Empirical Studies of Competitive Spinning for A Shared-Memory Multiprocessor

Anna R. Karlin[*], Kai Li[†], Mark S. Manasse[*], Susan Owicki[*]

Abstract

A common operation in multiprocessor programs is acquiring a lock to protect access to shared data. Typically, the requesting thread is blocked if the lock it needs is held by another thread. The cost of blocking one thread and activating another can be a substantial part of program execution time. Alternatively, the thread could spin until the lock is free, or spin for a while and then block. This may avoid context-switch overhead, but processor cycles may be wasted in unproductive spinning. This paper studies seven strategies for determining whether and how long to spin before blocking. Of particular interest are *competitive* strategies, for which the performance can be shown to be no worse than some constant factor times an optimal off-line strategy. The performance of five competitive strategies is compared with that of always blocking, always spinning, or using the optimal off-line algorithm. Measurements of lock-waiting time distributions for five parallel programs were used to compare the cost of synchronization under all the strategies. Additional measurements of elapsed time for some of the programs and strategies allowed assessment of the impact of synchronization strategy on overall program performance. Both types of measurements indicate that the standard blocking strategy performs poorly compared to mixed strategies. Among the mixed strategies studied, adaptive algorithms perform better than non-adaptive ones.

[*]DEC Systems Research Center, 130 Lytton Ave., Palo Alto, CA 94301.

[†]Dept of Computer Science, Princeton University, Princeton, NJ 08544.

1 Introduction

Lock acquisition on a shared-memory multiprocessor is commonly implemented using an interlocked instruction such as test-and-set or compare-and-swap. With such implementations, an available lock can be acquired with only one instruction. However, when a thread attempts to acquire a lock that is held by another thread, the operation is more costly. There are two common ways of handling this case. The first is *blocking*: the thread relinquishes the processor, after enqueuing itself to be awakened when the lock becomes free. The second is *spinning*: the thread continues to execute on its processor, repeatedly testing to see if the lock is available. Spinning may continue until the lock is free, or the thread may be blocked after some number of trials.

In the first case, processor time is consumed by software overhead, which usually includes saving the thread's state, enqueuing the blocked thread, and scheduling a ready thread on the current processor. In addition, the thread's state must be restored when it is awakened. This time is the cost of a *context-switch*. In the second case, processor time is consumed probing the lock. If the lock is released quickly, this cost may be small.

If there are no other threads waiting for a processor, it makes no sense to block: blocking would only leave an idle processor. On the other hand, if the lock is held by a thread which is waiting for a processor, it makes little sense to spin. No progress is possible until the thread that holds the lock is scheduled. We are primarily interested in situations where neither of these conditions holds. That is, (a) there are other threads waiting for a processor, and (b) the lock is held by a concurrently-running thread.

Condition (a) tells us that spinning would use valuable processor resources, while condition (b) suggests that spinning may be the most efficient thing to do, since the lock may be released soon. Thus an efficient locking algorithm may sometimes choose to spin and sometimes block. Unfortunately, identifying the minimum-cost action requires knowledge of how much longer the lock will be held. This information is usually not available at run time, although we can imagine that sophisticated compiler technology might be able to provide approximations to it. For now, we must settle for approaches that are less ambitious.

The traditional method of dealing with this situation is simple, but inefficient: the thread attempting to acquire the lock is blocked immediately and will only be awakened when the lock is released. We call this method *always-block*. In most implementations, each context-switch takes several hundred instructions. When parallel programs synchronize frequently, the overhead of synchronizations can be very significant. We have observed that, with immediate blocking, some application programs spend over a third of their elapsed time on context-switches.

The simplest alternative to blocking a thread is to spin or busywait until the lock is released. We call this method *always-spin*. If no other threads are ready to run, spinning is certainly the most efficient thing to do, since it reduces the overhead from that of a context-switch to that of the instructions (usually two or three) in the spinning loop. When there are other threads waiting for a processor, it is still advantageous to spin if the lock is held for a time shorter than the time to perform a context-switch. Nonetheless, the worst-case performance of an always-spin strategy is arbitrarily bad: if locks are always held for a length of time significantly longer than the context-switch time, it would be far better to block immediately. If we enter into unbounded spinning when the thread holding a lock is not scheduled, we may even introduce a deadlock! Other than deadlock, there is no reason to believe that this worst-case analysis is worrisome; most concurrent programming techniques suggest that critical sections be kept short.

A compromise between these two methods is to combine spinning and blocking: when a thread fails to acquire a lock, it spins for some time and then blocks. Such a scheme was implemented by Ousterhout [Oust82] in the Medusa system. More recently, it has been shown [KMMO89] that a variant of this method, where the time spent spinning is equal to a context-switch, is *competitive*. This means that the cost of this strategy, amortized over all attempts to acquire locks, is at most twice that of the optimal off-line algorithm. (An off-line algorithm has complete knowledge about how long locks will be held.) However, it has not been clear whether such a strategy would be practical in real system and application programs. In particular, we did not know how much observed performance would differ from that of the optimal off-line algorithm, and how it would compare to the simpler strategies of always blocking or spinning.

Measurements of programs running on shared-memory multiprocessors suggest that different locks have vastly different waiting-time distributions. Consequently, it is attractive to consider strategies that behave differently with different locks. These strategies are based on the assumption that lock-waiting times obey some unknown but time-invariant probability distribution. This assumption seems reasonable in situations where the same sets of instructions are usually executed inside the critical sections, and where lock contention is typically limited to two threads. Such strategies may be *adaptive*, in the sense that they use previously observed lock-waiting times in determining how long to spin in the future.

How do we empirically evaluate spinning strategies? We can implement different strategies and measure a suite of real systems and application programs to see whether the performance is better than the traditional always-block approach. This is a good method; it provides real, convincing data. However, it has two serious drawbacks. The first has to do with the nature of elapsed-time measurements. Many programs, such as operating system components and window systems, cannot be measured by elapsed time because machine clocks are too coarse and because the behaviors of the programs depend on many unpredictable, non-repeatable factors. The second drawback is that such measurements cannot tell us about the cost of the optimal off-line algorithm; implementing that algorithm would require information that is not normally available during program execution, namely, the time until a lock is released.

Another approach to evaluating spinning strategies is to collect information about the distribution of lock-waiting times during program execution and determine the costs of the various strategies by subsequent analysis. This approach is described in more detail in Section 3. Using this method, we can evaluate the cost of synchronization for programs where elapsed-time measurements are difficult. In addition, it provides enough information to compute the cost of the optimal off-line algorithm.

In this paper, we have used both elapsed-time and lock-waiting time measurements to evaluate spinning strategies. We collected lock-waiting time data for five real system and application programs on a 7-processor Firefly. We used the data to evaluate analytically four of the spinning strategies defined in the next section: fixed-spin, optimal-on-line, last-three-samples, and random-walk. We compared them with the optimal off-line, always-spin and always-block algorithms. For elapsed-time measurements, we have implemented most of the spinning strategies and measured those programs in the benchmark suite for which elapsed-time measurements are meaningful. Both evaluation methods show that competitive strategies are practical, that their costs relative to that of the optimal off-line algorithm are gratifyingly small and that they are significantly better than always blocking.

2 The Spinning Strategies

2.1 Background

We say that an on-line algorithm A is *c-competitive* if for every input sequence σ,

$$C_A(\sigma) \leq c \cdot C_{opt}(\sigma) + k,$$

where $C_A(\sigma)$ is the cost incurred by the algorithm A in processing the sequence σ, $C_{opt}(\sigma)$ is the cost incurred by the optimal off-line algorithm in processing σ, and k is some fixed constant. The input sequence σ for the spin-block problem is the sequence of lock-waiting times for each occurrence of a lock which is requested by one process and held by another. If this inequality holds, we say that c is the *competitive ratio* of the algorithm.

We are interested in two kinds of competitive algorithms: deterministic and randomized. Earlier work [KMMO89] has shown that there is a simple deterministic algorithm with a competitive ratio of 2, and that there is no deterministic algorithm that has a competitive ratio smaller than 2. On the other hand, randomized algorithms can achieve strongly competitive ratios approaching $e/(e-1) \approx 1.58$. Similarly, there is an adaptive algorithm that can achieve a competitive ratio also approaching $e/(e-1)$ on input sequences generated according to any time-invariant probability distribution. As mentioned previously, such an assumption about the input sequence is reasonable in situations where the same set of instructions are usually executed inside the critical sections, and where contention for locks is reasonably small.

In the rest of Section 2, we will describe the seven algorithms evaluated in this paper. We will use the optimal off-line algorithm as the baseline against which to compare the others. The algorithms are presented using pseudo-code. In fact, they can be implemented very efficiently in assembly language. `TestAndSet` and `Clear` are usually available as interlocked instructions. The spin loop can usually be implemented with three or four instructions.

2.2 Noncompetitive Algorithms

Always block

As mentioned in the introduction, always-block is the traditional approach to lock acquisition. To acquire a lock, a thread first uses an interlocked instruction such as test-and-set. If the lock is already held by another thread, the acquiring thread is blocked until the lock is released.

```
P_Always_Block( lock ):
  WHILE NOT TestAndSet( lock ) DO
    Block( lock );
  END;

V_Always_Block( lock ):
  Clear( lock );
  IF Blocked( lock ) THEN Wakeup( lock ) END;
```

The overhead of this approach includes blocking, scheduling, and waking up a thread. It is the only reasonable approach for uniprocessors, where the thread holding the lock cannot make progress until it is scheduled to run.

Always spin

Always-spin is also called busywait. Here the thread that failed to acquire a lock spins until the lock is released.

```
P_Always_Spin( lock ):
  WHILE NOT TestAndSet( lock ) DO
    WHILE lock # 0 DO END;
  END;

V_Always_Spin( lock ):
  Clear( lock );
```

On shared-memory multiprocessors, it is important to spin on a cached copy of the lock variable, using the normal test instructions rather than the interlocked test-and-set. This minimizes the traffic on the memory bus and avoids the high overhead of executing an interlocked instruction [And89].

2.3 The optimal off-line algorithm

This algorithm has the best performance that any algorithm can hope to achieve, and therefore we compare all other algorithms to it. Since it runs off-line, it knows how long it will be before the lock is released, and thus it knows whether spinning or blocking is the best choice in each instance. If the lock will be released in less than the context-switch time, the optimal action is to spin and pay the cost of the lock-waiting time. Otherwise, the optimal action is to block immediately and pay the cost of the context-switch time.

```
P_Optimal_Off-line( lock ):
  IF Time( P_Always_Spin, lock ) < ContextSwitch THEN
    P_Always_Spin( lock )
  ELSE
    P_Always_Block( lock );
  END;

V_Optimal_Off-line(lock) = V_Always_Block(lock);
```

2.4 Constant competitive algorithms

The following five on-line algorithms implement lock acquisition with a combination of spinning and blocking. All have the same general structure: on each attempt to acquire a lock, the parameter `SpinThreshold` specifies how many times the process should spin before blocking. The first two algorithms, fixed-spin and optimal on-line, always use the same value for `SpinThreshold`. They are described in this section. In Section 2.5 we describe three algorithms that adjust `SpinThreshold` based on observed waiting times.

```
P_Constant_Spin( lock, SpinThreshold ):
  IF NOT TestAndSet( lock ) THEN
    count := SpinThreshold;
    REPEAT
      WHILE ( lock # 0 ) AND ( count # 0 ) DO
        DEC( count )
      END;
      IF count = 0 THEN
        Block( lock );
      END
    UNTIL TestAndSet(lock);
  END;

V_Constant_Spin( lock ) = V_Always_Block( lock );
```

The parameter `SpinThreshold` may be identical for all locks or may vary for different locks.

Fixed spin

Fixed-spin [KMMO89] is the simplest competitive algorithm for spinning. The number of spins before blocking is taken as a fixed fraction of the spins required for a context switch. In our analysis, we considered two values for `SpinThreshold`: `C` and `C/2`, where `C` is the context switch time measured in spins.

It is easy to see that fixed-spin with `SpinThreshold` equal to `C` has a competitive ratio of 2. Suppose the lock is held for t time units. If $t < C$, the optimal off-line algorithm just spins, paying a cost of t, while fixed-spin spins until the lock is free, for a cost of t, the same as the optimal algorithm. If $t \geq C$, the optimal off-line algorithm blocks immediately, paying a cost of C, while fixed-spin spins for time C and then blocks, for a total cost of $2C$. Therefore, fixed-spin's cost is always at most twice that of the optimal off-line algorithm.

A similar analysis shows that fixed-spin with `SpinThreshold` equal to `C/2` has a competitive ratio of 3. For many of the distributions we collected, spinning for half a context-switch time gives a lower synchronization cost than spinning for an entire context-switch time, even though the competitive ratio is not as good. This is not at all surprising: competitive ratios say nothing about average performance.

Optimal on-line for a known distribution

The optimal on-line algorithm uses a lock-waiting distribution $\mathcal{P}_L$ for each lock L to calculate a spin threshold. If we consider the family of algorithms that spin for a time τ before blocking, then the best deterministic on-line algorithm is the one which minimizes the expected cost under the distribution $\mathcal{P}_L$ [KMMO89]. In practice, distributions are discrete data points. Thus, the optimal on-line method computes k to minimize:

$$\sum_{1 \leq i \leq k} i D_L[i] + (k + C) \sum_{k+1 \leq i \leq max} D_L[i].$$

where $D_L[i]$ is the probability that lock L is held for more than $i - 1$ and at most i spins. The value of k determined is used as the `SpinThreshold`.

Since the distribution D_L is not known before the program is executed, this method is not practical for programs that only execute once.

2.5 Variable competitive algorithms

The following three on-line algorithms are similar to those in the previous section. Here, however, the value of `SpinThreshold` varies during execution, based on the observed waiting times for lock acquisition. The waiting time for lock acquisition that we would like to measure is the number of spins that this process would have executed before successfully acquiring the lock had it spun. Unfortunately, if the process blocks, the history of events may change and it becomes impossible to determine this value exactly. Therefore, we approximate the lock waiting times, as specified in the following procedures. The three adaptive algorithms are described by the following code; they differ in the way procedure `Adjust` updates `SpinThreshold`. Note that we have modified Block to return the time at which the lock was most recently released.

```
P_Variable_Spin( lock, SpinThreshold ):
  IF NOT TestAndSet(lock) THEN
    spinsWaited := 0;
    count := SpinThreshold;
    REPEAT
      WHILE ( lock # 0 ) AND ( count # 0 ) DO
        DEC( count )
      END;
      IF count = 0 THEN
        now := GetCurrentTime();
        Block( lock , releaseTime );
        timeBlocked := MAX ( 0, releaseTime - now );
        INC( spinsWaited, timeBlocked * SpinsPerSecond )
      END
    UNTIL TestAndSet(lock);
    INC( spinsWaited, SpinThreshold - count );
    SpinThreshold := Adjust( SpinThreshold, spinsWaited )
  END;

V_Variable_Spin( lock ):
  Clear( lock );
  IF Blocked( lock ) THEN
    releaseTime[threadToBeAwakened, lock]
        := GetCurrentTime();
    Wakeup( lock );
  END;
```

Optimal on-line approximation

In order to apply the optimal on-line method to the first execution of a program, one has to use an approximation to the distribution D_L. One way to approximate D_L is to collect lock-waiting distributions, in terms of spins, as the execution proceeds. The P operation can collect such a distribution and then use it to recompute the optimal `SpinThreshold` in the `Adjust` procedure.

Unfortunately, the computation takes time linearly proportional to the number of data points in the distribution D_L. In order to make this method practical, the number of data points in D_L has to be small. This means that either *max* has to be rather small or each data point in the distribution has to be coarse.

The theory behind such a method is that if the distribution of lock-waiting times does not vary with time, the sample statistics will converge to the true values, and the cost will approach that of the optimal on-line algorithm. Let A^* be the deterministic algorithm that minimizes the expected cost on lock-waiting time sequences $\sigma(\mathcal{P})$, where $\sigma(\mathcal{P})$ is generated according to a time-independent distribution $\mathcal{P}$. It has been shown [KMMO89] that

$$\mathbf{E}C_{A^*}(\sigma(\mathcal{P})) \leq \frac{e}{e-1}\mathbf{E}C_{opt}(\sigma(\mathcal{P})).$$

So, an algorithm that uses sample statistics of lock-waiting times in order to estimate the distribution $\mathcal{P}$ converges to $e/(e-1)$ competitive behavior or better.

Last three samples

This algorithm is a simple but somewhat crude approximation to the optimal online algorithm. As the name suggests, the algorithm uses the last three waiting times to estimate the distribution of waiting times.

Procedure `Adjust` determines the value of `SpinThreshold` that minimizes expected cost exactly as the optimal on-line algorithm does. The only difference is in the distribution assumed for lock-waiting times. Suppose the last three waiting times for lock L were τ_1, τ_2 and τ_3. The optimal value for `SpinThreshold` is computed under the assumption that the lock-waiting distribution D_L is the average of $D_L^{\tau_1}$, $D_L^{\tau_2}$ and $D_L^{\tau_3}$, where D_L^{τ} is the distribution with a point mass at τ.

Random walk

The random-walk algorithm is based on the assumption that the probability of waiting more than a context-switch time for a lock to be released is high if a thread previously has waited for more than a context-switch time.

The `Adjust` procedure for this algorithm works as follows. If spinsWaited is more than a context-switch time, `SpinThreshold` is decremented by 1 unit so long as it is positive. Otherwise, `SpinThreshold` is incremented by 1 unit to a maximum of a context-switch time.

3 Lock-Waiting Distributions

As mentioned in Section 1, we compared the efficiency of lock-acquisition algorithms using two kinds of measurements. The first is direct measurement of elapsed time for program execution. The second involves recording the distribution of lock-waiting times during one execution of the program and deriving the performance of each algorithm based on that data. In this section we describe the Firefly computer used for our experiments and give some details on how the waiting-time distribution is obtained. We then describe the characteristics of the distributions observed for five system and application programs.

The Firefly [TSS88] is a shared-memory multiprocessor workstation developed at the Digital Equipment

Corporation's Systems Research Center (SRC). The Firefly used for our experiments has seven processors: one MicroVAX 78032 (used exclusively for I/O) and 6 CVAX 78034 processors. Each processor has a floating point accelerator and a cache. The MicroVAX processor has a cache of 16 Kbytes and each CVAX processor has a cache of 64 Kbytes. Our configuration included 32 Mbytes of main memory. The caches are coherent, so that all processors see a consistent view of main memory. The Firefly operating system is called Taos. It emulates the Ultrix system call interface and provides support for multiprocessing through multiple threads of control in a single address space.

There are a number of difficulties in collecting lock-waiting time distributions. The main problem is coarse clock granularity. Most machine clocks have a granularity of milliseconds, but many critical sections may execute in microseconds. Another problem is that measurement consumes time, and this may change the pattern of interaction among threads in a way that affects lock-waiting times. A third problem, is that it is not possible on our system to collect lock-waiting times without changing the implementation of the fast path code for lock acquisition, which is built into the compiler.

To deal with the first problem, we measured lock-waiting times by counting the number of times a spin loop was executed. We chose to use 16 spins as the unit for waiting-time distributions. This unit is small enough to allow comparison of different methods, yet coarse enough to require only a small amount of memory to store the distributions. We let each thread spin up to 1008 times (63 units of lock-waiting time) before blocking and then recorded the distributions using two-level hash tables. The last unit includes all cases where the lock-waiting time was greater than 992 spins, including cases where the thread eventually blocked. Each blocking time of a thread is measured using the system clock and then accumulated in spin-equivalents.

To deal with the second problem, our data collection is designed to have a minimal effect on program execution. We do not record information when a thread finds the lock available, since we are only interested in lock-waiting times. Thus, data collection does not introduce any overhead for the common, fast path of acquiring a lock. To further reduce overhead, we carefully hand-coded the implementation in assembly language and used data structure sizes that expedited hashing. The fast path for hashing takes only five VAX instructions. Since Fireflies maintain strong memory coherence at the level of a long word (4 bytes), we do not need critical sections to protect the shared hash tables, although some updates might be lost.

To deal with the third problem, we measured an approximate value for lock-waiting times. Our measurements required changes only to the slow path code, which is part of the run-time library and thus much easier to modify. With this change, we use the time until the lock is next released as an approximation to the lock-waiting time whenever a thread that is waiting for a lock spins at least that long. If the thread that is waiting for a lock spins a shorter time, we obtain the true lock-waiting time. The time until the lock is next released agrees with the true lock-waiting time as long as there are at most two contenders for the critical section. We expect this condition to hold most of the time. As discussed in section 7, this expectation was confirmed by additional measurements we obtained.

We selected the operating system and 4 programs to measure for lock-waiting time distributions: Taos, Ivy, Galaxy, Hanoi, and Regsim. They represent a spectrum of system and application programs that have a high degree of data sharing and parallelism. In the rest of this section, we present the total lock-waiting distributions for each program. In the next section, we use the distributions for individual locks to compare different spinning strategies.

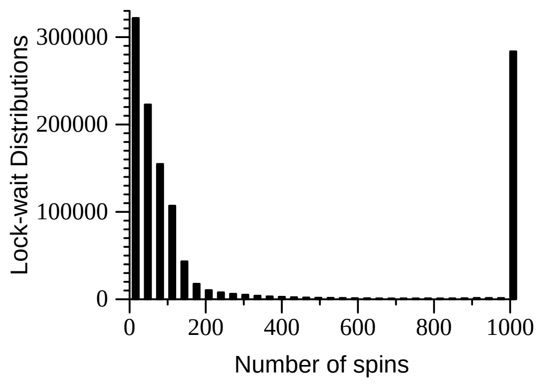
Taos

Taos is the operating system for the Firefly [MS87]. It has two component address spaces (and their associated threads of control): a "kernel" address space called the Nub and the Taos address space.

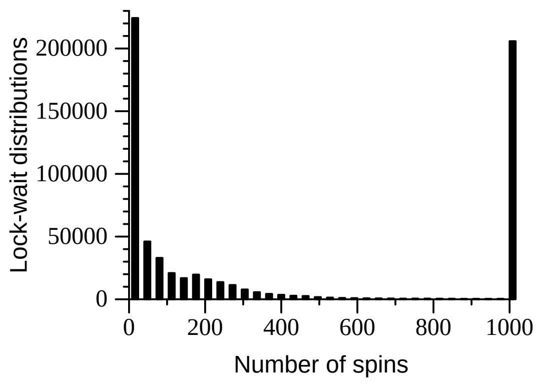
The Nub resides in the system region of the VAX virtual memory, and is effectively part of every address space. It implements virtual address spaces, threads, and basic device drivers. The Nub's implementation of threads schedules the highest-priority ready threads to available processors without regard to address space. Thus it is possible for one program/process to use all the processors at once.

The Taos address space contains the rest of the operating system. It implements processes, files, windows, and the like. Firefly applications communicate with the Taos address space by remote procedure call. In addition, Taos supports standard Ultrix applications (including csh, cc, emacs, and make) by emulating the Ultrix system-call interface.

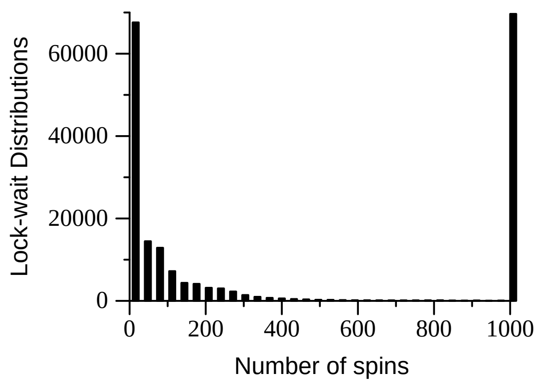
Since different parts of Taos are exercised by different tasks, we collected lock-waiting times for four instances. Figure 1 (a) shows the combined lock-waiting time distributions of 283 locks in the Nub address space over two hours, running a mixture of jobs. Figure 1 (b) shows the combined lock-waiting time distributions of 343 locks in the Taos address space over 24 hours. The



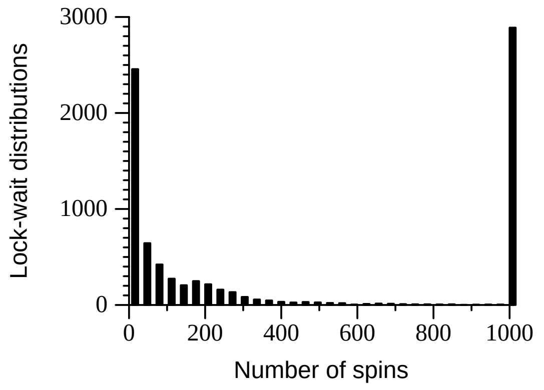
(a) Nub with mixed jobs (2 hours, 283 locks)



(b) Taos with mixed jobs (24 hours, 343 locks)



(c) Taos with parmake m2+ system (2 hours, 232 locks)



(d) Taos with Regsim (2.5 hours, 150 locks)

Figure 1: Distribution of lock-waiting times for Taos.

distributions are similar, although the Nub has more cases where the waiting time is small but exceeds our minimum sample.

Figure 1 (c) shows the combined waiting-time distributions of 232 locks while building the Modula-2+ compiler using a parallel version of make. This task took 2 hours. Figure 1 (d) shows the distributions for 150 locks during a 2.5 hour execution of Regsim, a register-level simulator. Although the running times are similar, there is substantially more contention while building the compiler, as can be observed by looking at the scale of the graphs. This is because the parallel make runs compilers and linkers concurrently, and the file system has a large degree of concurrency.

All three lock-waiting time distributions for the Taos address space are similar. A large percentage of the waits require only a few spins. Another large percentage require blocking. Such distributions imply that the amortized cost of competitive spinning strategies will be better than the always-block approach, but always-block can give acceptable efficiency.

Ivy

Ivy is a programmable text editor that runs in a single address space. It makes heavy use of the Trestle window system developed at SRC.

Ivy was implemented in Modula-2+ [Rov86] and Tinylisp, a lisp dialect that interfaces with Modula-2+ and its run-time library. Ivy was designed for the Firefly, and it makes heavy use of concurrency. A large number of locks are used for mutual exclusion.

Figure 2 shows the lock-waiting time distributions for two editing sessions, the first lasting 100 minutes and the second 18 hours. The first session was continuous and intense, while the second includes some idle time. In both cases, over 30 local and remote files were edited. The window system was exercised heavily to display a variety of windows on the files. The number of locks in the distributions is quite large; both Ivy itself and the underlying window system use concurrency and locks heavily.

In these cases, unlike the Taos address space, only a small percentage of waits led to blocking. A very large percentages of waits are shorter than 32 spins.

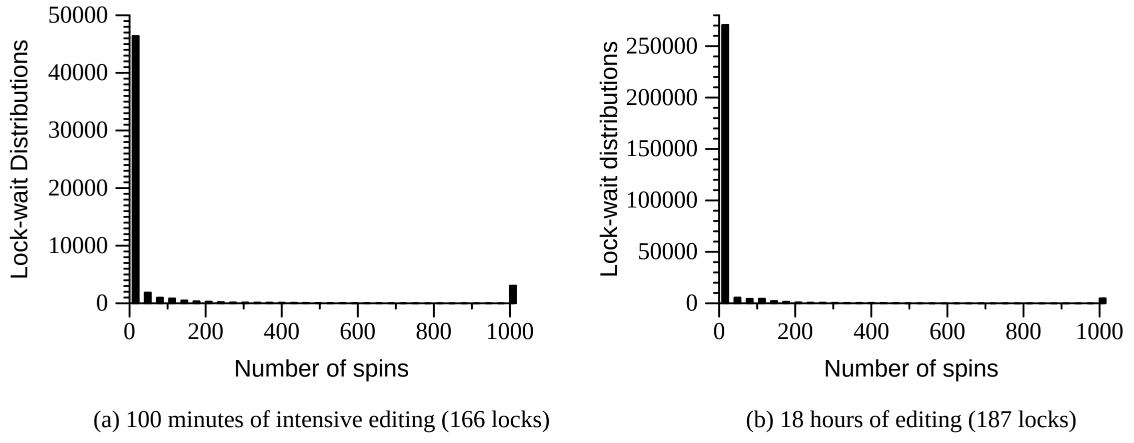


Figure 2: Distribution of lock-waiting times for Ivy.

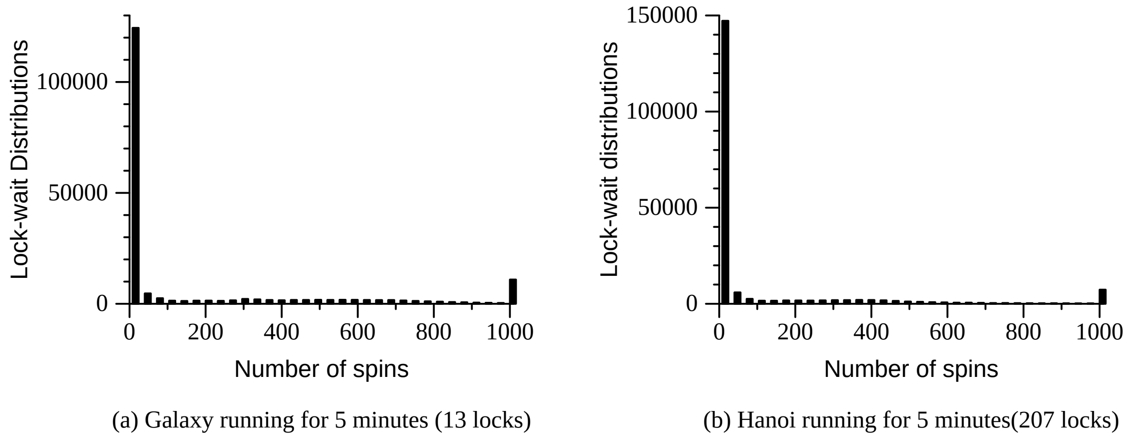


Figure 3: Distribution of lock-waiting times for Galaxy and Hanoi.

Such distributions indicate that competitive spinning strategies such as fixed-spin should significantly reduce synchronization overhead and outperform the always-block approach.

Galaxy and Hanoi

Galaxy is a parallel program that simulates near-collisions of two galaxies on the Firefly display. Physically, the mass of each galaxy is considered to be located at a point at its center, and the "stars" that surround the galaxy are taken to be of zero mass. Thus acceleration depends only on the relative position of the two centers.

Hanoi is a parallel program that solves the well-known problem of the Buddhist monks of Hanoi. The program uses five pegs and 64 disks of different sizes. At the start, all 64 disks are on the first peg, with larger disks lying below smaller ones. At the end, all 64 disks are on the last peg, in the same arrangement. Disks are moved from peg to peg, but a larger disk is never placed atop a smaller one. A large number of threads are used in the program, and there is frequent synchronization.

Both programs are written in Modula-2+ and use all 6 processors well.

Figure 3 shows the lock-waiting time distributions for both programs. The number of locks involved are quite different: 13 locks in Galaxy and 207 in Hanoi. The main reason for the large number of locks in Hanoi is heavy use of mutual exclusion in managing the display. The two distributions are similar to each other and to the Ivy distributions: most lock-waiting times are short.

Regsim

Regsim is a register-transfer-level simulator for synchronous digital logic. It has been used extensively for hardware design at SRC.

Regsim is a parallel program implemented in Modula-2+. It models collections of connected objects. Each object has a single output, which may connect to inputs of other objects, and an evaluation procedure. The procedures are evaluated in parallel until the output values stabilize. The objects are initially divided evenly among the available threads. During each phase, the thread that finishes last gives one of its objects to another thread.

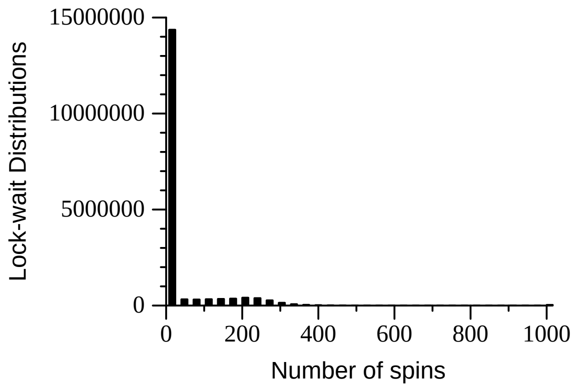


Figure 4: Distribution of lock-waiting times for Regsim.

Figure 4 shows the total lock-waiting time distribution for simulating a portion of a processor design, a task that took about 2.5 hours. This program has a large synchronization overhead, and very few waits require blocking. The distributions imply that competitive spinning strategies and always-spin should perform well.

4 Analytic Results

Our first approach to comparing spinning strategies is to estimate synchronization costs based on the observed distributions of waiting times described in the previous section. In this analysis, the probability that a lock-wait takes k time units is taken to be equal to the fraction of observed waits that took k time units. Given these probabilities, it is easy to compute the expected cost of each strategy, that is, the average time spent spinning or blocking for lock acquisitions that involve contention. We computed the cost of each of the strategies described in Section 2 for each of the programs described in Section 3. Note that the waiting-time distributions used in this analysis were accumulated separately for each lock. Thus they contained more detailed information than the combined distributions reported in Section 3.

Table 1 gives the cost of each spinning strategy, relative to the cost of the optimal off-line algorithm. For example, Galaxy spends 23.7% more time spinning and blocking with the optimal on-line strategy than it does with the optimal off-line strategy. These numbers allow us to compare the effectiveness of the strategies. However, the impact of strategy on overall program performance depends on the frequency of lock-contention, among other factors. This is discussed further in Section 5.

It is obvious that always-block and always-spin perform poorly in almost all cases. The fixed-spin approach is significantly better than always-block or always-spin, but generally not as good as the adaptive strategies. Among the three adaptive approaches, optimal on-line has the lowest cost, as we would expect. Last-three-samples might be slightly better than random-walk: it is never as badly fooled by the observations. Note that all the competitive methods have competitive ratios less than 1.8 for all programs.

Table 2 gives further information about the behavior of the optimal off-line algorithm. The first column gives the percentage of cases in which the optimal off-line algorithm blocks. The second gives the average time spent spinning in those cases where spinning is optimal. These two numbers completely determine the average cost, since the time for a context switch is assumed to be a constant 200 usec. The final column gives the fraction of the elapsed time spent on synchronization by the optimal off-line algorithm, for those programs where elapsed-time numbers are meaningful. This can be taken as a very rough estimate of synchronization overhead.

We can make two interesting observations about this data. First, the fraction of cases where blocking is optimal varies dramatically from program to program. This suggests that an adaptive algorithm has considerable potential for improving performance. Second, when spinning is the optimal choice, the average time spent spinning (22-64 usec) is considerably less the time for a context switch (200 usec). This suggests that a fixed-spin algorithm will do well to have a relatively short spin-time; this is confirmed by the relative performance of the two fixed-spin algorithms.

The analytic model gives precisely the same cost as a trace-driven simulation would give for those algorithms that are not sensitive to the order in which the

	Block	Spin	Fixed C/2	Fixed C	Opt Online	3-samples	R-walk
Nub (2h)	1.943	2.962	1.503	1.559	1.078	1.225	1.093
Taos (24h)	1.715	3.366	1.492	1.757	1.141	1.212	1.213
Taos (M2+)	1.776	3.535	1.483	1.750	1.106	1.177	1.160
Taos (Regsim)	1.578	3.293	1.499	1.748	1.161	1.260	1.268
Ivy (100m)	5.171	2.298	1.341	1.438	1.133	1.212	1.167
Ivy (18h)	7.243	1.562	1.274	1.233	1.109	1.233	1.141
Galaxy	2.897	2.667	1.419	1.740	1.237	1.390	1.693
Hanoi	2.997	2.976	1.418	1.726	1.200	1.366	1.642
Regsim	4.675	1.302	1.423	1.374	1.183	1.393	1.366

Table 1: Synchronization costs for each program relative to the optimal off-line algorithm

	Optimal Offline	Optimal Offline	Opt %
	% Block Cases	Average Spin Cost (usec)	Elapsed Time
Nub (2h)	28.7	63.77	
Taos (24h)	44.1	50.66	
Taos (M2+)	42.2	48.75	
Taos (Regsim)	47	60.66	
Ivy (100m)	8.4	23.7	
Ivy (18h)	3.2	21.87	
Galaxy	25.5	24.06	4.1
Hanoi	16.5	24.02	8.5
Regsim	8.3	27.88	9.9

Table 2: Behavior of the optimal off-line strategy.

waits occur (optimal off-line, block, spin, fixed-spin, and optimal on-line). The three adaptive algorithms (approximate optimal on-line, last-3-samples, and random walk) are influenced by the order in which waits occur, so the results of our analysis may differ from the true costs. If, for example, there was high locality in the sequence of waiting times, our results would actually be pessimistic. On the other hand, it is possible to construct perverse orders for which our predictions are overly optimistic.

Of course, actual measurements of synchronization costs could give different results from those predicted here, for a variety of reasons. The most striking is that lock-waiting times can be affected by the spinning strategy used.

5 Elapsed-time Measurements

In the previous section, we compared the synchronization costs of various spinning strategies. In this section, we will examine the effect of spinning strategy on overall program performance. Note that large differences in synchronization cost will not necessarily lead to correspondingly large differences in overall performance. There are several reasons for this:

- A program that does not spend much time in contention does not gain much by improved spinning strategies.
- The analytic results do not include the implementation overhead of the algorithms.
- Saving CPU time by blocking does not lead to an elapsed time gain unless there are unscheduled threads that can use the free processor.
- Saving CPU time on a noncritical path will not reduce elapsed time.

On the other hand, there are also reasons why elapsed-time measurements might show bigger improvements than the raw synchronization costs would lead us to expect:

- The actual context switch time may be significantly larger than the value used for the analytic results. That value was obtained by measuring a simple program that executed context switches as rapidly as it could. The work of Mogul and Borg [MB90] suggests that the a context switch should also take into account the increase in cache misses and bus utilization.

- When there are fewer schedulable threads than processors, our analytic results still assume that spinning is costly, whereas in reality it never slows the program down.

Thus elapsed-time measurements give information that is complementary to the analytic results on synchronization costs.

For the elapsed-time measurements, we implemented always-spin and the competitive strategies fixed-spin, optimal on-line (exact and approximate), last-three-samples, and random-walk. For always-block, we used the existing implementation in the run-time library. Here we sketch a few details of the implementations.

The fixed-spin method is straightforward. Since the time for a context-switch (C) is equivalent to about 200 spins, we measured our programs with both 100 ($C/2$) and 200 (C) spins.

The exact and approximate optimal on-line algorithms are implemented in different ways. The exact optimal on-line method uses the lock-waiting distributions collected for the analytic computation. For each lock, we pre-computed the optimum threshold to be used during program execution.

In the implementation of approximate optimal on-line and last-three-samples, a bound on the maximum observed waiting time is needed to limit the size of tables. We used 1008 spins in both cases, in order to compare the measurements with the analytic results. For the optimal on-line approximation, we also limited table size and running time by rounding spinning observations up to multiples of 64. For last-three-spins we rounded to multiples of 16 to save table space.

For the random-walk algorithm, the threshold is adjusted in increments or decrements of 16 spins.

For all the adaptive methods, a two-level hash table is used to maintain reduced observation data and the spinning threshold for individual locks. Due to the absence of fine-grained timing mechanisms on our processors, we use our coarse-grained clock to approximate the waiting time when a thread blocks.

We measured elapsed time for two programs, Regsim and Hanoi, using all strategies. The measurements were averaged over three runs. Other programs were not included, because most of our traces are for interactive use of the system, and we do not have a good way to measure interactive response time. However, the analytic results suggest that competitive strategies would improve response time for the interactive programs during periods of heavy contention.

Table 3 shows elapsed times for Regsim under different competitive spinning strategies. The performance of always-block, the currently-implemented strategy, is used as the baseline for comparisons.

	Max spins	Elapsed time (seconds)	Improvement
Always-block	N/A	10529.5	0.0%
Always-spin	N/A	8256.3	21.5%
Fixed-spin	100	9108.0	13.5%
	200	8000.0	24.0%
Opt-known	1008	7881.4	25.1%
Opt-approx	1008	8171.2	22.3%
3-samples	1008	8011.6	23.9%
Random-walk	1008	7929.7	24.7%

Table 3: Elapsed times of Regsim using different spinning strategies.

For Regsim, always-spin performs quite well, as predicted by the analytic results. This is because the number of computational threads in the program is the same as the number of processors. Although there are a few other threads, such as one for garbage-collection, they consume a small fraction of the available processing power. Thus spinning is usually the best strategy, because there is likely to be no other thread to use a freed processor.

The measurements show that the cost of synchronization in such a parallel program can be substantial. In Regsim, most of the competitive spinning methods save a quarter of the elapsed time over the traditional always-block approach. Thus speeding up the synchronization primitives yields a significant improvement in program performance. The exception is fixed-spin with a threshold of $C/2$, which only saves about half as much. We conjecture that there is rarely an advantage to blocking, because there is seldom a waiting thread to use a free processor. Fixed-spin with a small threshold forces more blocking than the other competitive strategies, so its performance is worse.

Table 4 shows the elapsed times for Hanoi under different spinning strategies. Once again, always-block is used as the baseline for comparisons.

As predicted in the analytic results, always-spin performs badly. In fact, it is worse than always-block, which would not be expected from the analytic results. This may be explained by noting that in the distribution data, long waiting times were cut off at 1000 spins. If the actual waiting times were significantly larger, the only analytic results that would change would be those for always-spin, which would become proportionally worse.

Fixed-spin did worse with a threshold of $C/2$ than with a threshold of C, which disagrees with the analytic results. Also, the adaptive algorithms do better

	Max spins	Elapsed time (seconds)	Improvement
Always-block	N/A	193.9	0%
Always-spin	N/A	241.2	-24.4%
Fixed-spin	100	170.8	11.9%
	200	164.8	15.0%
Opt-known	1008	122.1	37.0%
Opt-approx	1008	135.6	30.1%
3-samples	1008	134.6	30.5%
Random-walk	1008	131.6	32.1%

Table 4: Elapsed times for Hanoi using different spinning strategies.

than would be expected from the analytic results. A likely explanation for all of these anomalies is that the number of active threads in Hanoi is quite bursty. We have observed that there are intervals during execution when all processors are busy, and others where only half are busy. This is not accounted for in the analytic numbers.

All of the adaptive algorithms perform very well. The optimal on-line strategy for a known distribution performs the best. However, this method is not practical; it requires a known distribution and a hash table for each lock to store the precomputed values of `SpinThreshold`.

It is surprising that the improvements due to spinning strategies are much larger than can reasonably be explained by the distribution information collected for the analytic results. From that data, it would appear that the total time spent by always-block in context switches is approximately 38 seconds out of 194. Even the optimal algorithm would appear to require at least 10 seconds for synchronization, although this could be an overestimate due to the coarse bucket size used in accumulating the distribution.

It seemed odd to us that shaving 38 seconds off a program could save over 70, which is what was observed in the elapsed time measurements. So we spoke to the author of the program, who believes that competitive strategies actually reduced the amount of synchronization performed during program execution. Here is an explanation of how the program's structure makes that possible. There are many threads in the program. Each thread has a fixed task; to accomplish this task it needs to do an amount of synchronization that depends on the relative timing of events in other threads. A strategy that introduces spinning is likely to shift those relative timings in a way that reduces both lock contention and the total amount of synchronization that each thread must do. This could account for the improvement noted with all the competitive strategies. Always-spinning, however, is a mistake, because there are some locks that are held for a very long time. The adaptive strategies do best because they adjust to the waiting-times of the various locks in the program.

It is also possible that the analytic results systematically underestimate the cost of blocking in a heavily loaded system. This would be the case if context switches actually become more time-consuming under heavy loads. To determine how larger context-switch times would affect the analysis, we re-evaluated synchronization costs assuming that a context switch costs 500 microseconds. In that case, always-block would spend 116 seconds in synchronization, versus 38 seconds for exact optimal on-line.

We plan further experiments to gain a better understanding of this program's behavior.

6 Related work

There is a large body of literature related to implementing synchronization primitives on shared-memory multiprocessors.

Early work was concerned with implementing critical sections when the only atomic operations provided by hardware are memory read and write [Dij65, Knu66]. A number of software protocols have been developed, based on these operations and coherent shared memory. The main disadvantage of these approaches is their inefficiency. The most efficient approach [Lam87] still requires five writes and two reads in the absence of contention.

Greater efficiency is possible when the hardware provides more powerful atomic operations. Simple atomic operations such as test-and-set and compare-and-swap simplify synchronization implementations and can even allow certain certain concurrent data structures to be implemented without blocking [Her87, Her90]. More powerful atomic operations, such as fetch-and-add [GGK*83] allow certain common operations to be performed in parallel without critical sections.

Other work has evaluated the cost of synchronization, considering such complexities as the hardware cache-coherence algorithm, operating system implementation of threads, and the non-deterministic nature of the workload. Ousterhout [Oust82] noted that blocking should be avoided when a lock would be available in time less than a context switch. His Medusa system delayed blocking a thread for a fixed time determined by the user. Zahorjan, Lazowska, and Eager [ZLE88] assume that the decision to spin or block will be made by the user; they examine the extent to which multiprogramming and data-dependencies in an application complicate this decision. In later work [ZLE89] the same authors evaluate how the overhead of spinning is affected by various scheduling policies. Other studies [And89, ALL89, GT90] have compared alternatives for implementing synchronization and thread scheduling, using both modeling and empirical data. Among other things, they consider the cost of contention for a lock variable in shared memory, and describe a number of schemes aimed at minimizing the cost of that contention. Related work [ABLL90] proposes operating system support for moving many thread scheduling decisions into the application program.

In recent work, B. Lim and A. Agarwal [LA91] have studied algorithms for the producer-consumer type synchronization used by "futures". They developed analytic models for lock-waiting times and studied the performance of fixed-spin algorithms under these models. Their paper presents both analytic and simulation results.

Competitive algorithms were first used for paging and snoopy caches [ST85, KMRS88]. Recent work on competitive spinning strategies [KMMO89] found that the fixed spin competitive strategy costs at most twice as much as the optimal off-line algorithm and that a randomized algorithm can achieve strongly competitive ratios approaching $e/(e-1) \approx 1.58$.

7 Limitations

This study demonstrates the performance advantage of competitive spinning in a particular computing environment, the Firefly multiprocessor running the Taos operating system. It is not clear to what extent it is possible to predict the performance of competitive spinning on other multiprocessors by extrapolating from these results.

The Firefly has a relatively small number of processors, and these processors are old and slow. We do not have any data on whether the algorithms will perform well as the number and speed of the processors increases. We conjecture though, that on faster machines the advantages of competitive strategies will be even greater than those we observed for the following simple reason. Faster machines rely on effective caching. Since context switches reduce cache hit ratio [MB90], it is likely that the performance impact of these algorithms will become even more pronounced, at least when compared to always-block.

The thread system used in our measurements is a kernel-level implementation. The cost of a context-switch is larger than it would be with a user-level implementation of threads [ABLL90]. It is difficult to draw any conclusions from our measurements about the performance of the algorithms if context-switching is significantly cheaper. In general, though, reduced context-switch time might be expected to decrease the value of competitive algorithms, since the penalty for blocking inappropriately is reduced.

Taos provides no mechanism to prevent the preemption of a thread that is holding a lock. It is possible that some of the long waiting times we observed occurred because the lock holder was preempted. On the other hand, user-level thread packages [ABLL90] may have enough information to prevent preemption in this case. It is not clear how such cases would affect the performance of the competitive algorithms.

In order to compare different competitive spinning strategies with the optimal off-line algorithm, we collected lock-waiting time distributions for a number of programs including the operating system and application programs. Ideally, lock-waiting times would be measured from the time when acquisition was attempted until the lock was acquired. However, we had to settle for measuring the time until the lock was next released. Getting the ideal measurement would have required changing the implementation of the fast path code for lock acquisition, which is built into the compiler. Obtaining our measurements only required changes to the slow path code, which is part of the runtime library and thus much easier to modify. The two measurements will differ only when there are more than two contenders for a held lock. In order to determine how much our approximation to lock-waiting times affected the results, we performed some additional measurements on Regsim and Hanoi and found that for these programs, more than 90% of the time that there is contention for a lock, there are only two contenders. Therefore, at least with respect to the elapsed time results, our approximation is usually correct and thus has minimal effect. We suspect that our approximation to lock-waiting times is correct more generally.

An alternative to competitive strategies, in which the system decides whether and how long to spin, is to allow the programmer to make the decision. Knowledge of the application can be used to place spin-locks around short critical sections and blocking locks where relatively long waits are likely. If the programmer chooses wisely, performance may equal or exceed that of the competitive algorithms.

However, there are several advantages to competitive algorithms. First, they relieve the programmer from an unnecessary burden. Since the competitive strategies appear to perform well, there seems to be little reason to require the programmer to make the choice. Second, the choice is not always obvious; in these cases competitive algorithms may well give better performance than programmer-selected strategies. Finally, changes in an application, or in the environment where the application runs, may change the performance tradeoff between spinning and blocking. Competitive algorithms avoid the need to reconsider all decisions in such cases.

8 Conclusions

This paper evaluates several competitive spinning strategies for synchronization on shared-memory multiprocessors. Where possible, we measured improvements to elapsed time. Where that was not possible, we used an analytic approximation based on the observed distribution of lock-waiting times.

The lock-waiting distributions that we have collected indicate that there is large variability between lock-waiting times for different locks and between different programs. This suggests that no fixed non-adaptive algorithm can perform uniformly well, even on the small sample of programs we studied. Our results also suggest that easily-implementable spinning strategies can attain nearly optimal performance. At present, programmers often go to great lengths to avoid even momentary contention for a lock. Such care may not be necessary if competitive algorithms are used to implement lock acquisition.

Always-block performed poorly in both the analysis and elapsed-time measurements. Our figures indicate that choosing another strategy may lead to a significant performance improvement.

The elapsed-time measurements on a six-processor multiprocessor indicate that

- The simple fixed spin algorithm produces somewhat better performance than blocking, but somewhat worse performance than adaptive algorithms.
- The optimal on-line algorithm for a known distribution performs the best. However, the added benefit in elapsed times would not seem to justify requiring the programmer to declare the optimal spinning time for each lock or go to the effort of collecting the required data during a previous run.
- The optimal on-line approximation and the last-three-samples perform almost equally well. Both methods require a number of additional memory words for each lock to store history information for adjusting spin thresholds.
- The random-walk algorithm is the most efficient method among the three adaptive algorithms. It requires only one additional memory word for each lock and performs almost as well as the optimal on-line algorithm.

The implementation of the random-walk algorithm uses 16 spins as a unit to increment or decrement the spin threshold for each lock. It would require experiments to choose an appropriate unit for other multiprocessors and for other thread packages with different context-switch times.

It seems likely that on newer, faster machines, the advantages of competitive strategies may be even greater than those we observed. As mentioned earlier, faster machines rely on effective caching. Since context switches reduce the hit ratio of a cache, we expect that the performance impact of these algorithms will become even more pronounced.

Acknowledgements

We would like to thank Marc H. Brown for his help with our measurements. We would also like to thank Mike Burrows, Hendrik Goosen, Jim Horning and Tim Mann for their very helpful comments on previous drafts of this paper. Kai Li was supported in part by NSF grant CCR-8814265 and Digital Systems Research Center and External Research program.

References

[ABLL90] T.E. Anderson, B.N. Bershad, E.D. Lazowska, and H.M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. University of Washington Technical Report 90-04-02, October 1990.

[ALL89] T.E. Anderson, E.D. Lazowska, and H.M. Levy. The Performance Implication of Thread Management Alternatives for Shared-Memory Multiprocessors. In *ACM SIGMETRICS Conference on Measurement and Modeling Computer Systems*, 1989.

[And89] Thomas E. Anderson. The Performance Implication of Spin-Waiting Alternatives for Shared-Memory Multiprocessors. In *Proceedings of the 1989 International*

Conference on Parallel Processing, pages II:170–174, 1989.

[Dij65] E.W. Dijkstra. Solution of a Problem in Concurrent Programming Control. *Communications of the ACM*, 8(9):569, September 1965.

[GGK*83] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer – Designing an MIMD Shared Memory Parallel Computer. *IEEE Transactions on Computers*, C-32(2):175–189, February 1983.

[GT90] Gary Graunke and Shreekant Thakkar. Synchronization Algorithms for Shared-Memory Multiprocessors. *Computer*, 23(6), pp 60-69, June 1990.

[Her87] Maurice Herlihy. Impossibility and Universality Results for Wait-Free Synchronization. In *Seventh ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, August 1987.

[Her90] Maurice Herlihy. A Methodology for Implementing Highly Concurrent Data Structures. In *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 197–206, March 1990.

[KMMO89] A.R. Karlin, M.S. Manasse, L. McGeoch, and S. Owicki. Competitive Randomized Algorithms for Non-Uniform Problems. In *1st Annual ACM Symposium on Discrete Algorithms*, pages 301–309, 1989.

[KMRS88] A.R. Karlin, M.S. Manasse, L. Rudolph, and D.D. Sleator. Competitive Snoopy Caching. *Algorithmica*, 3(1):79–119, 1988.

[Knu66] Donald E. Knuth. Additional Comments on A Problem in Concurrent Program Control. *Communications of the ACM*, 9(5):321, May 1966.

[Lam87] Leslie Lamport. An Efficient Mutual Exclusion Algorithm. *ACM Transactions on Programming Languages and Systems*, 5(1), 1987.

[LA91] Beng-Hong Lim and Anant Agarwal, Waiting Algorithms for Synchronization in Large-Scale Multiprocessors. MIT VLSI Memo No. 91-632

[MB90] Jeffrey C. Mogul and Anita Borg, The Effect of Context Switches on Cache Performance. In Proc. *4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 75–84, Santa Clara, CA April, 1991

[MS87] P.R. McJones and G.F. Swart. Evolving the UNIX System Interface to Support Multithreaded Programs. Research Report 21, DEC Systems Research Center, September 1987.

[Oust82] John K. Ousterhout. Scheduling Techniques for Concurrent Systems. *Proc. 3rd International Conference on Distributed Computing Systems*, pp. 22-30, October 1982.

[Rov86] Paul Rovner. Extending Modula-2 to Build Large, Integrated Systems. *IEEE Software*, 3(6), November 1986.

[ST85] D.D. Sleator and R.E. Tarjan. Amortized Efficiency of List Update and Paging Rules. *Communications of the ACM*, 28(2):202–208, February 1985.

[TSS88] Charles Thacker, Lawrence Stewart, and Edwin Satterthwaite. Firefly: A Multiprocessor Workstation. *IEEE Transactions on Computers*, 37(8):909–920, August 1988.

[ZLE88] John Zahorjan, Edward D. Lazowska, and Derek L. Eager. Spinning Versus Blocking in Parallel Systems with Uncertainty. *Proc. Intern. Seminar Performance of Distributed and Parallel Systems*, North Holland, December 1988.

[ZLE89] John Zahorjan, Edward D. Lazowska, and Derek L. Eager. The Effect of Scheduling Discipline on Spin Overhead in Shared Memory Parallel Processors. Technical Report 89-07-03, University of Washington, July 1989. Also *IEEE Transactions on Parallel and Distributed Systems*, to appear.