Performance of Locking Primitives at Low Levels of Contention

Paul E. McKenney
Sequent Computer Systems, Inc.
July 25, 1998

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 and rules of thumb are compared to measurements taken on real hardware, thereby showing where each may be safely used.

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 [Hennessy91, Stone91, Burger96]. 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.

However, 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 demonstrates its use on a number of locking primitives. This technique has been used successfully on moderately large real-world programs. This technique is similar to that used by Magnusson [Magnusson94], but unlike Magnusson's, does not require detailed analysis of assembly code. Magnusson's approach is nevertheless the technique of choice in situations requiring exact analysis of existing short sequences of code. Hardware measurement techniques are available when it is acceptable to focus the analysis on existing code running on a particular machine [Anderson97]. 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.

Section 3 introduces the technique for estimating performance of algorithms running on symmetric multiprocessor (SMP) and CC-NUMA architectures. Section 4 presents an analysis of a simple spinlock, Section 5 presents an analysis of a centralized reader-writer spinlock, Section 6 presents an analysis of a distributed reader-writer spinlock, and Sections 7 and 9 summarize the results for these primitives and for read-copy update [McKenney98b]. Section 10 presents results, Section Error: Reference source not found presents related work, and Section 11 presents summary and conclusions.

3 CC-NUMA Memory-Latency Model

Memory latency is the dominating factor in low-contention lock-overhead analysis due to the high cost of memory accesses compared to instruction execution overhead. Caches can cause memory latency to vary dynamically depending on the order of accesses and updates.

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 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 L2 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_s, the latter quantity 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 [Magnusson94], however, this will usually require that the program be coded and compiled to assembly language, and is often infeasible for large programs.

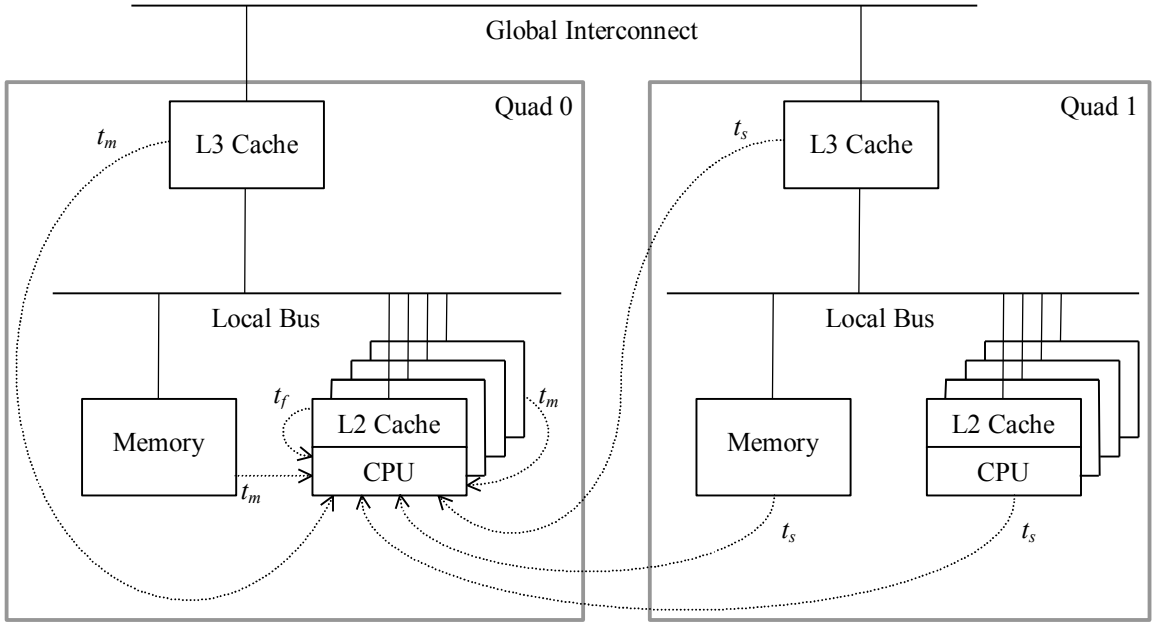

Figure 1: CC-NUMA Memory Latency

Once a given data item has been accessed by a CPU, it is cached in that CPU's L2 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 L3 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 either that CPU's L2 cache (with access latency t_f) or in its L1 cache (with access latency approximated by zero). In other words, at low contention, we assume that there is insufficient cache pressure to force data out of a CPU's L2 cache. However, data is assumed to remain in the much smaller L1 caches only if that data was accessed very recently, for example, during the execution of a single simple function.

3.1 Assumptions

A CC-NUMA system contains *n* quads and *m* CPUs per quad (two and four, respectively, in the example shown in the figure). To simplify analysis, each quad is assumed to contain the same number of CPUs. In addition, the analysis makes the following assumptions:

1) 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 probability of one CPU attempting to acquire a lock held by another CPU.

2) 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. Updates and read-only accesses are assumed to occur randomly with probability *f* and *1-f*, respectively

3) 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 small

number of "heavyweight" instructions (such as atomic read-modify-write instructions) by adding an additional t_l for these heavyweight instructions.

4) 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 "L1 cache". Instruction fetches and stack references (function calls and returns, accesses to auto variables) are assumed to hit this L1 cache.

5) Cache pressure is assumed low, so that a variable that resides in a given cache remains there until it is invalidated by a write from some other CPU.

6) CC-NUMA systems with multiple levels of off-chip cache are assumed to have geometric memory latencies so that $t_s/t_m=t_m/t_f$. This assumption is generally very nearly true in practice for non-atomic memory accesses, and greatly simplifies the equations. However, substitution of r for t_f, t_m, and t_s is done as a last step so that the equations may be easily adapted to any particular real-world situation.

3.2 Summary of Nomenclature

Table 1 summarizes the symbols used in the derivations.

	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 L2 cache.

Table 1: Nomenclature for Lock Cost Derivation

3.3 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.

3.4 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, but 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 back-of-the envelope analysis that can often be applied to large systems during early design.

Note that because the actual behavior is depends critically on cache state, actual results can deviate significantly from the analytic results presented in this paper. For example, if the L2 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 this model have proven quite useful in practice.

4 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.

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

Table 2: Simple Spinlock Access-Type Probabilities and Latencies

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.

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.

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

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.

$$\frac{(n-1)t_s + (n+1)t_f}{n} \qquad \textbf{Equation 2}$$

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

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

$$\frac{(n-1)mr + (m-1)\sqrt{r} + (nm+1)}{nm} \qquad \textbf{Equation 3}$$

$$\frac{(n-1)r+(n+1)}{n}$$ **Equation 4**

5 Centralized Reader-Writer Spinlock

A centralized reader-writer spinlock may be constructed by guarding three counters with a simple spinlock. The first counter counts the number of requests, the second counter counts the number of grants, and third counter counts the number of outstanding read-side locks. All the counters and the spinlock share a single cache line.

Pseudo-code for the read-side acquisition is as shown in Figure 2. Alternative algorithms that are optimized for high contention are well known [Magnusson94, Scott92].

1) Acquire the simple spinlock.
2) Increment the count of requests, retaining the initial value in a local variable.
3) Increment the count of outstanding readers, thereby blocking future writers.
4) If the initial value from step 2 is equal to the count of grants, we hold the lock:
   a) Increment the count of grants, thereby allowing subsequent readers to proceed.
   b) Release the simple spinlock.
   c) Return to the caller (do not perform the following steps).
5) Otherwise, we must spin:
   a) Release the simple spinlock.
   b) Busy-wait until the count of grants is equal to the result of the increment in step 2
   c) Acquire the simple spinlock.
   d) Repeat from step 4

Figure 2: Centralized Reader-Writer Spinlock Read-Side Acquisition

Under low contention, the lock will almost always be available, and we will almost never reach step 5. The cost of this code fragment will be that of simple spinlock (less t_f, since the lock will be in the L1 cache at release), since all variables share a cache line, which, after step 1, will reside in the CPU's L1 cache.

Pseudo-code for read-side release is as shown in Figure 3.

1) Acquire the simple spinlock.
2) Decrement the count of outstanding readers, thereby allowing future writers to proceed.
3) Release the simple spinlock.

Figure 3: Centralized Reader-Writer Spinlock Read-Side Release

Under low contention, the cache line containing the spinlock and counters would at worst reside in the CPU's L2 cache. The lock acquisition would bring the cache line into the CPUs L1 cache, so that the total cost of the release would be t_f.

Pseudo-code for the write-side acquisition is as shown in Figure 4.

1) Acquire the simple spinlock.
2) Increment the count of requests, retaining the initial value in a local variable.
3) If the initial value from step 2 is equal to the count of grants, and if the count of outstanding readers is zero, we hold the lock:
   a) Release the simple spinlock, but do *not* increment the count of grants, thereby blocking readers.
   b) Return to the caller (do not perform the following steps).
4) Otherwise, we must spin:
   a) Release the simple spinlock.
   b) Busy-wait until the count of requests is equal to the result of the increment in step 2 and the count of outstanding readers is zero.
   c) Acquire the simple spinlock.
   d) Repeat from step 4

Figure 4: Centralized Reader-Writer Spinlock Write-Side Acquisition

Again, under low contention, the lock will almost always be available, and we will almost never reach step 4. The cost of this code fragment will be that of simple spinlock, (less *tf*, since the lock will be in the L1 cache at release), since all variables share a cache line, which, after step 1, will reside in the CPU's L1 cache.

Pseudo-code for the write-side release is as shown in Figure 5.

1) Acquire the simple spinlock.
2) Increment the number of grants, thereby allowing either read or write requests to proceed.
3) Release the simple spinlock.

Figure 5: Centralized Reader-Writer Spinlock Write-Side Release

And once again, under low contention, the cache line containing the spinlock and counters would at worst reside in the CPU's L2 cache. The lock acquisition would bring the cache line into the CPUs L1 cache, so that the total cost of the release would be t_f.

The cost of centralized reader-writer spinlock is therefore almost exactly that of simple spinlock under light contention. However, centralized reader-writer spinlock does have significant advantage in situations with high read-side contention.

6 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 [McKenney96]. 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. An example layout for a four-CPU system is shown in Figure 6.

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

Figure 6: 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 spinlock, a CPU simply 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 spinlock, a CPU simply releases its lock. Again, assuming low *f*, the cost of the release will be roughly t_f.

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 $5t_f$.

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 , 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.

	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 3: Unnormalized Probability Matrix for Distributed Reader-Writer Spinlock

6.1 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 costing t_f.

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

And 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.

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

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

$$\frac{(nm-m)f\,t_s + (m-1)f\,t_m + (2+(nm-1)f)\,t_f}{1+(nm-1)f} \qquad \textbf{Equation 6}$$

6.2 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 4. 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 , 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.

6.2.1 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 , 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.

$$\frac{(n-1)mf\,t_s + (1+(m-2)f)t_m + f\,t_f}{1+(nm-1)f}$$ **Equation 7**

6.2.2 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 , 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.

$$\frac{(1+(nm-m-1)f)t_s + (m-1)f\,t_m + f\,t_f}{1+(nm-1)f}$$ **Equation 8**

6.2.3 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 (nmt_f). Combining these equations and simplifying yields Equation 9.

$$\frac{\left[\begin{array}{c}\left(n^2m^2+(1-m)nm-m\right)t_s + \\ \left((m-1)nm+m-1\right)t_m + \\ \left(n^2m^2+2nm+1\right)t_f\end{array}\right]}{nm}$$ **Equation 9**

6.3 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.

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

$$\frac{\left[\begin{array}{c}\left(\left(n^3-2n^2+1\right)f^2+\left(2n^2-n-1\right)f\right)t_s+\\ \left(\left(n^3-1\right)f^2+\left(2n^2-n+1\right)f+2n\right)t_f\end{array}\right]}{\left(n^2-n\right)f+n} \qquad \textbf{Equation 11}$$

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

$$\frac{\left[\begin{array}{c}\left[\begin{array}{c}\left(n^3m^3-\left(m-1\right)n^2m^2+\left(m-1\right)nm+m\right)f^2\\ \left(2n^2m^2-\left(2m-1\right)nm-m\right)f\end{array}\right]r+\\ \left(\left(\left(m-1\right)n^2m^2-\left(m-1\right)nm-m+1\right)f^2+\left(2\left(m-1\right)nm+m-1\right)f\right)\sqrt{r}+\\ \left(\left(n^3m^3-1\right)f^2+\left(2n^2m^2-nm+1\right)f+2nm\right)\end{array}\right]}{\left(nm-1\right)nmf+nm} \qquad \textbf{Equation 12}$$

$$\frac{\left(\left(n^3-2n^2+1\right)f^2+\left(2n^2-n-1\right)f\right)r+\left(\left(n^3-1\right)f^2+\left(2n^2-n+1\right)f+2n\right)}{\left(n^2-n\right)f+n} \qquad \textbf{Equation 13}$$

7 Read-Copy Update

Read-copy update is an update discipline used for read-intensive data structures that allows read-side access with no synchronization operations whatsoever [Slingwine95, McKenney98a]. It may be used to implement a deferred-free operation posited by several researchers [Kung80, Manber84, Pugh90] or to eliminate reliance on a garbage collector for the deferred-free operation [Lea97]. Analytic expressions for read-copy update's performance have been derived elsewhere [McKenney98b], and are presented in the following section.

8 NUMA Performance Summary

Simple spinlock and centralized reader-writer spinlock:

$$\frac{\left(n-1\right)mr+\left(m-1\right)\sqrt{r}+\left(nm+1\right)}{nm} \qquad \textbf{Equation 14}$$

Distributed reader-writer spinlock:

$$\frac{\left[\begin{array}{c}\left[\begin{array}{c}\left(n^3m^3-\left(m-1\right)n^2m^2+\left(m-1\right)nm+m\right)f^2\\ \left(2n^2m^2-\left(2m-1\right)nm-m\right)f\end{array}\right]r+\\ \left(\left(\left(m-1\right)n^2m^2-\left(m-1\right)nm-m+1\right)f^2+\left(2\left(m-1\right)nm+m-1\right)f\right)\sqrt{r}+\\ \left(\left(n^3m^3-1\right)f^2+\left(2n^2m^2-nm+1\right)f+2nm\right)\end{array}\right]}{\left(nm-1\right)nmf+nm} \qquad \textbf{Equation 15}$$

Best-case read-copy lock cost is simply 7 L2 cache hits.

Worst-case read-copy lock:

$$\left(3n+2nm-m+3\right)r+\left(2nm+m-1\right)\sqrt{r}+\left(3nm^2+7nm+8\right)+nm^2t_c \qquad \textbf{Equation 16}$$

Typical read-copy lock with read-copy update arrival rate of λ arrivals per read-copy latency period:

$$\frac{1}{e^{\lambda}-1}\sum_{k=1}^{\infty}\frac{\lambda^{k}\left[\begin{array}{l}\left(3n+5nm-m\right)r-3nm\left(1-\frac{1}{nm}\right)^{k}r+\\ \left(2nm+m-1\right)\sqrt{r}+\left(10nm+7k+1\right)+nmt_{c}\end{array}\right]}{k!k}$$

Equation 17

In both read-copy results, t_c is the latency to access the real-time clock. This latency will vary depending on the hardware platform.

9 SMP Performance Summary

Simple spinlock and centralized reader-writer spinlock:

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

Equation 18

Distributed reader-writer spinlock:

$$\frac{\left(\left(n^{3}-2n^{2}+1\right)f^{2}+\left(2n^{2}-n-1\right)f\right)r+\left(\left(n^{3}-1\right)f^{2}+\left(2n^{2}-n+1\right)f+2n\right)}{\left(n^{2}-n\right)f+n}$$

Equation 19

Best-case read-copy lock cost is simply 7 L2 cache hits.

Worst-case read-copy lock:

$$\left(4n+5\right)r+\left(3n^{2}+7n+8\right)+n^{2}t_{c}$$

Equation 20

Typical read-copy lock with read-copy update arrival rate of λ arrivals per read-copy latency period:

$$\frac{1}{e^{\lambda}-1}\sum_{k=1}^{\infty}\frac{\lambda^{k}\left[\left(7n+2\right)r-3n\left(1-\frac{1}{nm}\right)^{k}r+\left(10n+7k+1\right)+nt_{c}\right]}{k!k}$$

Equation 21

In both read-copy results, t_c is the latency to access the real-time clock. This latency will vary depending on the hardware platform.

10 Results

Section 10.1 presents graphs of the costs of the various locking primitives under low-latency conditions, and Section 10.2 presents graphs of the breakeven points showing under which conditions the various primitives are optimal.

10.1 Costs

The traces are labeled as follows:

Label	Description
drw	Distributed (cache-friendly) reader-writer spinlock [McKenney96]
qrw	Per-quad distributed reader-writer spinlock
sl	Simple spinlock
rcb	Best-case read-copy update (infinite degree of batching)
rcp	Poisson read-copy update arrivals with $\lambda=10$ updates per CPU per quiescent period
rcz	Poisson read-copy update arrivals with $\lambda=1$ updates per CPU per quiescent period
rcn	Poisson read-copy update arrivals with $\lambda=0.1$ updates per CPU per quiescent period
rcw	Worst-case read-copy update (isolated update)

Table 4: Trace Labels

The first set of figures display read-copy update overhead as a function of the number of CPUs (see Table 4 for definitions of the trace labels). At these typical latency ratios and moderate-to-high update fractions, read-copy update outperforms the other locking primitives. Note particularly that the overhead of the non-worst-case read-copy overheads do not increase with increasing numbers of CPUs. This excellent scaling behavior is due to the batching capability of read-copy update. Although simple spinlock and centralized reader-writer spinlock also show good scaling, this good behavior is restricted to low-contention conditions. The poor behavior of these primitives under conditions of high contention is well known. Use of read-copy update is quite advantageous in systems with large numbers of CPUs.

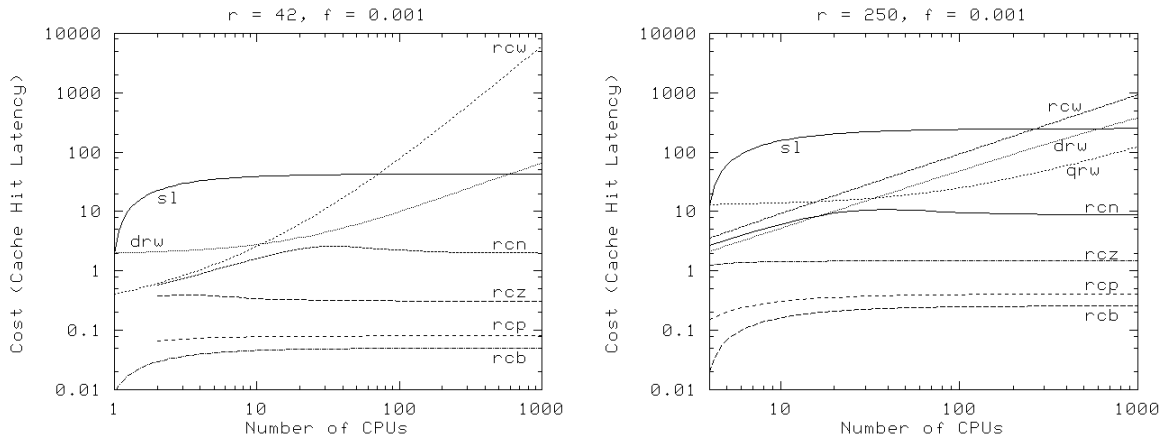

Figure 7: Read-Copy Overhead as Function of Number of CPUs

The next set of figures shows read-copy overhead as a function of the update fraction *f*. As expected, read-copy update performs best when the update fraction is low. Update fractions as low as 10^{-10} are not uncommon [McKenney98a].

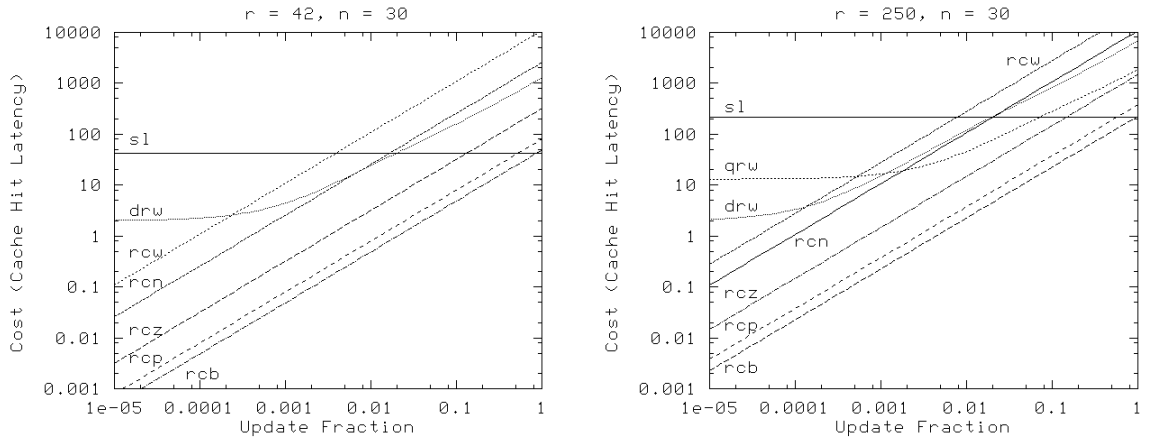

Figure 8: Read-Copy Overhead as Function of Update Fraction

Figure 9 shows read-copy overhead as a function of the memory-latency ratio *r*. The distributed reader-writer primitives have some performance benefit at high latency ratios, but this performance benefit is offset in many cases by high contention, by larger numbers of CPUs, or by lower update fractions, as shown in Figure 10.

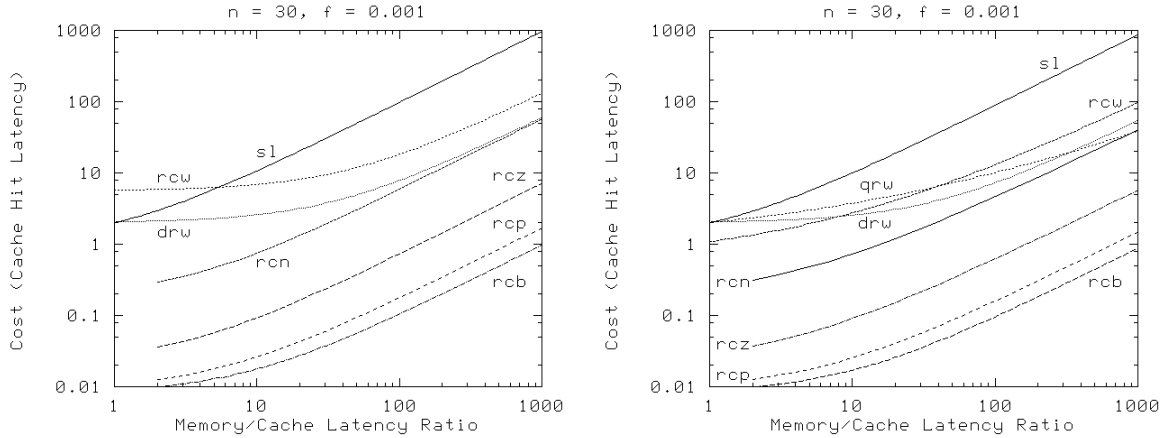

Figure 9: Read-Copy Overhead as Function of Memory-Latency Ratio

The situation shown in Figure 10 is far from extreme. As noted earlier, common situations can result in update fractions below 10^{-10}.

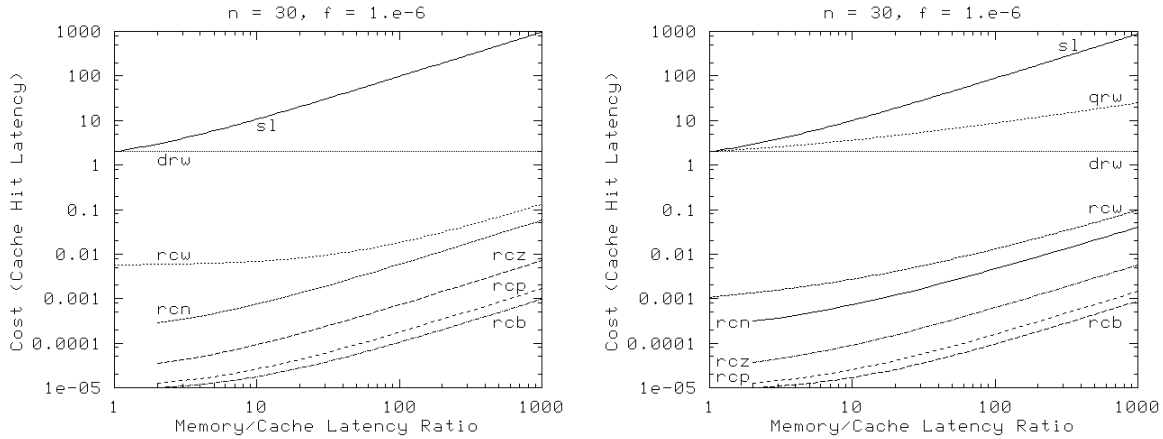

Figure 10: Read-Copy Overhead as Function of Memory-Latency Ratio for Lower Value of *f*

Note finally that all of these costs assume that the update-side processing for read-copy update is guarded by a simple spinlock. In cases where the update-side processing may use a more aggressive locking design (for example, if only one CPU, process, or thread alters the data), read-copy update will have an even greater performance advantage.

10.2 Breakevens

This section presents graphs that depict the regions where simple spinlock (labelled "A"), reader-writer spinlock (labelled "B"), and read-copy update (labelled "C") are optimal. The graphs on the left-hand side are for a Sequent Symmetry SMP machine, while those on the right are for a Sequent NUMA-Q CC-NUMA machine.

The boundary between regions "A" and "B" in the left-hand graph in Figure 11 was obtained by setting Equation 18 equal to Equation 19 and solving for *f*. The boundary between regions "B" and "C" was obtained by setting Equation 19 equal to Equation 20 and solving for *f*. The boundaries in the right-hand graph were obtained in a similar manner from Equation 14, Equation 15, and Equation 16.

These graphs show three well-defined regions in which each of simple spinlock, distributed reader-writer spinlock, and worst-case read-copy update are optimal. As expected, read-copy update is always optimal for sufficiently small values of *f*. These graphs also give evidence to support the rule of thumb that distributed reader-writer spinlock should be favored over simple spinlocks when the number of CPUs is less than *1/f*.

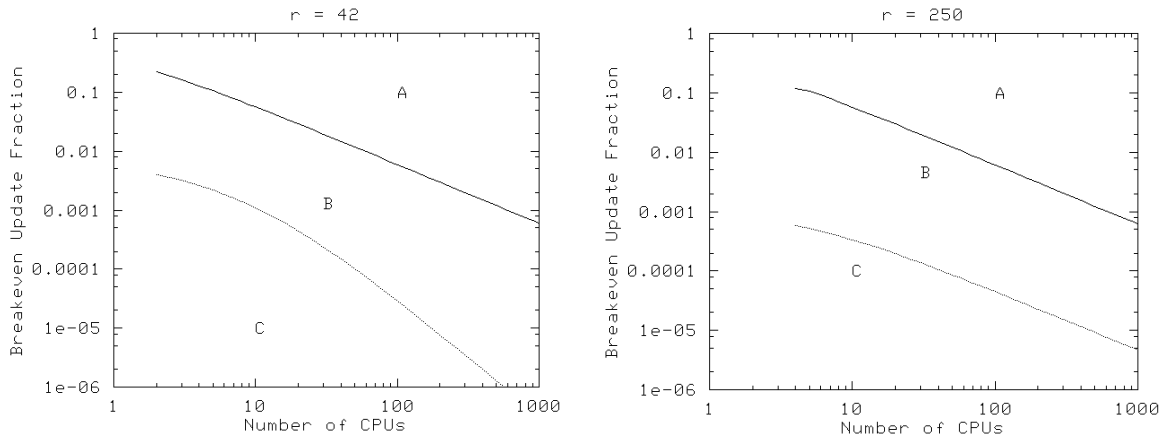

Figure 11: Breakevens for Worst-Case Read-Copy Update

The pair of graphs in Figure 12 are derived in a similar manner, however, the solution must be obtained numerically rather than analytically. The graph on the left-hand side was derived from Equation 14, Equation 15, and Equation 17 with *r* set to 42 and λ set to 0.1, as appropriate for a Sequent Symmetry. The graph on the right-hand side was derived from Equation 18, Equation 19, and Equation 21 with *r* set to 250 and λ set to 0.1, as appropriate for a Sequent NUMA-Q.

Again, these graphs show three well-defined regions in which each of simple spinlock, distributed reader-writer spinlock, and $\lambda=0.1$ read-copy update are optimal. Note that the three breakeven curves intersect at a single point, as required, given that each locking primitive has a region in which it is optimal.

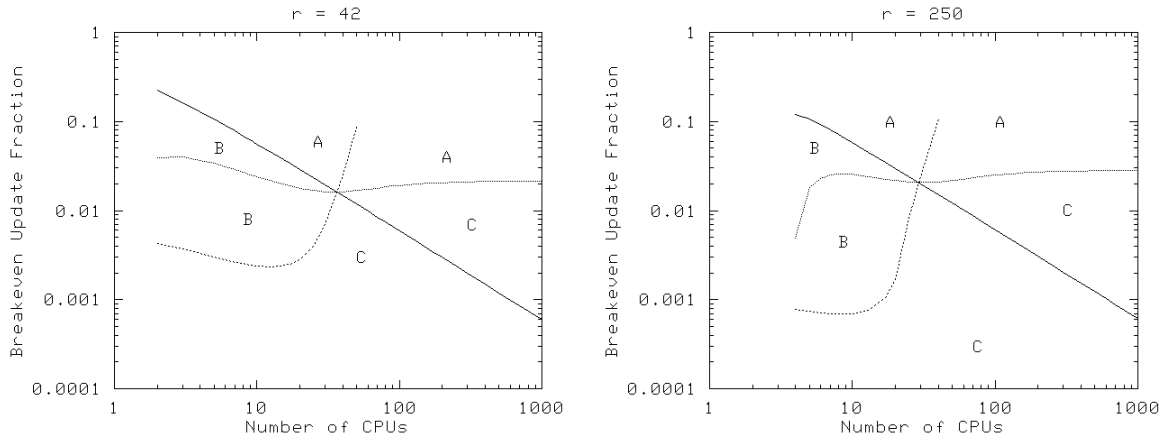

Figure 12: Breakevens for Read-Copy Update With λ=0.1

The pair of graphs in Figure 13 are obtained in the same way as the pair in Figure 12, but with λ set to 1 rather than 0.1. This higher update intensity allows read-copy update to enjoy greater batching and thus higher performance. There are still three well-defined regions where each locking primitive is optimal.

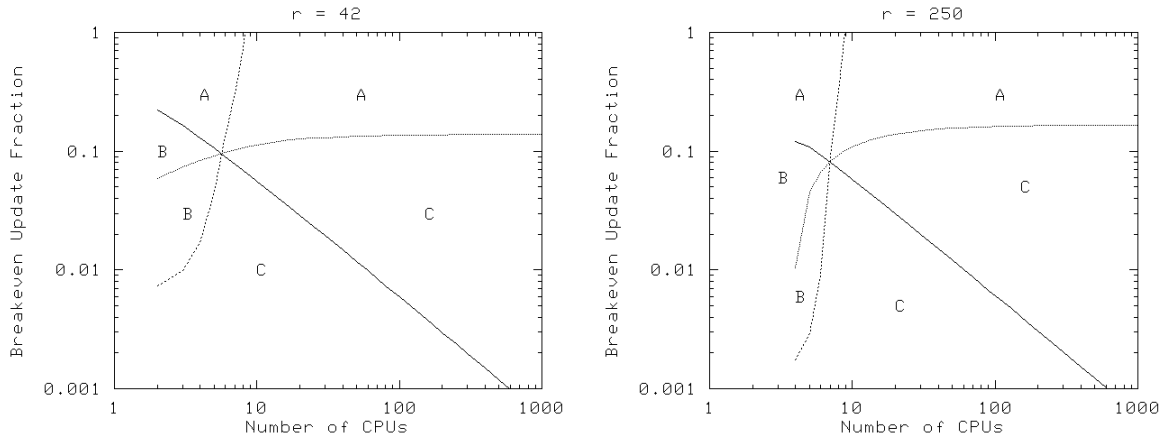

Figure 13: Breakevens for Read-Copy Update With λ=1

The pair of graphs in Figure 14 set λ to 10 rather than 1, so that read-copy update gains even greater benefit from batching. In the left-hand figure, there is no longer a region where distributed reader-writer lock is optimal, and in the right-hand figure, distributed reader-writer spinlock's region of optimality is very small.

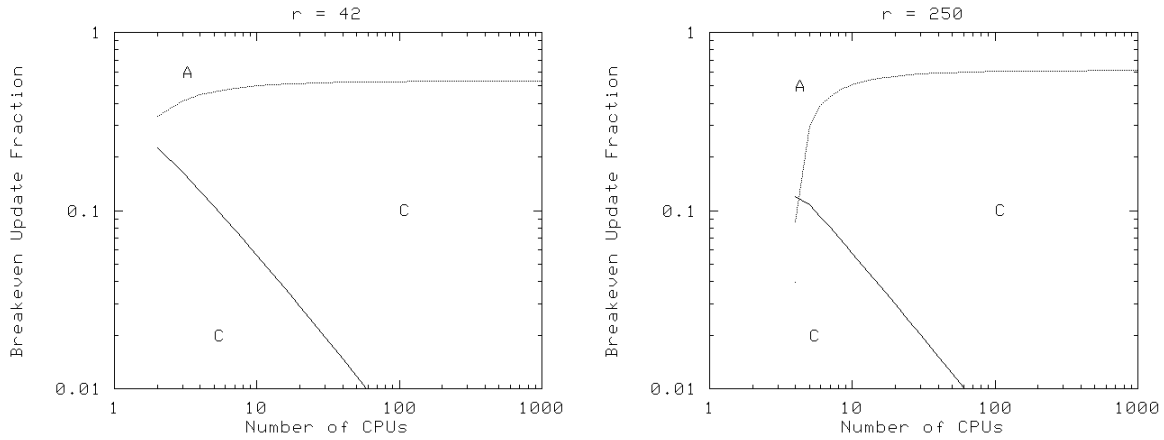

Figure 14: Breakevens for Read-Copy Update With λ=10

The graphs shown in Figure 15 compare simple spinlock to best-case read-copy update. Again, there is no region where distributed reader-writer spinlock is optimal. Note that these graphs use a linear y-axis so that the breakeven curve may be more easily distinguished from the upper border of the graphs.

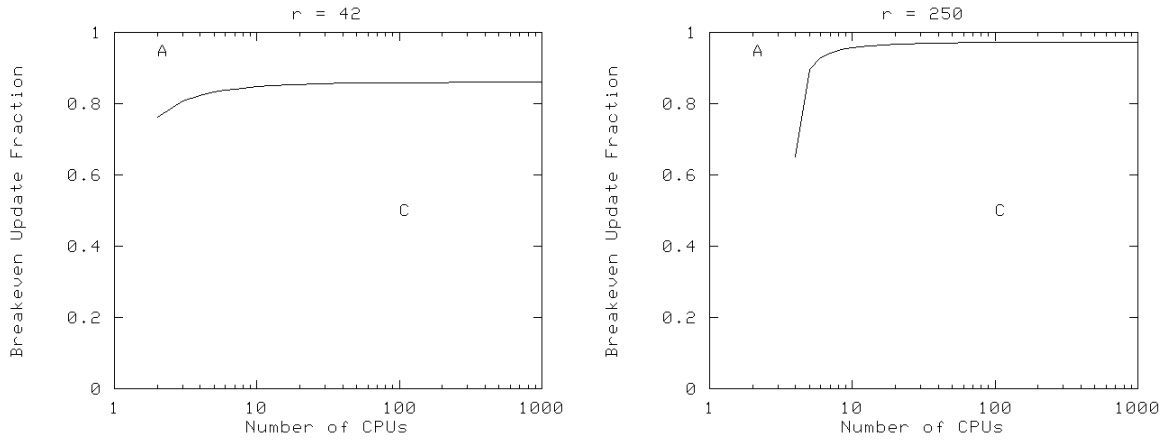

Figure 15: Breakevens for Best-Case Read-Copy Update

11 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.

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 (CPUID) followed an RDTSC 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, 80 NOPs, an RDTSC, and a CPUID is used to calibrate the measurements.

11.1 Rules of Thumb

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

1. The Alpern approximation [Alpern90] that predicts that the ratios of the memory latencies of adjacent levels of the memory hierarchy are equal. In other words, Alpern's model predicts that $t_f=t_m^2/t_s$.
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 CPU's read-side locks.

11.2 Simple Spinlock

Figure 16 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.

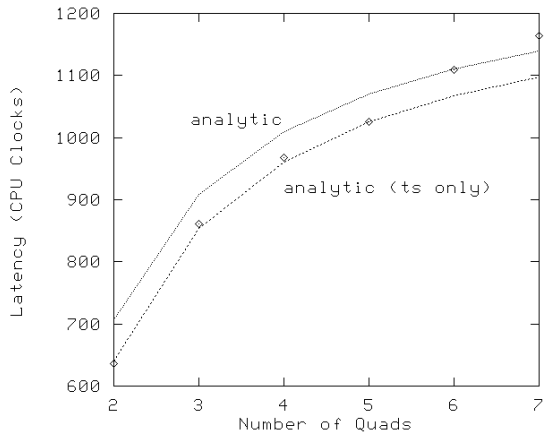

Figure 16: Simple Spinlock

These results validate the rule of thumb that simple spinlock performance can be closely estimated by counting the remote references.

11.3 Distributed Reader-Writer Spinlock

Figure 17 compares the measurements (ticks) against the predictions of the full analytic model. The full analytic model achieves excellent agreement with the measured data.

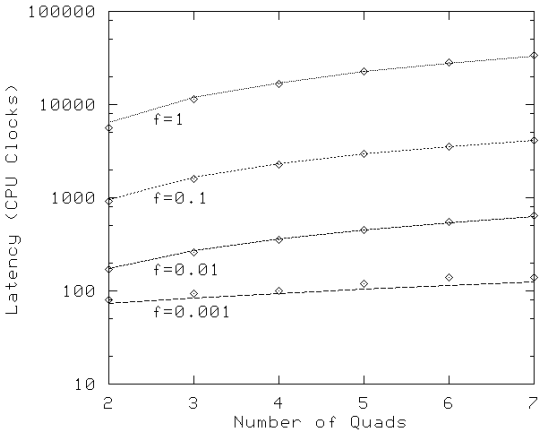

Figure 17: Full Analytic Model, Measured Latencies

Figure 18 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 overhead of locked instructions, which is significant when compared to t_f. 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.

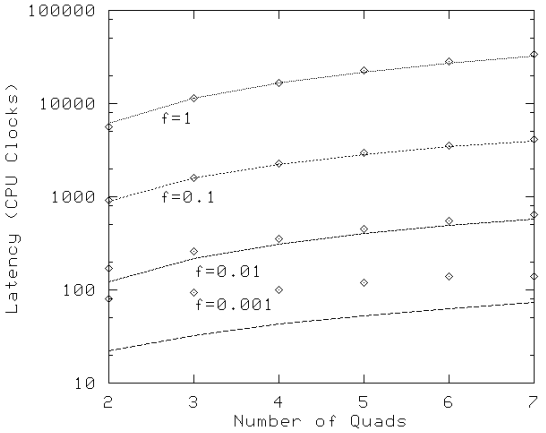

Figure 18: Full Analytic Model, Alpern Approximation

Figure 19 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 give significant error in the small-f regime.

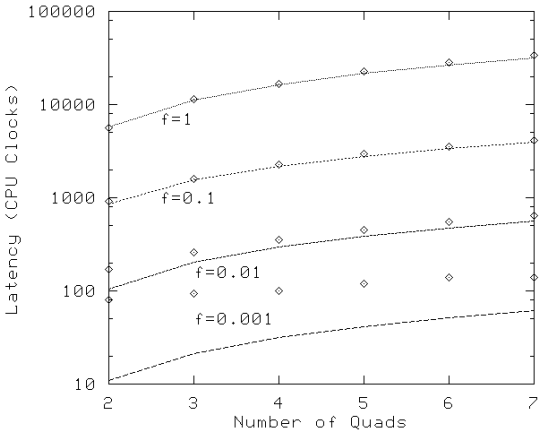

Figure 19: Analytic Model, t_s Term Only

Figure 20 compares the measured data to the naïve model. The naïve model is accurate only for very large values of *f*.

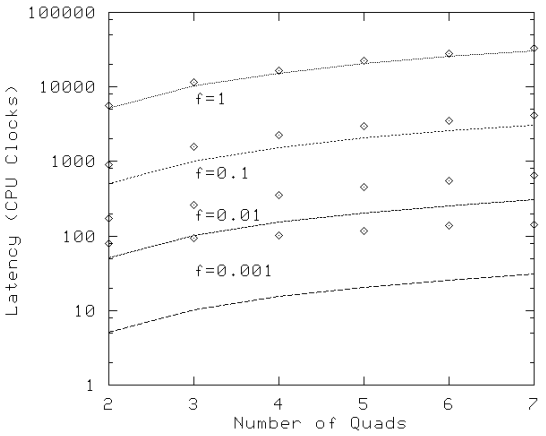

Figure 20: Naïve Model

11.4 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, as was done in Section 10.2. More accurate comparisons for a particular machine require accurate measurement of t_f as well as t_s and t_m.

12 Summary and 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.

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*.

Breakeven curves between read-copy update and the other locking primitives show that read-copy update outperforms the other primitives in a variety of conditions and situations, especially for low update ratio *f*. These comparisons are quite conservative: even greater savings would be realized under conditions of high lock contention.

13 Acknowledgements

I am greatful to Dale Goebel for his support of this work. I owe special thanks to Jack Slingwine for the collaboration that lead to the concept and implementation of read-copy update.

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.

14 References

[Alpern90] Bowen Alpern, Larry Carter, Ephraim Feig, and Ted Selker. The uniform memory hierarchy model of computation. *Foundations of Computer Science Conference Proceedings*, 1990

[Anderson97] 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.

[Burger96] Doug Burger, James R. Goodman, and Alain Kägi. Memory bandwidth limitations of future microprocessors, *ISCA'96*, (May 1996), pages 78-89.

[Hennessy91] John L. Hennessy and Norman P. Jouppi. Computer technology and architecture: An evolving interaction. *IEEE Computer*, page 18-28, Sept. 1991.

[Kung80] H. T. Kung and Q. Lehman. Concurrent manipulation of binary search trees, *ACM Trans. on Database Systems*, Vol. 5, No. 3 (Sept. 1980), 354-382.

[Lea97] Doug Lea. Concurrent Programming in Java, Addison-Wesley, 1997.

[Lovett96] 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*, pages 308-317, May 1996.

[Magnusson94] Peter S. Magnusson, Anders Landin, and Erik Hagersten. Queue Locks on Cache Coherent Multiprocessors. In *8th International Parallel Processing Symposium (IPPS)*, Mexico 1994. (abstract)

[Manber84] Udi Manber and Richard E. Ladner. Concurrency control in a dynamic search structure, *ACM Trans. on Database Systems*, Vol. 9, No. 3 (Sept 1984), 439-455.

[McKenney96] Paul E. McKenney. Selecting locking primitives for parallel programs, *Communications of the ACM*, Vol. 39, No. 10 (1996).

[McKenney98a] Paul E. McKenney. Read-copy update: using execution history to solve concurrency problems, to appear in PDCS'98, October 1998.

[McKenney98b] Paul E. McKenney. Implementation and performance of read-copy update, Sequent TR-SQNT-98-PEM-4, March 1998.

[MellorCrummey91] John M. Mellor-Crummey and Michael L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors, Proceedings of the Third PPOPP, Williamsburg, VA, April, 1991, pages 106-113.

[Pugh90] William Pugh. Concurrent Maintenance of Skip Lists, Department of Computer Science, University of Maryland, CS-TR-2222.1, June 1990.

[Scott92] Michael L. Scott and John M. Mellor-Crummey, Fast, contention-free combining tree barriers, University of Rochester Computer Science Department TR#CS.92.TR429, June 1992.

[Slingwine95] John D. Slingwine and Paul E. McKenney. System and Method for Achieving Reduced Overhead Mutual-Exclusion in a Computer System. *US Patent # 5,442,758*, August 1995.

[Stone91] Harold S. Stone and John Cocke. Computer architecture in the 1990s. *IEEE Computer*, pages 30-38, Sept. 1991.