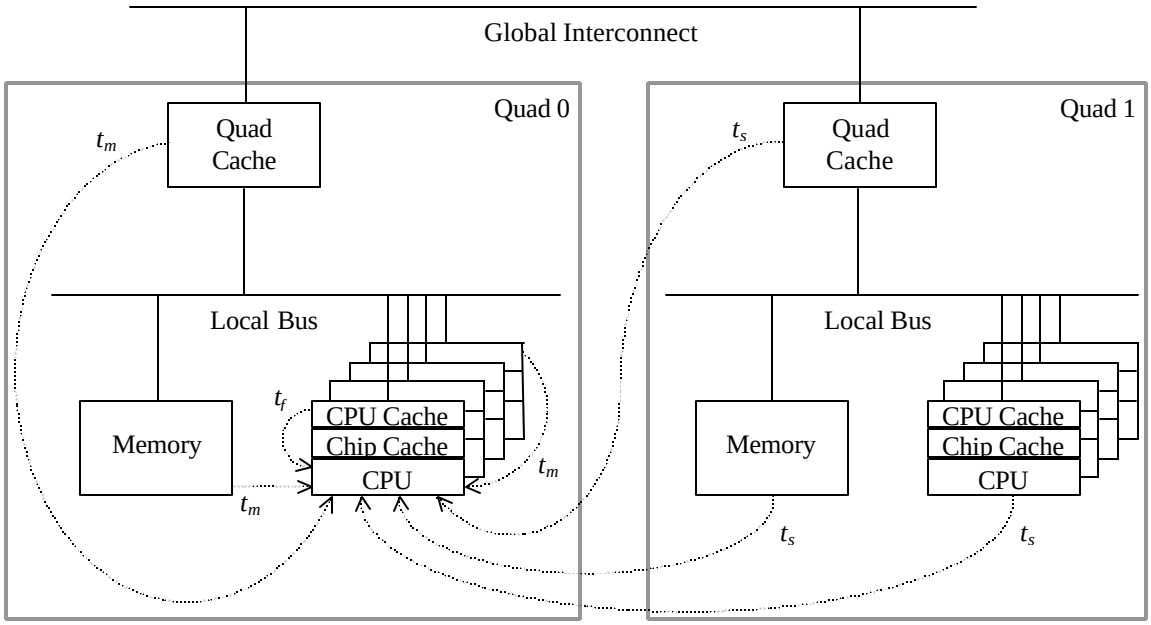

Figure 1: CC-NUMA Memory Latency

4 Memory-Latency Model

The solution put forward in this paper relies on the fact that memory latency is the dominating factor in well-constructed parallel programs. Such programs avoid highly contended locks, leaving the memory latency as the dominating execution cost, since memory accesses are increasingly expensive compared to instruction execution overhead [1, 2, 3].

Since memory latency dominates, we can accurately estimate performance by tracking the flow of data among the CPUs, caches, and memory. For SMP and CC-NUMA [7] architectures, this data flow is controlled by the cache-coherence protocol, which moves the data in units of cache lines. Figure 1 shows a cache line's possible locations relative to a given CPU in a CC-NUMA system. As shown in the figure, a CC-NUMA system is composed of modules called *quads*, which contain both CPUs and memory. Data residing nearer to a given CPU will have shorter access latencies. As the figure shows, data that is already in a given CPU's cache may be accessed with latency t_f. Data located elsewhere on the quad may be accessed with latency t_m, while data located on other quads may be accessed with latency t_s.

On large-scale machines where t_s overwhelms t_m and t_f, the latter quantities may often be ignored, further simplifying the analysis, but decreasing accuracy somewhat. If more accuracy is required, the overheads of the individual instructions may be included [6], however, this will usually require that the program be coded and compiled to assembly language, and is often infeasible for large programs.

Once a given data item has been accessed by a CPU, it is cached in that CPU's cache. If the data's *home* is in some other quad's memory, then it will also be cached in the accessing CPU's quad's cache. In both cases, the caching allows subsequent accesses from the same CPU to proceed with much lower latency. Data that has been previously accessed by a given CPU is assumed to reside in that CPU's cache (with access latency t_f). In other words, at low contention, we assume that there is insufficient cache pressure to force data out of a CPU's cache. Most modern CPUs also have a small on-chip cache, which can deliver multiple data items in parallel to the CPU in a single clock. This on-chip cache is modeled as having zero latency, but is assumed only to hold data across a single function call.

5 Conditions & Assumptions

A CC-NUMA system contains *n* quads and *m* CPUs per quad (two and four, respectively, in the example shown in the figure). The analysis makes the following assumptions:

1) Quads contain the same number of CPUs.

2) Contention is low and lock-hold times are short compared to the interval between lock acquisitions. This means that the probability of two CPUs attempting to acquire the same lock at the same time is vanishingly small, as is the probability of one CPU attempting to acquire a lock held by another CPU.

3) CPUs acquire locks at random intervals. This means that when a given CPU acquires a lock, that lock was last held, with equal probability, by any of the CPUs. Exclusive and non-exclusive accesses are assumed to occur randomly with probability *f* and *1-f*, respectively.

4) The overhead of instructions executed wholly within the microprocessor core is insignificant compared to the overhead of data references that miss the cache. The model can be extended to handle programs with a significant number of "heavyweight" instructions (such as atomic read-modify-write instructions) by adding an additional t_h for these heavyweight instructions.

5) The CPU is assumed to have a single-cycle-access on-chip cache. This cache is considered part of the CPU core, and for purposes of these derivations is called the "on-chip cache". Instruction fetches and stack references (function calls and returns, accesses to auto variables) are assumed to hit this on-chip cache, and are modeled as having zero cost. Indeed, modern microprocessors are frequently able to perform multiple accesses to this on-chip cache in a single clock cycle.

6) Cache pressure is assumed low (outside of the on-chip cache), so that a variable that resides in a given cache remains there until it is invalidated by a write from some other CPU.

7) Memory-access times are independent of the number of copies that appear in different caches. Although directory-based cache-coherence schemes can in theory deviate significantly from this ideal, in practice, this assumption is usually sufficiently accurate [7], particularly for design purposes.

8) Speculative references are ignored. In principle, speculation can result in large quantities of useless but expensive memory references, but in practice, this is often at least partially balanced by the fact that a speculating CPU can fetch multiple data items simultaneously.

6 Details of Solution

This section gives a step-by-step method of using the latency model to estimate the overhead of an

algorithm. It also describes some simplifications that may apply in some commonly occurring situations.

6.1 Summary of Nomenclature

Table 1 shows the symbols used in the derivations.

6.2 Adaptation to Large-Scale SMP Machines

The large caches and large memory latencies on large-scale SMP machines allow them to be modeled in a similar manner. In many cases, substituting t_s for t_m, 1 for m, and n for m reduces a CC-NUMA model to the corresponding SMP model.

These substitutions may be used except where the algorithm itself changes form in moving from a CC-NUMA to an SMP environment. Software that is to run in both CC-NUMA and SMP environments will generally be coded to operate well in both, often by considering the SMP system to be a CC-NUMA system with a single large quad.

	Definition
f	Fraction of lock acquisitions that require exclusive access to the critical section.
m	Number of CPUs per quad in NUMA systems. Not applicable to SMP systems. Equations that apply to both SMP and NUMA systems will define m to be one unless otherwise stated.
n	Number of CPUs (quads) in SMP (NUMA) systems.
r	Ratio of t_s to t_f.
t_s	Time required to complete a "slow" access that misses all local caches.
t_m	Time required to complete a "medium" access that hits memory or a cache shared among a subset of the CPUs. This would be the latency of local memory or of the remote cache in CC-NUMA systems.
t_f	Time required to complete a "fast" access that hits the CPU's cache.

Table 1: Nomenclature for Lock Cost Derivation

6.3 Use and Simplifications

The model is a four-step process:

1. Analyze the CPU-to-CPU data flow in your algorithm.

2. For each point in the algorithm where a CPU must load a possibly-remote data item, determine the probabilities of that data item being in each of the possible locations relative to the requesting CPU. It is usually best to make a table of the probabilities.

3. For each location, compute the cost.

4. Multiply the probabilities by the corresponding costs, and sum them up to obtain the expected cost.

This process is illustrated on locking primitives in the following sections.

One useful simplification is to set t_f and possibly t_m to zero. This greatly simplifies the analysis, and provides accuracy sufficient for many uses, particularly when the ratio r between t_s and t_f is large.

A further simplification is to assume that the data is maximally remote each time that a CPU requests it. This further reduces accuracy, but provides a very simple and conservative back-of-the envelope analysis that can often be applied to large systems during early design.

Note that because the actual behavior depends critically on cache state, actual results can deviate significantly from the analytic results presented in this paper. For example, if the CPU cache was fully utilized, the added cache pressure resulting from the larger size of higher-performance locking primitives might well overwhelm their performance benefits. Therefore, analytic results should be used only as guidelines or rules of thumb, and should be double-checked by measuring the actual performance of the running program. Nevertheless, results obtained from these models have proven quite useful in practice.

7 Analytical Analysis

This section illustrates the use of the methodology on simple spin-lock and on a number of variants of reader-writer spin-lock.

7.1 Simple Spinlock

A simple spinlock is acquired with a test-and-set instruction sequence. Under low contention, there will be

almost no spinning, so the acquisition overhead is just the memory latency to access the cache line containing the lock. This latency is incurred when acquiring and when releasing the lock, and will depend on where the cache line is located, with the different possible locations, probabilities, latencies, and weighted latencies shown in Table 2.

The entries in this table are obtained by considering where the lock could have been last held, and, for each possible location, how much it will cost for the current acquisition. In a NUMA system, there are *nm* CPUs distributed over *n* quads, so there is probability *1/nm* that the CPU currently acquiring the lock was also the last CPU to acquire it, as shown in the upper-left entry in the table. In this case, the cost to acquire the lock will just t_f, as shown in the left-most entry of the second row. The weighted latency will be the product of these two quantities, shown in the left-most entry of the third row.

	Same CPU	Different CPU, Same Quad	Different Quad
Probability	*1/nm*	*(m-1)/nm*	*(n-1)/n*
Acq. Latency	t_f	t_m	t_s
Wtd. Latency	t_f/nm	$t_m(m-1)/nm$	$t_s(n-1)/n$

Table 2: Simple Spinlock Access-Type Probabilities and Latencies

Similarly, there will be probability *(m-1)/nm* that one of the *m-1* other CPUs on the same quad as the current CPU last acquired the lock, as shown in the upper-middle entry in the table. In this case, the cost to acquire the lock will be t_m, as shown in the middle entry of the second row. Again, the weighted latency will be the product of these two quantities, as shown in the lower-middle entry of the table.

Finally, there will be probability *(n-1)/n* that one of the CPUs on the other *n-1* quads last acquired the lock, as shown in the upper right entry in the table. In this case, the cost to acquire the lock will be t_s, as shown in the right-hand entry of the middle row. The weighted latency will once again be the product of these two quantities, as shown in the lower right entry of the table.

Under low contention, the overhead of releasing the lock is just the local latency t_f, since there is vanishingly small probability that some other CPU will attempt to acquire the lock while a given CPU holds it. Therefore, the overall NUMA lock-acquisition overhead is obtained by summing the entries in the last row of Table 2 and then adding t_f, as shown in Equation 1.

$$\frac{(n-1)mt_s + (m-1)\ t_m + (nm+1)\ t_f}{nm}$$

Equation 1

An *n*-CPU SMP system can be thought of as a single-quad NUMA system with *n* CPUs per quad. The SMP overhead is therefore obtained by setting *n* to 1, t_m to t_s, and then *m* to *n*, resulting in Equation 2.

$$\frac{(n-1)\ t_s + (n+1)\ t_f}{n}$$

Equation 2

Both of these expressions approach t_s for large *n*, validating the common rule of thumb which states that under low contention, the cost of a spinlock is simply the worst-case memory latency.

Normalizing with $t_s = rt_f$ [8] yields the results shown in Equation 3 and Equation 4.

$$\frac{(n-1)mr + (m-1)\sqrt{r} + (nm+1)}{nm}$$

Equation 3

$$\frac{(n-1)r + (n+1)}{n}$$

Equation 4

7.2 Distributed Reader-Writer Spinlock

Distributed reader-writer spinlock is constructed by maintaining a separate simple spinlock per CPU, and an additional simple spinlock to serialize write-side accesses [9]. Each of these locks is in its own cache line in order to prevent false sharing. However, it is possible to interleave multiple distributed reader-writer spinlocks so that the locks for CPU 0 share one cache line, those for CPU 1 a second cache line, and so on. Table 3 shows an example layout for a four-CPU system.

	Lock							
CPU	A	B	C	D	E	F	E	F
0	0	1	2	3	4	5	6	7
1	8	9	10	11	12	13	14	15
2	16	17	18	19	20	21	22	23
3	24	25	26	27	28	29	30	31
W	32	33	34	35	36	37	38	39

Table 3: Distributed Reader-Writer Spinlock Memory Layout

Each row in the figure represents a cache line, and each cache line is assumed to hold eight simple spinlocks. Each cache line holds simple spinlocks for one CPU, with the exception of the last cache line, which holds the writer-gate spinlocks. If the entire data structure is thought of as a dense array of forty simple spinlocks, then Lock A occupies indices 0, 8, 16, 24, and 32, Lock B occupies 1, 9, 17, 25, 33, and so on.

To read-acquire the distributed reader-writer spin-lock, a CPU acquires its lock. If the write fraction *f* is low, the cost of this acquisition will be roughly t_f. To release a distributed reader-writer spin-lock, a CPU releases its lock. Again, assuming low *f*, the cost of the release will be roughly t_f.

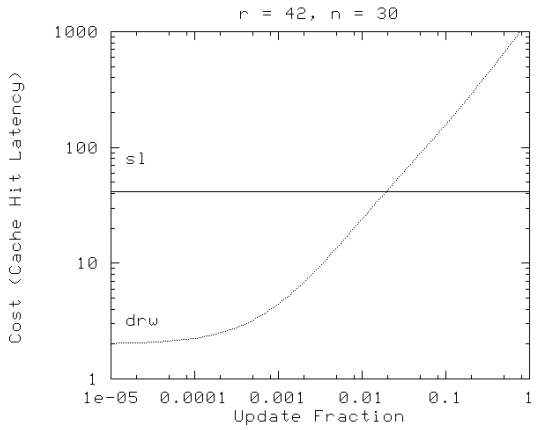

Figure 2: Costs of Simple Spinlock and Distributed Reader-Writer Spinlock

To write-acquire the distributed reader-writer spinlock, a CPU first acquires the writer gate, then each of the CPU's spinlocks in order. If the write fraction *f* is low, the cost of the write-acquisition in this four-CPU example will be roughly $4t_s+t_f$. To release the distributed reader-writer spinlock, a CPU releases the per-CPU locks in order, then the writer gate. Assuming low *f*, the cost of the release will be roughly $5t_f$. See Appendix A for a detailed derivation of more exact results, which are plotted in Figure 2: Costs of Simple Spinlock and Distributed Reader-Writer Spinlock. The labels in this plot are defined in Table 4.

Label	**Description**
drw	Distributed (cache-friendly) reader-writer spinlock [9]
sl	Simple spinlock

Table 4: Trace Labels

More extensive plots of the costs and breakevens for these and other locking primitives are available elsewhere [10].

7.3 Segmented Reader-Writer Spinlock

Although distributed reader-writer lock performs very well when the update fraction is low, it is orders of magnitude slower than simple spinlock when the update fraction approaches 1. Hsieh and Weihl [11] suggest use of a bitmask to track which CPUs have read-acquired the lock, so that the write-acquiring CPU need only reference those CPU's locks. However, this approach requires all read-acquiring CPUs to reference and update a single variable, which could lead to performance degradation due to memory contention.

This section analyzes a different approach, namely, allowing small numbers of CPUs to share read-side locks. Table 5 shows how the lock might be laid out in memory for a four-CPU system in which each read-side lock is shared by a pair of CPUs.

The analysis for segmented locks is very similar to that for distributed reader-writer spinlocks. The breakevens for different segment sizes are plotted in Figure 3.

	Lock							
CPU	A	B	C	D	E	F	E	F
0-1	0	1	2	3	4	5	6	7
2-3	8	9	10	11	12	13	14	15
W	16	17	18	19	20	21	22	23

Table 5: Segmented Reader-Writer Spinlock Memory Layout, Segment Size 2

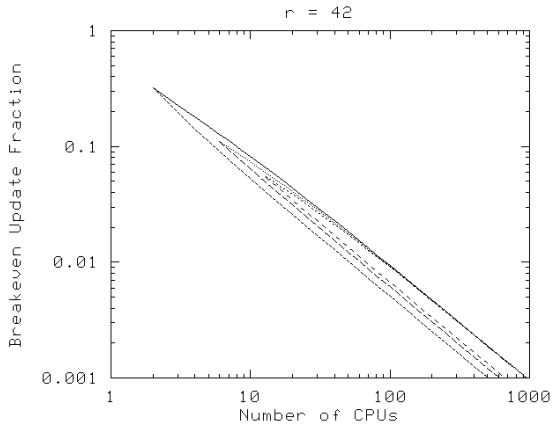

Figure 3: Segmented Reader-Writer Spinlock

The x-axis is the number of CPUs, and the y-axis is the breakeven update fraction *f*, or the fraction of the time that the lock will be write-acquired. When *f=1*, the lock is always write-acquired, and when *f=0*, the lock is always read-acquired.

In the region above the upper line, simple spinlock is optimal. In the region below the lower line, distributed reader-writer spinlock, which is the same as segmented reader-writer spinlock with a segment size of one, is optimal. The second line from the top marks the breakeven between a segment size equal to half the number of CPUs and a segment size equal to one third the number of CPUs, while the second line from the bottom marks the breakeven point between a segment size of 2 and a segment size of 3.

These results show that segmented reader-writer spinlocks can be useful, but only in a narrow range of conditions. The results also support the rule of thumb that says that distributed reader-writer spinlock should be used only if less than *1/mn* of the accesses are write-size accesses (where *mn* is the number of CPUs).

8 Comparison to Measurements

The following two sections compare the predictions of the model to measurements taken on a Sequent NUMA-Q 2000 with eight quads each containing four Intel 180 MHz Pentium Pro Processors. Measurement code runs on the first up to seven quads: the eighth quad runs code that sequences and controls the tests. First, the values of t_s, t_m, and t_f were measured using the on-chip time-stamp counters. The latency of the lock operations was measured using the same methodology and compared to the analytic predictions.

Since the Pentium Pro is a speculative CPU that can execute instructions out of order, special care is required to measure the latencies of single memory references and of single locking primitives. The instruction sequence under test is preceded by a sequence consisting of a serializing instruction (namely CPUID) followed by an RDTSC (read timestamp counter) instruction followed by 40 NOP instructions. The sequence under test is followed by the same sequence in reverse, that is, by 40 NOP instructions followed by an RDTSC instruction and a CPUID instruction. The 40 NOP instructions were observed to be sufficient to force the RDTSC instructions to be executed a predictable amount of time before and after the the instruction sequence under test. A sequence consisting of a CPUID, an RDTSC, a varying number of NOPs, an RDTSC, and a CPUID was used to calibrate the measurements.

8.1 Rules of Thumb

These sections will examine the applicability of some commonly used rules of thumb:

1. The Alpern approximation [8], which assumes that the ratios of the memory latencies of adjacent levels of the memory hierarchy are equal. In other words, Alpern's model assumes that $t_f/t_m=t_m/t_s$.

2. The t_s-only model, where only the t_s term of the full analytic model is used.

3. The naïve model, which simply counts the expected number of remote memory references without considering the probabilities of past histories. This model would for example predict that the write-acquisition overhead of a reader-writer spinlock is nt_s: one remote

reference for the write-side guard lock, and *n-1* references for each of the other CPUs' read-side locks.

The naïve model allows rough performance estimates to be derived early the design process. The following set of remote-reference counting rules have been quite helpful for large real-world scalable parallel programs:

1) Count these as one remote memory reference:

a) Acquisition of a contended spinlock counts.

b) The first reference to a given cacheline of a data structure that is frequently modified counts.

c) The first write to a given cache line of a data structure that is frequently referenced.

2) Count these as *zero* remote memory references:

a) The second read or write to a given cache line of a data structure.

b) References or modifications to auto (on-stack) variables.

c) Releases of locks.

These counts may then be combined an estimate or measurement of remote latency to yield an overall performance estimate.

This approach assumes good locking design, so that locks are not heavily contended. If a given lock is suspected to be a bottleneck, this same procedure may be applied to the code in that lock's critical sections. The result of this procedure is an estimate of the system throughput limit imposed by this lock. Such a per-lock application of this method may be used to determine how aggressive a locking strategy is required.

These rules of thumb have proven themselves to be extremely useful in generating design-time performance estimates. However, they are in no way intended to supplant the simulation, measurement, and analysis methodologies that are used to evaluate the performance of production systems.

8.2 Simple Spinlock

Figure 4 compares the measurements (ticks) to the predictions of the full analytic model, and to the predictions using only the t_s term of the analytic model. Since simple spinlock performance is dominated by remote latency, the two variants of the model are in close agreement with each other. The t_s-only model gives better predictions at lower numbers of quads because the hardware contains optimizations for small numbers of quads that are not captured by the analytic model. Note that the naïve model closely approximates the t_s-only model in this case.

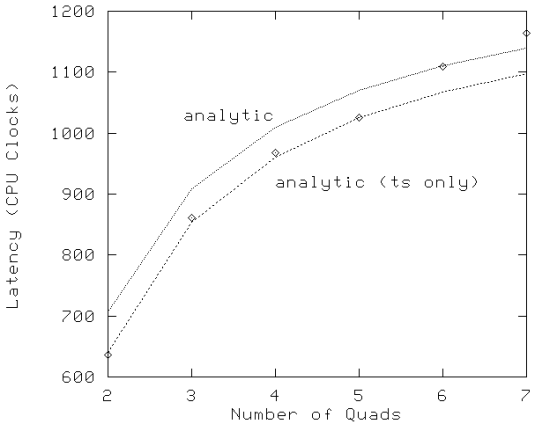

Figure 4: Simple Spinlock

These results validate the rule of thumb that simple spinlock performance can be closely estimated by counting the remote references. Note that there are small but measurable deviations at larger numbers of quads. This is due in part to longer sharing list and in part to speculation. It is possible to design more complex locking primitives that do not exhibit these deviations, but such designs are beyond the scope of this paper.

8.3 Distributed Reader-Writer Spinlock

Figure 5 compares the measurements (ticks) against the predictions of the full analytic model. The full analytic model achieves good agreement with the

measured data. Note that the goodness of the agreement is similar to that of simple spinlock—the log-scale y-axis in Figure 5 makes the differences less apparent.

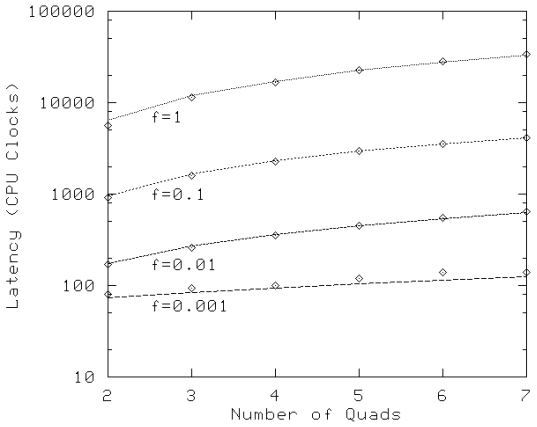

Figure 5: Full Analytic Model, Measured Latencies

Figure 6 compares the measured data to the analytic model using the Alpern approximation. Agreement is excellent for larger update ratios, but there are significant deviations for small update ratios. These deviations are due to the fact that the measured values of t_f for distributed reader-writer spinlock include the pipeline-flushing overhead of locked instructions, which is significant when compared to t_f.

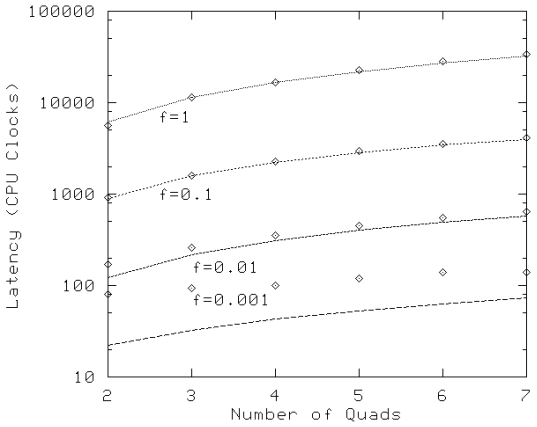

Figure 6: Full Analytic Model, Alpern Approximation

However, the Alpern approximation is reasonably accurate for non-locked instructions on the hardware under test, as well as for the common situation where performance is dominated by t_s.

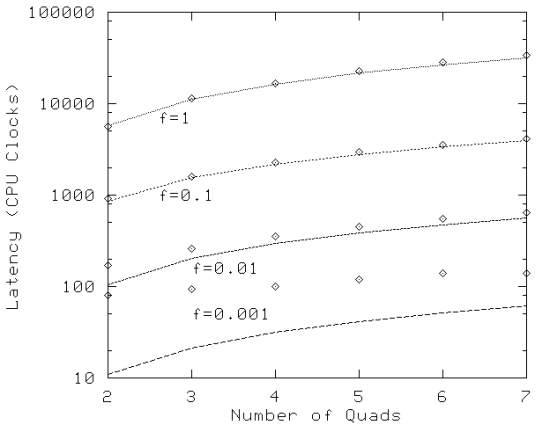

Figure 7: Analytic Model, t_s Term Only

Figure 7 compares the measured data to the t_s-only term of the analytic model. Again, this approximation is quite accurate in the large-*f* regime where remote memory latency dominates, but gives significant error in the small-*f* regime.

Figure 8 compares the measured data to the naïve model. The naïve model is accurate only for large values of *f*.

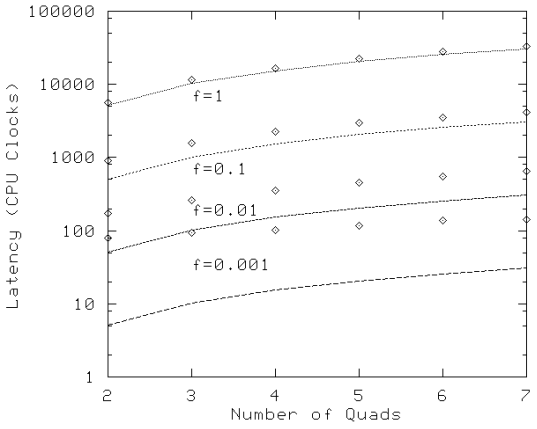

Figure 8: Naïve Model

8.4 Case Study

Sequent's DYNIX/ptx UNIX kernel was ported to NUMA hardware using concurrent development techniques. Many of the NUMA enhancements were implemented and tested well before NUMA hardware was available. Although a NUMA simulator was available for testing functionality, there was insufficient time for use of full performance-level simulation, and there was very

limited time for use of cut-and-try cache-simulation techniques.

Several of the kernel subsystems were therefore adapted to the NUMA environment using the naïve model to guide design decisions. The subsystems include the kernel memory allocator, the timeout subsystem, and the read-copy update mechanism. Although some performance tuning was needed, the naïve model was accurate enough that these tuning changes were quite small.

8.5 Discussion

All of the models give accurate results when remote latency dominates. In these situations, the simplest model (the naïve model) is the model of choice, especially since it is simple enough to be used during design. This simple model has been useful as a rule of thumb for predicting the performance of moderately large parallel programs that do not have highly optimized data sharing. More highly optimized programs, in particular, programs that avoid remote memory references, must use the full analytic model in order to properly account for the local memory references.

Although Alpern's approximation introduces some inaccuracy, it is useful for comparing lock-primitive overhead over a wide span of computer architectures. More accurate comparisons for a particular machine require accurate measurement of t_f as well as t_s and t_m.

In particular, improved remote latencies (t_s) from hard-wired cache-coherence protocol engines will reduce the accuracy of Alpern's approximation, as shown in Figure 9. This plot shows the number of instructions that can be executed in the time required to complete a memory reference on Sequent computers. This line bifurcates in 1996 with the introduction of NUMA systems. These early NUMA systems used a microcoded cache-coherence protocol engine, which resulted in large remote latencies. For these systems, Alpern's approximation was reasonably accurate. However, the introduction of prototype hardwired cache-coherence protocol engines in 2001 resulted in the ratio between remote (t_s) and local (t_m) memory latencies being much smaller than the ratio between local memory (t_m) and on-chip (t_f) latencies, as can be seen from the chart.

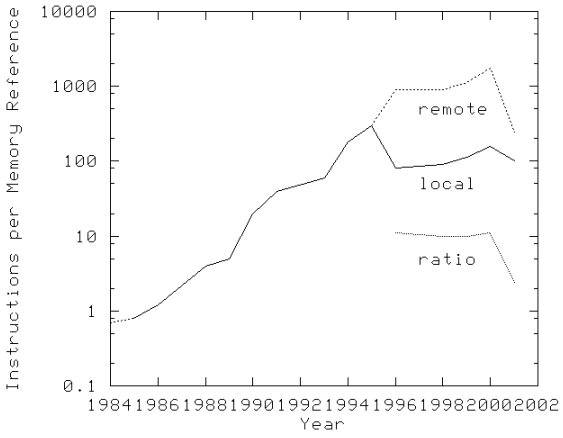

Figure 9: Memory-Latency Trend

Therefore, although Alpern's approximation may still be useful for generic cross-architecture comparisons, it is unlikely to generate accurate results for specific computer systems.

9 Conclusions

This paper presents a performance-evaluation methodology and uses it to compare the costs of several lock primitives. Several variants of the methodology are compared to measured data, with excellent agreement for the full analytic model, and good agreement in situations dominated by remote latency for the other models. These models validate the rule of thumb that the cost of a simple spinlock under low contention may be approximated by t_s.

Breakeven curves for the locking primitives were computed, and the breakeven between simple spinlock and distributed reader-writer spinlock supports the rule of thumb that states that simple spinlock should be used in cases where the update ratio f is greater than the reciprocal of the number of CPUs, $1/nm$.

The full methodology as well as the t_s-only and the naïve simplifications have been used in real-life situations

with good success, for small-scale software exemplified by the locking primitives shown in this paper, as well as for larger-scale software making up a parallel Unix kernel. The t_s-only simplification has proven particularly useful during early design efforts and for obtaining rapid estimates of the performance of large software systems.

10 Acknowledgements

I owe thanks to my training-class students for their insightful questions, and to Dale Goebel for his support of this work.

This work was done with the aid of Macsyma, a large symbolic manipulation program developed at the MIT Laboratory for Computer Science and supported from 1975 to 1983 by the National Aeronautics and Space Administration under grant NSG 1323, by the Office of Naval Research under grant N00014-77-C-0641, by the U. S. Department of Energy under grant ET-78-C-02-4687, and by the U. S. Air Force under grant F49620-79-C-020, between 1982 and 1992 by Symbolics, Inc. of Burlington Mass., and since 1992 by Macsyma, Inc. of Arlington, Mass. Macsyma is a registered trademark of Macsyma, Inc.

11 References

[1] John L. Hennessy and Norman P. Jouppi. Computer technology and architecture: An evolving interaction. *IEEE Computer*, Sept. 1991, page 18-28.

[2] Harold S. Stone and John Cocke. Computer architecture in the 1990s. *IEEE Computer*, Sept. 1991, pages 30-38.

[3] Doug Burger, James R. Goodman, and Alain Kägi. Memory bandwidth limitations of future microprocessors, *ISCA'96*, May 1996, pages 78-89.

[4] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. Continuous profiling: Where have all the cycles gone? In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, New York, NY, October 1997.

[5] Donald E. Knuth. *The Art of Computer Programming, Fundamental Algorithms*. (Reading, MA: Addison-Wesley, . 1973).

[6] Peter S. Magnusson, Anders Landin, and Erik Hagersten. Queue Locks on Cache Coherent Multiprocessors. In *8th International Parallel Processing Symposium (IPPS)*, 1994. (abstract)

[7] T. Lovett and R. Clapp. STiNG: A CC-NUMA computer system for the commercial marketplace. In *Proceedings of the 23rd International Symposium on Computer Architecture*, May 1996, pages 308-317.

[8] Bowen Alpern, Larry Carter, Ephraim Feig, and Ted Selker. The uniform memory hierarchy model of computation. *Foundations of Computer Science Conference Proceedings*, 1990

[9] Paul E. McKenney. Selecting locking primitives for parallel programs, *Communications of the ACM*, 39(10) Oct. 1996.

[10] Paul E. McKenney. Read-copy update: using execution history to solve concurrency problems, *Parallel and Distributed Computing and Systems*, Oct. 1998.

[11] Wilson C. Hsieh and William E. Weihl: Scalable reader-writer locks for parallel systems. *MIT Laboratory for Computer Science technical report MIT/LCS/TR-521*, Cambridge, MA. 1991.

	Quad 0				Quad 1				...	Quad n			
CPU	0	1	2	3	*m*	*m+1*	*m+2*	*m+3*	...	*nm-3*	*nm-2*	*nm-1*	*nm*
Read	*1-f*	*1-f*	*1-f*	*1-f*	*1-f*	*1-f*	*1-f*	*1-f*	...	*1-f*	*1-f*	*1-f*	*1-f*
Write	*f*	*f*	*f*	*f*	*f*	*f*	*f*	*f*	...	*f*	*f*	*f*	*f*
Cost	t_f	t_m	t_m	t_m	t_s	t_s	t_s	t_s	...	t_s	t_s	t_s	t_s

Table 6: Unnormalized Probability Matrix for Distributed Reader-Writer Spinlock

A Derivation for Distributed Reader-Writer Spinlock

Computing costs for large *f* is a bit more involved, since a write-side lock will force *all* CPU's locks into the write-locking CPU's cache. Assuming independent interarrival distributions, the probability of a CPU's lock being in its cache is the probability that this CPU did either a read- or write-side acquisition since the last write-side acquisition by any of the other *(n-1)* CPUs. Similarly, the probability of some other CPU's lock being in a given CPU's cache is the probability that the given CPU did a write-side acquisition since both: (1) the last read-side acquisition by the CPU corresponding to the lock and (2) the last write-side acquisition by one of the *(n-1)* remaining CPUs. These probabilities may be more easily derived by referring to Table 6, which shows the relative frequency and cost of the read- and write-acquisition operations.

It is important to note that the only events that can affect a given per-CPU lock are read-acquisitions by that CPU and write-acquisitions by all CPUs. These events have a total weighting of *1+(nm-1)f.* This important quantity will be found in the denominator of many of the subsequent equations.

A.1 Read Acquisition and Release

Suppose CPU 0 is read-acquiring the lock. As noted earlier, the only events that can affect the cost are CPU 0's past read-acquisitions and all CPUs' write-acquisitions, for a total weighting of *1+(nm-1)f.* Of this, only read- and write-acquisitions, with combined weight of *(1-f+f)=1*, will leave CPU 0's element of the distributed reader-writer spinlock in CPU 0's cache.

Therefore, the cost of CPU 0's read operation has probability $1/(1+(nm-1)f)$ of being t_f.

Similarly, there is probability $(m-1)f/(1+(nm-1)f)$ that the last operation was a write-acquisition by another CPU on Quad 0, in which case the cost will be t_m.

Finally, there is probability $(nm-m)f/(1+(nm-1)f)$ that the last operation was a write-acquisition by one of the CPUs on the *n-1* other quads, in which case the cost will be t_s.

Weighting these costs by their probability of occurrence gives the expected cost of a read acquisition shown in Equation 5.

$$\frac{(nm-m)\,f\quad t_s+(m-1)\,f\quad t_m+t_f}{1+(nm-1)\,f} \qquad \textbf{Equation 5}$$

Read release will impose an additional cost of t_f, as shown in Equation 6.

$$\frac{(nm-m)\,f\quad t_s+(m-1)\,f\quad t_m+\left(2+(nm-1)\,f\right)\,t_f}{1+(nm-1)\,f}$$

Equation 6

A.2 Write Acquisition

Suppose CPU 0 is write-acquiring the lock. It must first acquire the writer gate, the cost of which was derived in Section 7. It must then acquire each of the per-CPU locks. There are three cases to consider:

1. The lock for the write-acquiring CPU (this cost was derived in Section A.1, Equation 5).

2. The locks for other CPUs on the write-acquiring CPU's quad.

3. The locks for CPUs on other quads.

Expressions for these last two are derived in the following sections.

A.2.1 Different CPU, Same Quad

If the write-acquiring CPU last write-acquired the lock, the cost will be t_f. If some other CPU, including the owning CPU, last write-acquired the lock, the cost will be t_m. If the owning CPU last read-acquired the lock, the cost will also be t_m. Finally, if a CPU on some other quad last write-acquired the lock, the cost will be t_s.

Referring again to Table 6, the probability that the write-acquiring CPU last write-acquired the lock is just *f/(1+(nm-1)f)*. The probability that the owning CPU last read- or write-acquired the lock is *1/(1+(nm-1)f)*, and the probability that another CPU on the same quad last write-acquired the lock is *(m-2)f/(1+(nm-1)f)*, assuming *m>2*. Finally, the probability that a CPU on some other quad last write-acquired the lock is *(nm-m)f/(1+(nm-1)f.*

Weighting the costs by their respective probabilities of occurrence gives the expected cost of acquiring the per-CPU locks for the CPUs on the same quad as the write-acquiring CPU, as shown in Equation 7.

$$\frac{(n-1)mf \quad t_s + (1+(m-2)f) \quad t_m + f \quad t_f}{1+(nm-1)f}$$

Equation 7

A.2.2 Different Quad

If the write-acquiring CPU last write-acquired the lock, the cost will be t_f. If some other CPU on the same quad last write-acquired the lock, the cost will be t_m. Finally, if a CPU on some other quad last write-acquired the lock, or if the owning CPU last read-acquired the lock, the cost will be t_s.

Referring again to Table 6, the probability that the write-acquiring CPU last write-acquired the lock is just *f/(1+(nm-1)f)*. The probability that another CPU on the same quad last write-acquired the lock is *(m-1)f/(1+(nm-1)f)*, assuming *m>2*. The probability that the owning CPU last read- or write-acquired the lock is *1/(1+(nm-1)f).* Finally, the probability that some other CPU on some other quad last write-acquired the lock is *(nm-m-1)f/(1+(nm-1)f.*

Weighting the costs by their respective probabilities of occurrence gives the expected cost of acquiring the per-CPU locks for the CPUs on the same quad as the write-acquiring CPU, as shown in Equation 8.

$$\frac{(1+(nm-m-1)f) \quad t_s + (m-1)f \quad t_m + f \quad t_f}{1+(nm-1)f}$$

Equation 8

A.2.3 Overall Write Acquisition and Release Overhead

The overall write-acquisition and release overhead is the overhead of a simple spinlock (Equation 1), plus the overhead of acquiring the per-CPU lock owned by this CPU (Equation 5), plus the overhead of acquiring the per-CPU locks owned by the other CPUs on the same quad (m-1 times Equation 7), plus the overhead of acquiring per-CPU locks owned by the CPUs on other quads (nm-m times Equation 8), plus the additional overhead of releasing the per-CPU locks (nmtf). Combining these equations and simplifying yields Equation 9.

$$\frac{\left[\begin{array}{c}(n^2m^2+(1-m)nm-m) \quad t_s + \\ ((m-1)nm+m-1) \quad t_m + \\ (n^2m^2+2nm+1) \quad t_f\end{array}\right]}{nm}$$

Equation 9

A.3 Overall Overhead

The overall overhead is 1-f times the overall read overhead (Equation 6) plus f times the overall write overhead (Equation 9), as shown in Equation 10.

$$\frac{\left[\begin{array}{l}\left[\begin{array}{c}\left[\begin{array}{c}n^3m^3-(m+1)n^2m^2+\\(m-1)nm+m\end{array}\right]f^2+\\\left(2n^2m^2-(2m-1)nm-m\right)f\end{array}\right]t_s+\\\left[\begin{array}{c}\left[\begin{array}{c}(m-1)n^2m^2-\\(m-1)nm-m+1\end{array}\right]f^2+\\\left(2(m-1)nm+m-1\right)f\end{array}\right]t_m+\\\left[\begin{array}{c}\left(n^3m^3-1\right)f^2+\\\left(2n^2m^2-nm+1\right)f+2nm\end{array}\right]t_f\end{array}\right]}{(nm-1)nmf+nm}$$

Equation 10

An *n*-CPU SMP system can be thought of as a single-quad NUMA system with *n* CPUs per quad. The SMP overhead is therefore obtained by setting *n* to 1, t_m to t_s, and then *m* to *n*, resulting in Equation 11.

$$\frac{\left[\begin{array}{c}\left(\left(n^3-2n^2+1\right)f^2+\left(2n^2-n-1\right)f\right)t_s+\\\left(\left(n^3-1\right)f^2+\left(2n^2-n+1\right)f+2n\right)t_f\end{array}\right]}{\left(n^2-n\right)f+n}$$

Equation 11

This expression approaches nft_s for large *n* and large memory-latency ratios, validating the rule of thumb often used for distributed reader-writer spinlock.

Normalizing with $t_s=rt_f$, $t_m=\ddot{O}r$, and $tf=1$ yields the results shown in Equation 12 and Equation 13.

$$\frac{\left[\begin{array}{l}\left[\begin{array}{c}\left[\begin{array}{c}n^3m^3-(m+1)n^2m^2+\\(m-1)nm+m\end{array}\right]f^2+\\\left(2n^2m^2-(2m-1)nm-m\right)f\end{array}\right]r+\\\left[\begin{array}{c}\left[\begin{array}{c}(m-1)n^2m^2-\\(m-1)nm-m+1\end{array}\right]f^2+\\\left(2(m-1)nm+m-1\right)f\end{array}\right]\sqrt{r}+\\\left[\begin{array}{c}\left(n^3m^3-1\right)f^2+\\\left(2n^2m^2-nm+1\right)f+2nm\end{array}\right]\end{array}\right]}{(nm-1)nmf+nm}$$

Equation 12

$$\frac{\left[\begin{array}{c}\left(\left(n^3-2n^2+1\right)f^2+\left(2n^2-n-1\right)f\right)r+\\\left(\left(n^3-1\right)f^2+\left(2n^2-n+1\right)f+2n\right)\end{array}\right]}{\left(n^2-n\right)f+n}$$

Equation 13