# The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors

Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy

Department of Computer Science
University of Washington
Seattle WA 98195

## Abstract

Threads ("lightweight" processes) have become a common element of new languages and operating systems. This paper examines the performance implications of several data structure and algorithm alternatives for thread management in shared-memory multiprocessors. Both experimental measurements and analytical model projections are presented.

For applications with fine-grained parallelism, small differences in thread management are shown to have significant performance impact, often posing a tradeoff between throughput and latency. Per-processor data structures can be used to improve throughput, and in some circumstances to avoid locking, improving latency as well.

The method used by processors to queue for locks is also shown to affect performance significantly. Normal methods of critical resource waiting can substantially degrade performance with moderate numbers of waiting processors. We present an Ethernet-style backoff algorithm that largely eliminates this effect.

## 1. Introduction

The purpose of this paper is to study the performance implications of thread management alternatives for shared-memory multiprocessors.

In traditional operating systems, a process, consisting of a single address space and a single thread of control within that address space, is used to execute a program. Within the process, program execution entails initializing and maintaining a great deal of state information. Page tables, swap images, file descriptors, outstanding I/O requests, and saved register values are all kept on a per-program, and thus per-process, basis. The sheer volume of this information makes processes expensive to create and maintain.

Threads, or "lightweight" processes, separate the notion of execution from the rest of the definition of a process. A single thread executes a portion of a program, cooperating with other threads concurrently executing within the same address space. Like processes, every thread must have a separate program counter and stack of activation records, describing the state of its execution. However, much of what is normally kept on a per-process basis can be maintained in common for all threads executing in a single program, with dramatic reductions in overhead.

Thread packages have become a common element of new languages and operating systems for both uniprocessor and multiprocessor architectures. Mach [Accetta et al. 1986], Topaz [Thacker et al. 1988], Psyche [Scott et al. 1988], DYNIX [Sequent 1988], and several extensions to UNIX [Bach & Buroff 1984; Edler et al. 1988] are examples of operating systems that provide explicit support for concurrent or parallel execution of programs. Ada [Mundie & Fisher 1985], CSP [Hoare 1978], PRESTO [Bershad et al. 1988a], Mesa [Lampson & Redell 1980], Concurrent Euclid [Holt 1982], and Emerald [Jul et al. 1988] evidence equal interest within the language community.

On uniprocessors, threads are used as a program structuring aid or to overlap I/O with processing. The metric of goodness for these thread management implementations is simply processing cost per thread creation or context switch. No locking is needed inside thread routines, since only one routine can be executing at any one time.

Programs on multiprocessors use threads to exploit parallelism. The speedup achievable by any given application depends on the availability of thread management routines that provide low cost facilities that are not a serial bottleneck. In Sequent's DYNIX operating system, for example, applications must use normal UNIX-like processes for parallelism [Sequent 1988]. Since process creation in DYNIX takes over 25 milliseconds, only very coarse-grained parallelism can be exploited. As another example, the Topaz kernel provides relatively inexpensive thread creation and synchronization, but the routines are protected by a single lock [Thacker et al. 1988]. While this may be appropriate for architectures with small numbers of processors, as the number of processors increases, the single lock could limit speedups for applications with fine-grained parallelism.

Our initial experience in the area of high-performance thread packages was with PRESTO, an application-level runtime library that relies on the kernel only for processor allocation and memory management [Bershad et al. 1988a]. This work showed that there is an order of magnitude performance advantage to using threads instead of DYNIX processes for exploiting parallelism. Drawing on this experience, we implemented a thread package that is, in turn, an order of magnitude faster than PRESTO. This basic package was then modified to implement each alternative we wanted to explore.

One consequence of the speed of our basic thread package is that small changes in the organization of data structures and locks have a significant impact on performance. Often, the choice involves a tradeoff between latency and throughput. Per-processor data structures can sometimes be used to avoid locking, however, improving latency and throughput at the same time.

This material is based on work supported by the National Science Foundation (Grants No. CCR-8619663, CCR-8703049, and CCR-8700106), the Naval Ocean Systems Center, U S WEST Advanced Technologies, the Washington Technology Center, and Digital Equipment Corporation (the Systems Research Center and the External Research Program).

Another consequence of the speed of our thread package is that its performance depends noticeably on the algorithm used to queue for locks. Earlier, we studied the relative performance of spinning and blocking locks [Zahorjan et al. 1988]. In general, a thread that tries to acquire a lock that is already held can either spin ("busy-wait") until the lock is released, or relinquish the processor. However, within the thread management routines themselves, spinning is the only option. Thus, blocking at user level may require spinning in thread management routines. Spinning has a cost both to the processor awaiting the lock and to processors doing useful work. The degradation of other processors becomes substantial for moderate numbers of waiting processors, especially for small critical sections. We present an Ethernet-style backoff algorithm that largely eliminates this effect.

The following sections describe these issues in more detail. In Section 2 we present an abstraction of a thread package: its objects, resources, and operations. Section 3 outlines the strategies for thread management that we examined and presents measurements of their relative performance. Section 4 compares methods of queueing for locks. Section 5 combines these results in an analytical model. Section 6 summarizes our experiences.

## 2. An Abstract Thread Package

As noted in Section 1, threads gain efficiency by separating the notion of execution from the rest of the definition of a process. The data structures needed by each thread are a program counter, a stack, and a control block. (The control block contains state information needed for thread management. Through the control block, the thread can be put onto lists and other threads can synchronize with it.) Another important data structure is the ready queue, which lists threads that are ready to run. Lampson and Redell [1980] provide a good description of the functionality of a uniprocessor thread package.

Thread operations are shown in Table 2.1. Creating a thread can be viewed as calling a procedure, except that the callee can execute in parallel with the caller. In both cases, the caller specifies a place to begin executing and some number of arguments. In fact, thread creation and startup is semantically equivalent to an asynchronous procedure call.

Thread Creation
- Allocate and initialize a control block, saving the initial PC.
- Allocate a stack and copy in the thread's arguments.
- Place new thread on the ready queue.

Thread Startup
- Remove thread from ready queue and begin to execute it.

Thread Block (wait on blocking lock, condition variable, or message)
- Save register values and PC on the thread's stack.
- Place thread on the condition queue for the event.
- Look for a thread in the ready queue, and start or resume it.

Signal a Blocked Thread
- Remove thread from the condition queue.
- Place the thread on the ready queue.

Thread Resume
- Remove thread from the ready queue.
- Restore registers.
- Continue executing it from the saved PC.

Thread Finish
- Deallocate the stack and control block.
- Look for a thread in the ready queue, and start or resume it.

**Table 2.1: Thread operations**

As Table 2.1 shows, a program can create a thread even if there is no idle processor available to run it. Because the parallelism cannot be immediately exploited in this case, it might seem that the overhead of thread creation should be avoided. The program may run faster by creating the thread, however, if at some future time there will be an idle processor that can be used to execute the thread. This idea of creating parallelism for future use is very powerful. Unfortunately, in the above framework, its space cost is prohibitive. Each thread must be initially allocated a large amount of space for its stack, since it is expensive to dynamically expand the space if the thread later runs out of it. In Table 2.1, the thread is allocated space for a stack when it is created, but the space is largely wasted until the thread is actually started. Using virtual memory could remove the need to allocate physical memory to back the stack space until the thread begins to run; however, allocating extra virtual memory also is expensive.

An important optimization to Table 2.1, therefore, is to copy a thread's arguments into its control block when the thread is created. This way, the stack need not be allocated until thread startup; the arguments can be copied from the control block to the stack at that time. WorkCrews [Vandevoorde & Roberts 1988] and PRESTO [Bershad et al. 1988a] both take this approach.

Another important optimization is to store deallocated control blocks and stacks in free lists [Bershad et al. 1988a]. If these data structures were individually allocated out of the heap, thread overhead would include the cost of finding a free block of the correct size as well as possibly coalescing the block when it is returned to the heap. By using free lists, both allocation and deallocation can normally be simple list operations.

We begin our study by assuming these optimizations. For simplicity, we will focus on the effect of thread management alternatives on the performance of only a few thread operations: creation, startup, and finish. These operations manipulate each of the three shared data structures: the ready queue, the stack free list, and the control block free list. Most of the discussion applies as well to threads that block and resume.

## 3. Thread Management Alternatives

In a parallel environment, access to shared data structures must be serialized to ensure consistency and correctness. Our thread package uses spin locks for this purpose: when a processor tries to modify a data structure, it must first lock it to obtain exclusive access; if some other processor already holds the lock, the processor loops until the lock is released.

Locking implies dual concerns of latency and throughput [Kumar & Gonsalves 1977]. Latency is the cost of thread management under the best case assumption of no contention for locks. Throughput, on the other hand, is the rate at which threads can be created, started, and finished when there is contention. If part of thread management must be done serially, then no matter how many processors work on a problem, there will be some maximum rate of thread creation.

There are several ways of defining latency, with different implications for different types of applications. If an application keeps all of its processors continually busy, for instance by creating threads before they are needed, then any time spent in creating, starting, or finishing a thread is time that could have been spent doing other useful work. When a thread finishes, however, if there is no other work for the processor to do, the time spent deallocating the thread's data structures is unimportant. Instead, the relevant issues include how much a creating processor is delayed, since it has a thread to run, and how much time it takes for the created thread to begin running on a processor.

In the following subsections, we define five alternative thread management strategies, and describe some of the potential advantages and disadvantages of each approach. We then provide measurement and analytical comparisons of these alternatives.

## 3.1. Single lock: central data structures protected by one lock

The most obvious approach to thread management is to protect all data structures under a single lock. Once the lock is acquired by a processor, the processor is assured that it can modify any stored state. To perform a thread operation, a processor must first acquire the lock, then do what is needed to the shared data, and finally release the lock when done. In this way, only a single lock is needed per thread operation, but, since most of the thread management path is serialized, throughput is limited. In the typical scheme, idle processors loop checking the ready queue for work to do, causing useless contention for the ready queue lock; however, this can be avoided if idle processors check that the ready queue is not empty before acquiring the lock. (Ni and Wu [1985] present a different approach.)

## 3.2. Multiple locks: central data structures protected by separate locks

A somewhat more modular approach to locking is to separately protect each data structure with its own lock [Lampson & Redell 1980]. Each operation on the data structure can then be surrounded by a lock acquisition and release. For thread management, this involves separately locking each enqueue and dequeue operation on the ready queue, stack free list, and control block free list, the three shared data structures.

There is a basic tradeoff between latency and throughput in the choice between using a single lock or multiple locks in protecting shared data structures [Kumar & Gonsalves 1977]. Since less of the total thread activity is in a critical section, and since it is split among several locks, the maximum rate of thread creation is higher with multiple locks than with a single lock. There is a cost to this increased throughput, however: more lock accesses are needed, increasing latency.

## 3.3. Local freelist: per-processor free lists without locks

One way of avoiding locking is to maintain as much state as possible locally, with each processor. If each processor maintains its own free lists of control blocks and stacks, these need not be locked, since only one processor will access them. As before, there is a single shared ready queue whose accesses are locked.

The tradeoff between latency and throughput can be largely avoided by using local free lists. Since fewer lock acquisitions are needed per thread, latency is lower than with multiple locks, yet since only accesses to the ready queue are serialized, throughput is better.

Local free lists need to be balanced. Control blocks and stacks can migrate between free lists if the thread is created or started on one processor and finished on another. Thus, one free list can be empty, requiring the processor to obtain more space from the heap, while another free list has many entries. In the worst case, some processors only create and start threads (allocate structures), while other processors only finish them (deallocate structures). Without balancing, the deallocated structures are never re-used; a separate stack and control block are needed for every thread. In contrast, with a centralized free list, only as many are needed as there are active (created or started, but not finished) threads.

It is inexpensive, however, to balance free lists by using a global pool and a threshold $T$ on the maximum size of each list. When the size of a free list reaches the threshold, half the list can be returned to the global pool; when a free list empties, $T/2$ entries can be removed from the pool. The global pool must be locked, of course. For efficiency, it can be organized as a list of lists. The processing cost to balancing is thus one locked pool access amortized across at least $T/2$ free list accesses. Let $P$ be the number of processors. An application using balanced local free lists will use no more than $O(P \times T)$ more space than one using a central list; the worst case occurs when one processor's free list is empty while all other free lists are almost full.

Thus, local free lists trade space for time. This tradeoff is practical for control blocks. Utilization of the pool lock is at most $O(PR/T)$, where $R$ is the rate of thread creation on a single processor. To ensure that the pool lock is not a source of contention (which would inflate the overhead per free list access), we can set the threshold $T$ to be equal to $P$. Control blocks are relatively small objects (in our implementation, roughly 100 bytes); provided $P$ is not excessively large, using $100P$ bytes per processor is not onerous. If $P$ is large, then a tree of pools could be used to limit the cost to balancing to $O(P/\log P)$ bytes per processor.

The tradeoff is not practical for stacks, however. Stacks are at least two orders of magnitude larger than control blocks. Even if sufficient memory were available, using that memory entails processing costs for initializing page tables and increased cache miss rates that could easily overwhelm the advantage gained from decreased locking. Instead, we use single element stack free lists. In this way, stacks need be allocated from the global pool only when a processor blocks a thread and then starts up a different thread, and deallocated only when a processor finishes a thread and then resumes another thread.

## 3.4. Idle queue: a central queue of idle processors

None of the algorithms described so far exploit parallelism in thread creation. The creating processor allocates and initializes the control block; when it is done, the starting processor allocates and initializes the stack. The cost of thread creation could be reduced if some of the work was done by idle processors in parallel with the creating processor.

In addition to a central queue of threads, we can maintain a central queue of idle processors. When there is a backlog of ready threads, there is no point to attempting parallel thread creation since all processors are already doing useful work. When a processor becomes idle and there is no backlog, it pre-allocates a control block and stack, puts itself on the idle queue, and spins on a local flag waiting for work. Thread creation then dequeues the idle processor, initializes the pre-allocated control block and stack, and sets that processor's flag, indicating that it now has a thread that is ready to run. Instead of processors searching for work, work searches for processors.

In fact, this approach does not alter the essentially sequential nature of thread creation. The idle processor must first queue itself before the creating processor can dequeue it, which in turn must set the flag before the idle processor can start running the thread. The critical path between the beginning of thread creation and when the thread starts running is reduced by doing some of the work (allocating structures, acquiring a lock, enqueueing) before the critical path begins. Since this adds complexity, and there is no benefit in the absence of idle processors, the effect is to trade off reduced latency when there are idle processors for increased latency when all processors are busy. Maximum throughput should be unchanged since two locked queue operations are still needed per thread life cycle. Wagner et al. [1988] describe a different way of using of idle processors to avoid work during blocking and resuming.

## 3.5. Local readyq: per-processor ready queues

Once free lists are made local, the ready or idle queue lock can become a serial bottleneck as the rate of thread creation or the number of processors increases [Dritz & Boyle 1987]. One way of increasing throughput is to divide the load on a single lock among several locks. An application of this idea is to keep a ready queue per processor. In this way, enqueueing and dequeueing threads can occur in parallel, with each processor using a different queue. There is again a tradeoff between latency and throughput in the choice between using one or more ready queues.

Unlike the case of control block free lists, unlocked local ready queues are inefficient even if balanced through a global pool. Runnable threads are a scarce resource. An idle processor might have an empty queue, yet a ready thread that the processor could run is in some other processor's queue, while the global pool is empty. Performance can be arbitrarily bad in any scheme where a processor can be idle indefinitely while there is even one ready thread in some other queue. In the worst case, $P$ identical threads are created, but due to an imbalance, only $P - 1$ are started while one processor idles. The runtime would then be twice as long as with any of the centralized queueing strategies.

One simple way of avoiding indefinite idling is to lock each ready queue; each idle processor can then scan the ready queues for work, beginning with its own [Dritz & Boyle 1987]. If there is a ready thread, an idle processor will eventually find it. Processors can queue created threads locally, since balancing is achieved by idle processors. The worst case for this approach is when a single processor creates every thread, since that processor's queue would operate much as a central ready queue would, except that idle processors would have to waste time scanning for it. A simple way of avoiding this situation is for each processor to randomly choose a queue for each thread creation.

If each queue is equally likely to get a new ready thread, latency is bad when the number of runnable threads is near to the number of processors. There are two cases. Consider the cost of scheduling a thread onto a newly idle processor. If there are no ready threads, there is effectively no cost until a new thread is created. If there are ready but not running threads, any time spent finding a thread to run could have been spent running that thread. This time is small when there are many ready threads, because the idle processor will find the thread after scanning only a few queues; when there is only a single ready but not yet running thread, the processor will have to examine on average half of the queues in order to find it. The cost of scheduling a newly created thread onto an idle processor is similar: the thread will be found quickly if there are many idle processors and more slowly if there are only a few.

One reason to have a one-to-one correspondence between processors and ready queues is to maintain locality. Presumably, migrating a newly active or resumed thread has a cost, due to increased cache misses. On the other hand, threads can only be maintained locally if there is a large backlog of ready threads [Eager et al. 1986]. While there are some message passing applications where this holds, there is little reason to create a new thread if it will simply run on the same processor that created it. In any case, the cost of migration is certainly application-specific.

If maintaining locality is unimportant, there is a tradeoff between latency and throughput in choosing the number of queues [Ni & Wu 1985]. Up to some point, throughput is higher with more queues, but the number of queues that must be scanned to find work, and thus the latency, is also higher. We set the number of queues equal to the number of processors for all measurements.

## 3.6. Measurement results

To validate our intuitions about the relative merits of the alternative approaches, we implemented each on a Sequent Symmetry Model A shared-memory multiprocessor. All code was written in C and compiled with Sequent's standard compiler, with the exception of the locking and context switching code, which was programmed in assembler. Our Symmetry has twenty Intel 386 processors, a shared bus, and a write-through cache coherency protocol [Lovett & Thakkar 1988]. The Symmetry has a timer with microsecond resolution that was used for all measurements. Table 3.1 contains times for sample Symmetry operations.

| Operation | Runtime (μsec.) |
|---|---|
| Acquire and release a lock | 5.6 |
| Procedure call with no arguments | 3.6 |
| Each call argument | 1 |
| Iteration of null loop | 2.5 |

**Table 3.1: Runtimes for Symmetry operations (measured)**

For all measurements, free lists were "warm started": sufficient control blocks and stacks were pre-allocated for use by the benchmark. Our purpose was to measure the relative merits of each alternative, rather than the efficiency of the underlying memory management. The cache was not warm-started, but we ran each benchmark long enough for this effect to become insignificant.

Figure 3.1 is the principal performance comparison: it shows the elapsed time in seconds for each thread management alternative to create, start, and finish one million "null" threads, for varying numbers of processors. Initially, $P$ threads are created; each recursively creates a thread then finishes, allowing that processor to start up one of the waiting threads. The test terminates when each processor has executed $1M/P$ threads. For the multiple ready queue alternative, each newly created thread was added to a random queue to avoid biasing the results with the effect of locality. This test is not intended to be representative of a real parallel program, but it does expose the tradeoffs among the alternatives. (The one processor case shows the latency for a single thread in microseconds when there is no contention for locks.)

Figure 3.2 shows the inverse graph: the rate of thread creation (throughput) for each alternative, in units of 1000s per second.

Before examining the relative performance of the five alternatives, we note that each of them has quite good performance. Threads are only an order of magnitude more expensive than a procedure call, and 500 times less expensive than normal DYNIX process creation. Threads in PRESTO [Bershad et al. 1988a] cost 600 μsec. on the same Symmetry hardware, an order of magnitude worse than our threads although an order of magnitude better than DYNIX processes.

While PRESTO's speedup relative to DYNIX is due to using threads instead of processes, our speedup relative to PRESTO is due to attention to implementation details. We implemented PRESTO in C++; while this enhanced its ability to be modified [Bershad et al. 1988b], its C++ was first pre-processed into C, then compiled. This resulted in much less efficient code than could be achieved by direct coding in C. Another factor is that we stripped thread control blocks of all non-essential state, reducing the cost of initialization dramatically. We did not remove functionality: our thread package could be given PRESTO's user interface without sacrificing its performance.

Because our threads are inexpensive, the choice of alternatives has a large relative impact on both latency and throughput for applications with fine-grained parallelism. Specifically:

- Adding even a single lock acquisition into the thread management path can increase latency significantly. Locking each of the data structures separately results in a much higher latency than locking all data structures under the same lock. Using per-processor data structures to avoid locking is thus crucial to decreasing latency without sacrificing throughput.

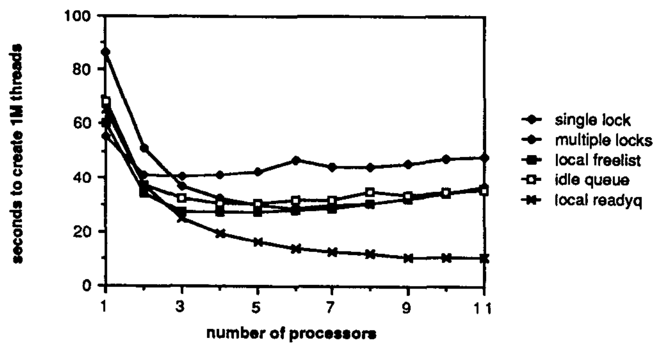

**Figure 3.1: Principal results for thread management – elapsed time to create, start and finish 1M null threads (measured)**

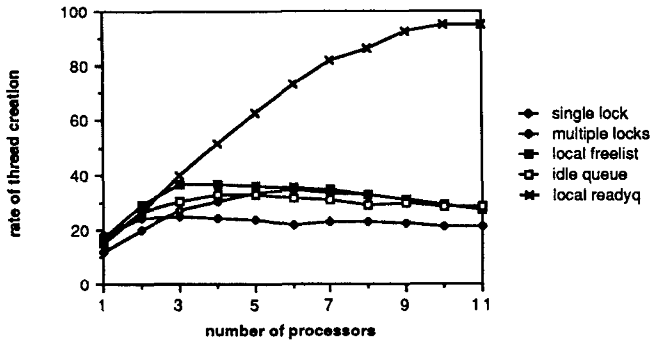

**Figure 3.2: Rate of thread creation, 1000s of threads per second (inverse of Figure 3.1)**

- Additional complexity results in a noticeable increase in latency. There are on the order of 100 instructions in the thread management path; adding even a few extra instructions impacts performance. For example, the idle queue strategy checks for idle processors on thread creation. If the idle queue is always empty, as in the measurements in Figure 3.1, it defaults to a normal ready queue. Even this simple a check markedly increases the cost of threads. This implies that thread management routines must be kept simple; enhancements that would otherwise seem plausible but add complexity are unlikely to work, since there is little computation to save, and it is easy to swamp the savings with increased overhead.
- A large portion of the thread management path is locked, since little work is required beyond manipulation of shared data. When all data is kept under a single lock, throughput is limited by contention for this lock. However, even with local free lists, the lock on the ready queue limits throughput to only a few concurrent thread operations. Only local ready queues can support high rates of thread creation.

When lock contention is not a problem, the bandwidth of the bus limits the thread creation rate. The throughput in Figure 3.2 levels out for the local ready queue alternative, even though there is no significant contention for locks. While the heavy bus demand per thread may be specific to the write-through cache protocol on the Symmetry, bus contention is likely to be a problem on any bus-structured shared-memory system.

In Figures 3.1 and 3.2, threads do no work except to create other threads. It is natural to ask whether the performance implications of the thread management alternatives would still be significant in the presence of user-mode computing. Figure 3.3 graphs thread creation rate as a function of the number of processors, when the amount of user work per thread averages 300 μsec, taken from a uniform distribution. This is representative of applications with fine-grained parallelism. Differences appear as the number of processors increases.

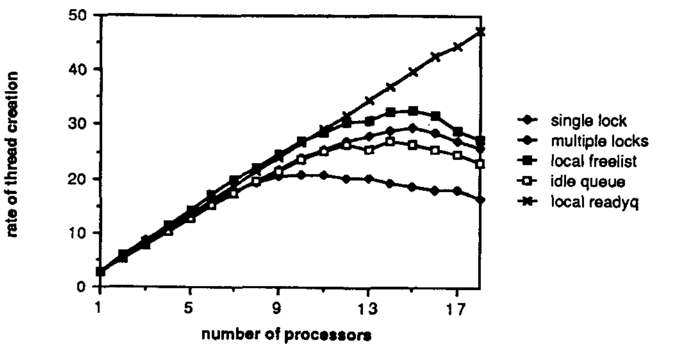

**Figure 3.3: Rate of thread creation, 1000s of threads per second, user work = 300 μsec. (measured)**

Figure 3.4 graphs thread cost in μsec. as a function of the number of runnable threads (parallelism). When there are fewer threads than processors, thread cost is taken to be the time to create and start running a new thread. The time to finish a thread is unimportant if the idling processor has no work to do. When there are as many or more runnable threads as processors, the cost is the sum of the time to create a thread plus the time to finish it and start a new thread. This difference in the definition of cost results in the jump in Figure 3.4 when the number of runnable threads reaches the number of processors. Note that the thread latency reported in Figure 3.1 with one processor corresponds closely to the latency reported in Figure 3.4 when there are more runnable threads than processors.

Thread cost was directly measured by taking timestamps before and after each thread was created and whenever a thread started or finished. Multiple creations were measured and averaged to improve accuracy. Creations and completions were synchronized to avoid measuring lock contention.

As expected, an idle queue is faster when there are idle processors, but slower when there are more runnable threads than processors. Thread creation is faster if an idle processor can be used to do work before the thread is created, but checking the idle queue incurs overhead even if it is not used. Whether a particular application will run faster with an idle queue depends on how much time it spends in each case.

The spike in the curve when using per-processor ready queues shows that finding a ready thread among many queues is expensive when the parallelism of the application is near to the number of processors, but the expense fades when more ready threads or more idle processors are available.

One area of further research is to examine hybrid thread management strategies to combine the advantages of some of the alternatives we have presented. For example, both central and per-processor ready queues could be used, by placing created threads in a local queue if the lock on the central queue is busy. As another example, a creating processor could probe randomly to find an idle processor, and if none were found, place the thread in a central queue. The drawback to any such approach is that complexity adds cost which may outweigh any benefits.

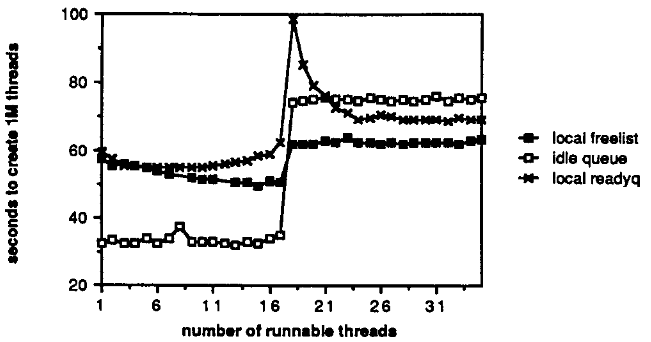

**Figure 3.4: Latency (μsec.) vs. number of runnable threads, 18 processors (measured)**

## 3.7. Analytical explanation of Figure 3.4

We now derive a formula that explains in detail the spike for the per-processor ready queue alternative in Figure 3.4. When there are idle processors, we need to know the time between the queueing of a ready thread and the dequeueing of that thread by an idle processor; when there is a backlog of ready threads, we need to know how long it takes a newly idle processor to find one.

Let $E(r,q)$ be the expected number of queues examined by a newly idle processor to find one of $r$ ready threads, which are randomly distributed among $q$ queues. Without loss of generality, let the queues be numbered from 1 to $q$, let threads be numbered from 1 to $r$, let $i_j$ be the queue containing the $j$th thread, and let the idle processor begin searching with queue 1. The idle processor must examine the number of queues equal to the lowest numbered non-empty queue. The number of ways of putting $r$ threads into $q$ queues is $q^r$.

$$E(r,q) = \frac{1}{q^r} \sum_{i_1,i_2,\cdots i_r=1}^{q} \textit{minimum of } (i_1, i_2, \cdots i_r)$$

We can separately sum when each $i_j$ is the minimum. When more than one thread is at the minimum, we count the value once in the sum for the least numbered thread. Thus, the value of $i_j$ is counted only if for all $k<j$, $i_k>i_j$, and for all $k>j$, $i_k \geq i_j$.

$$E(r,q) = \frac{1}{q^r} \left[ q + \sum_{j=1}^{r} \sum_{i=1}^{q-1} i(q-i+1)^{r-j}(q-i)^{j-1} \right] \quad (3.1)$$

By symmetry, Equation 3.1 also holds when there are more processors than runnable threads. Let $r$ be the number of idle processors, let $i_j$ be the queue currently scanned by the $j$th idle processor, and let the newly created thread be put into queue 1. Then the processor that actually dequeues the thread will have to look through $E(r,q)$ queues, after the thread is queued, to find it.

Figure 3.5 graphs Equation 3.1 for 18 processors. To compare to Figure 3.4, the x-axis is the number of runnable threads, rather than the number of ready but not running threads or the number of idle processors. Noting that part of the spike in Figure 3.4 is due to the difference in the measurements when there are idle processors or not, Figures 3.4 and 3.5 correspond well.

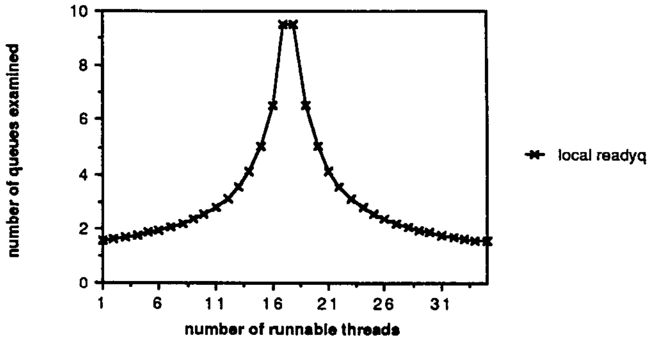

**Figure 3.5: Queues examined vs. number of runnable threads, 18 processors (Equation 3.1)**

The above analysis assumes that events occur one at a time. Since finding a ready thread among a number of queues can take a non-trivial amount of time, it is reasonable to consider what happens when another thread is created or another processor becomes idle during the interim. Suppose another thread is created before an idle processor finds one of the $r$ ready threads. Let $C$ be the cost of finding a thread in this situation. If the new thread is the one that is found, then $C$ is no better than if the new thread had been there all along. If a different thread is found, then $C$ is no worse than if the new thread is ignored. In other words, $E(r+1,q) \leq C \leq E(r,q)$. Similarly, if another processor becomes idle in the interim, provided $r \geq 2$, the combined cost for both processors to find threads is $E(r,q)+E(r-1,q)$, assuming the processors do not contend for the same queue, independent of which processor finds a ready thread first.

## 4. Spinlock Management Alternatives

If a processor finds a thread management lock busy, it must spin wait for the lock to be released. Since any other work the processor might do instead is also controlled by a lock, the processor does not have the option of doing other work while it is waiting.

At the user level, a thread does have a choice between spinning for a busy lock or blocking, relinquishing the processor to do useful work while the lock is busy. Since finding that work requires access to thread management data structures, however, blocking at the user level may result in spinning in a thread routine.

Spin-waiting has a hidden cost. Processors doing useful work may be slowed by processors that are merely waiting for a lock, due to bus contention. As a result, adding to the number of processors executing an application may in fact slow it down by increasing the average number of spinning processors. Worse, the more spinning processors, the more the processor holding the lock is slowed, increasing the effective size of the critical section, resulting in even more waiting processors.

Here we evaluate three different approaches to spin-waiting.

### 4.1. Hardware description

On the Symmetry Model A, each processor has its own cache; provided all of its memory references can be satisfied out of that cache, a processor's progress is independent of the activity of other processors. Whenever a processor reads data that is not in its cache, it must wait for the data to come from memory via the bus; with a write-through protocol, a processor may also have to wait for writes to be sent to memory. In both cases, the processor's progress can be slowed by bus contention.

The Symmetry has a basic test-and-set instruction, xchgb (exchange byte), that atomically reads a memory location and writes in a new value. The atomicity of the xchgb operation is enforced by the bus: a copy of the memory location is brought into the processor's cache, modified there, and then written back to memory. Any requests for that memory location in the interim are delayed until the processor is done modifying it [Lovett & Thakkar 1988].

The Sequent locking protocol is as follows: To lock, a processor exchanges in a 1. If the old value was a 0, it got the lock; if the value was a 1, the lock was already held by someone else, and the processor must try again. In either case, the value is 1 afterwards. The lock is released by exchanging in a 0; this allows some other processor to get a 0 back in exchange for a 1. There are several potential protocols for spin-waiting, which are described below.

### 4.2. Spin on xchgb

The simplest way to implement spin-waiting is for each processor to loop on the xchgb instruction until it succeeds. The drawback to this approach is that every xchgb instruction consumes bus resources, whether or not it succeeds [Sequent 1988]. A copy of the lock must be brought into the processor's cache; since the lock is written whether or not it is acquired, any copy of the lock in another cache is invalidated. As additional processors spin on the lock, the holder of the lock is slowed both because the bus is busier and because to free the lock it must contend with atomic operations of processors uselessly trying to acquire the lock.

### 4.3. Spin on memory read

An alternative would be for each processor to try to acquire the lock once; if this fails, the processor can spin reading the lock memory location. As long as the value is 1, the lock is still held. Looping on a read is done in the cache, avoiding bus traffic. When the lock is released, the cache copy will be invalidated; the spinning processor will see the value change to 0, and can then try to acquire the lock using an xchgb operation. Sequent's runtime library uses this implementation [Sequent 1988].

A problem arises when there are a number of processors waiting for a small critical section. When the lock is freed, every spinning processor's copy is invalidated, causing each processor to miss in turn. The first to try to acquire the lock succeeds. Any processor that reads the value before this occurs will see a 0 and will attempt to acquire the lock (and fail); any processor that reads the value afterwards will see a 1 and will return to looping in its cache. Unfortunately, each processor that does an unsuccessful xchgb operation invalidates all cache copies, forcing all processors that had seen a 1 to read miss again. After each such operation, virtually every spinning processor must contend for the bus, some still waiting to do an xchgb and some waiting for a read miss. Eventually, the last processor to have seen a 0 will attempt to acquire the lock and fail; each spinning processor can then read miss and quiesce, looping in its cache.

The performance of this algorithm, therefore, improves as the critical section gets longer, assuming that contention does not increase. After the lock is released and before quiescence, each spinning processor spends most of its time with a pending bus request; any normal bus request during this time will be correspondingly delayed. After quiescence, the spinning processors place no load on the bus, allowing the processor holding the lock to progress unhindered. With longer critical sections, the initial degradation is less significant. By contrast, spinning on the xchgb instruction degrades bus performance evenly throughout the critical section.

### 4.4. Ethernet-style backoff

The source of the difficulty is that there is a cost to attempting to acquire the lock. A generic solution to problems of this sort is to have each processor estimate its likelihood of success, and only try the lock when the probability is high. The estimate can be made from experience. The more times a processor has tried and failed, the more likely it is that many processors are spinning for the lock. When the lock is released, then, instead of every processor rushing to try to get it, each waits a period of time dependent on the number of past failures. If the lock is still free after this period, then the probability of success is high enough to try the lock. We used this algorithm for our measurements in Section 3.

The analogy with Ethernet is revealing. In the Ethernet protocol, a processor can start a network transmission in any time slot that the network is free [Metcalfe & Boggs 1976]. If two try to start transmitting in the same slot, both fail and must be retried later. To avoid further collisions, the length of time before retrying depends on the number of collisions encountered so far. In our case, when a number of processors simultaneously try to acquire a lock, one will succeed, but its progress will be slower than if there were no collisions.

The downside to Ethernet-style protocols is that they are unfair. A processor that has just arrived is more likely to acquire the lock (or network) than one who has been waiting, and failing, for some time. Spinning on a test-and-set instruction and spinning on a copy of the lock location are both probabilistically fair; each spinning processor has an equal likelihood of getting the lock, even though the possibility of indefinite starvation exists. Lock fairness is sometimes important to an application.

Another drawback of the backoff algorithm is that it takes longer for a spinning processor to acquire a newly free lock. The processor must check the lock value, delay, and check it again before trying the lock. Once the lock is acquired, however, the processor will proceed faster, relatively unimpaired by other spinning processors.

Even using this algorithm, there will be processor degradation when there are large numbers of spinning processors. When the lock is released, every spinning processor encounters a cache miss. After this initial miss, most processors delay locally until some other processor has acquired the lock, and then miss again to see that the lock has been acquired. With enough spinning processors, the bus can be saturated with these misses, slowing down the processor executing in the critical section.

These cache misses can be avoided. A processor can delay whenever it reads the lock value as busy. If the lock is not busy, the processor can immediately try to acquire it. Thus, spinning processors miss their cache every time the delay period expires, rather than every time the lock is released. This is analogous to the Ethernet notion of persistence [Metcalfe & Boggs 1976]. A result of this variation is an even greater delay between when a lock is released and when a spinning processor will acquire the lock. Nevertheless, this type of spin-waiting may be appropriate for systems without hardware-coherent private caches. In this case, spinning on a memory read until the lock is released is impractical since each read consumes bus resources; backoff adapts the frequency of reads to the number of waiting processors.

While most practical applications will not waste large numbers of processors, this can be a problem with idle processors polling a central or distributed ready queue. When a ready thread is queued, if each idle processor rushes to acquire the lock, bus saturation will result. Even if each idle processor delays after observing that a thread is queued, then makes sure that it is still queued, each idle processor will still perform a cache miss, hurting performance for large numbers of idle processors.

If idle processors are kept on a queue, this problem does not occur. Each idle processor spins on a local flag. When a thread is created, only one processor's flag is modified; every other processor continues spinning without even a cache miss. The performance advantage of having work look for processors instead of processors looking for work will therefore be more important in systems with large numbers of processors. This effect can be seen in Figure 3.4; the cost of the central ready queue is higher when there are only a few runnable threads, since there are more idle processors spin-waiting for work to appear in the ready queue.

## 4.5. Measurement results

Figure 4.1 shows the elapsed time to increment and test a shared counter in a critical section 1 million times, for each method of spin-waiting. Each processor executed a loop: wait for the lock, increment the counter, and release the lock. If spin-waiting did not slow the processor holding the lock, the elapsed time for twenty processors would be no more than for one.

The magnitude of this effect is striking. Both spinning on the xchgb instruction and spinning on the copy of the lock degrade processor performance badly for even a moderate number of spinning processors. For small critical sections, in either alternative, every spinning processor spends all of its time doing cache read misses or atomic xchgb operations, consuming bus resources as fast as possible. By contrast, the backoff algorithm results in only slight degradation for less than ten spin-waiting processors.

Figure 4.2 shows the effect of increasing the size of the critical section for each algorithm. In addition to incrementing a counter, the critical section contained varying amounts of other work. We then normalized the time for the counter to be cooperatively incremented by eight processors by the time for one processor. This measures relative processor speed. Again, if spin-waiting did not slow the processor holding the lock, one processor would not be faster than eight, and the relative processor speed would always be equal to 1. As expected, spinning on memory read degrades performance less as the size of the critical section grows, while spinning on the xchgb instruction degrades performance evenly throughout the critical section.

To test the tradeoff between processor degradation and the delay in acquiring a newly released lock, we measured the elapsed time for a number of processors to each increment a shared counter within a critical section. Once a processor acquired the lock and bumped the counter once, it was set to loop until all processors were done. This test is indicative of the cost of using a lock for barrier synchronization. Figure 4.3 shows the elapsized time divided by the number of processors. If there is no processor degradation or delay in acquiring the lock, the elapsed time to achieve the barrier should increase linearly with each additional processor; the normalized curve in Figure 4.3 should be flat.

Figure 4.3 shows that for small numbers of processors, spinning on the xchgb instruction is fastest, since a processor immediately acquires the lock when it is released. As more processors are added, however, this benefit is outweighed by the degradation of the processor holding the lock. The backoff algorithm shows a similar curve to spinning on a memory read, but for a different reason. Initially, many processors are queued for the lock; this leads spinning processors to guess large delay times. As more processors acquire the lock, there are fewer queued processors, and the delays become inappropriate.

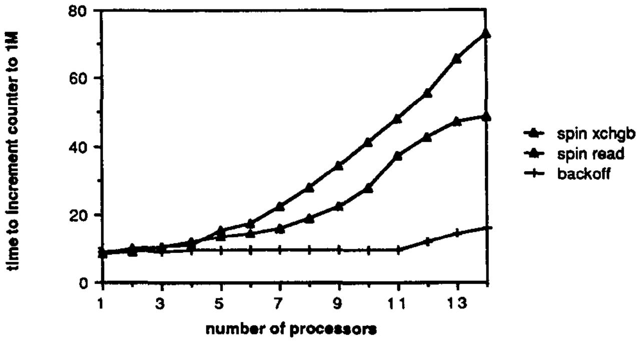

**Figure 4.1: Principal results for spin-waiting: elapsed time to increment a shared counter to 1,000,000 (measured)**

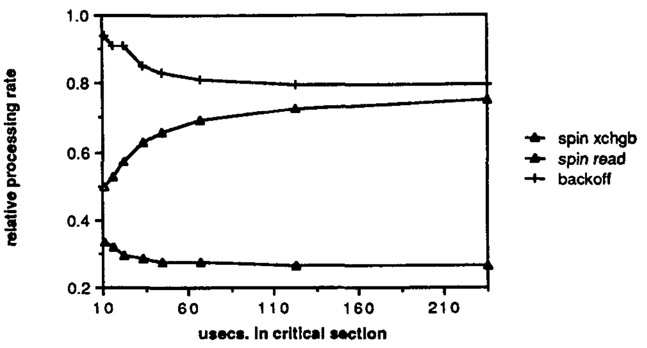

**Figure 4.2: Relative processor speed (8 processors to 1 processor) vs. critical section size (measured)**

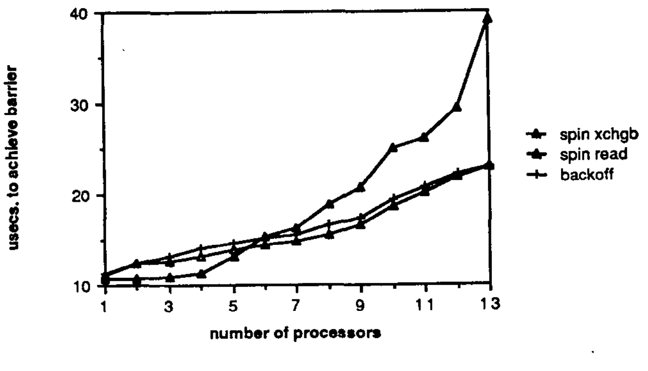

**Figure 4.3: Normalized time (μsec. per processor) to achieve barrier (measured)**

Processors doing work are slowed proportional to the number of times they access the bus. Thus, the results of these tests depend somewhat on the content of the critical section. However, since the purpose of a critical section is to serialize modifications to shared data, its code is likely to be bus intensive. Our measurements indicate that almost half of the bus service demand of thread management is due to the critical section. Further, thread management critical sections also tend to be small. For example, enqueueing or dequeueing a ready thread in a critical section both take less than 10 μsec., roughly the same as for Figure 4.1.

### 4.6. Implications for other systems

The Symmetry Model A has a write-through protocol: when a processor modifies a location, the value is written to memory and all old copies of the location in other caches are invalidated. There is a cost to spin-waiting, even in architectures with a write-back cache coherency protocol. In a write-back protocol, the value is stored in the cache and later written to memory when the cache block is replaced. There are two major approaches to keeping other caches consistent with the new value: all old copies in other caches can either be invalidated or updated with the new value (distributed-write) [Archibald & Baer 1986].

In the case of an invalidation-based write-back protocol, the spin-waiting alternatives have much the same effect as with write-through. If processors spin on the atomic test-and-set operation, the valid copy of the lock bounces from cache to cache, consuming bus resources. Provided more than one processor is spin-waiting, when one processor tries the lock, it invalidates every other cache copy, requiring the lock value to be copied to the cache of the next processor to try the lock. Spinning on a memory read does not solve this problem, since the cache copy of a looping processor is still invalidated, resulting in a cache miss, by each successive processor trying to acquire the lock. The Sequent Symmetry Model B, the successor to the architecture we used for our measurements, uses such a protocol.

The performance with distributed write-back is better, but it does not eliminate the problem. When a processor performs an atomic operation, every cache with an old copy is updated with the new value. If processors spin on the atomic operation, the bus can be saturated doing these updates. If processors spin on the memory read, however, each cache is kept up-to-date, eliminating the cascade of cache misses as each spinning processor tries to acquire the lock. The rush of processors to try the lock when it is first released still results in some bus traffic for distributing each update, but quiescence will occur faster. Since the backoff algorithm reduces the number of lock attempts, it reduces the bus load due to spinning even further.

A hardware mechanism for queueing processors without consuming bus resources would also solve this problem. In fact, the Symmetry has such a mechanism, but it is less than completely useful. While one processor is performing an atomic operation, any other processor attempting to access that memory location is delayed before using the bus [Sequent 1988]. Unfortunately, only single instructions can be made atomic; it is rare in practice to be able to complete a critical section in one instruction.

## 5. Analytical Results

We developed a queueing network model of our thread package to demonstrate that the combination of processor degradation due to bus contention and the effect of lock contention can account for our measurements. We used the validated model to project the performance of our package under varying conditions.

Our model is hierarchical. The low level model represents the effect of bus contention on processor speed. The high level model represents the effect of lock contention on throughput and response time. Since processor speed affects the amount of lock contention and the number of spinning processors affects bus contention and thus processor speed, we iterate between levels to convergence. We describe the two sub-models in more detail below.

### 5.1. Modelling bus contention

In the low level model, we represent each processor as a customer in a closed queueing network. The network has two service centers: a queueing center for the bus and a delay center for non-bus activity. Each processor spends some of its time referencing memory through the bus and thus contending with other processors also using the bus, and some of its time processing out of its cache, independent of the activity of other processors. Processor speed is degraded by the percentage of time spent queueing, but not in service, at the bus.

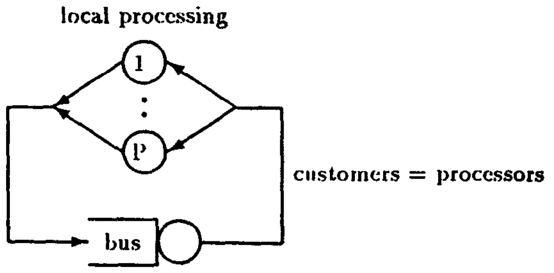

**Diagram 5.1: Low level model of bus contention**

This model is an approximation of the real bus mechanism, which is considerably more complex [Lovett & Thakkar 1988]. At moderate loads, our model will be pessimistic by predicting more contention than is actually experienced. Because of the regularity of the time each processor spends computing between accesses to the bus, if two processors collide at the bus, they are unlikely to collide at their next visit. Our model assumes that arrivals are more nearly independent.

There are three components to bus utilization. A processor can be executing user code, thread management code, or spin-waiting, each with different service demands on the bus. Given these service demands and the ratio of time each processor spends in each type of activity, we determine the aggregate service demands at the bus and at the delay center and use these aggregate demands to solve the model.

Since it is difficult to analytically determine the bus demand of a section of code, we determine a portion of it inductively from measurements. We provide each processor with its own copy of all data structures; we then run the code in parallel on each processor. Since there is no shared data, there can be no contention for software resources; any delay experienced by a processor relative to when it is running the code by itself must be due to contention for hardware resources, such as for memory or the bus. We then match a curve from our model of the bus to the measured curve and use the result as the service demand for that section of code. The curves matched well in practice, deviating only at moderate loads, as expected.

Since bus contention may disproportionately impact the critical section execution time, affecting lock contention in the high level model, we used this approach separately for the critical section and non-critical section code within thread management. The critical section code turns out to account for much of the bus demand of thread management.

Even though it could affect bus usage, we did not include in our model the effect of different numbers of processors on cache hit ratios. When a processor writes a location, the Symmetry updates both memory and that processor's cache. As a result, on a single processor, data that is both written and read will tend to stay in the cache, avoiding cache misses. When multiple processors read and write shared data, the cache copies of the data will be repeatedly invalidated as different processors update it, resulting in more cache misses than in the single processor case. Our model therefore underestimates bus demand, making it optimistic, especially as the bus nears saturation.

The bus demand of spinning processors was also determined inductively. $P$ processors were set to run the critical section with separate copies of the data structures; by the experiment described above, we know the bus service demand of these processors. $Q$ processors were set to run a shared copy of the critical section; one of these processors has the normal bus service demand, and $Q-1$ spin-wait. By measuring the processor degradation of the $P$ copies, we can determine the aggregate bus demand of the $Q-1$ spinning processors. A two class model was used, one class representing processors executing critical sections and one representing spinning processors. Only the response time of the processors executing the critical section is important.

The bus demand, at least for the backoff algorithm, is linear with the number of processors. While there is no *a priori* reason for this, it intuitively makes sense. The effect of adding a spinning processor with the backoff algorithm is to add two cache misses per execution of the critical section. The bus demand of other processors is relatively unaffected. While this invariance would also hold for the spin on xchgb algorithm, it is less true when processors spin on memory reads, because the cascade of cache misses is longer for every processor when more processors are spinning. Note that the graphs in Section 4 could be used to infer the bus demand of spinning processors. We did not choose this approach because there is a correlation between when the processor holding the lock and when the processors spinning on the lock use the bus. The curve for the backoff algorithm in Figure 4.1, e.g., is similar to that of an optimistic asymptotic bound.

## 5.2. Modelling lock contention

In the high level model, we represent each lock in the thread management path by a separate queueing center. Processing time spent not holding a lock is modelled as a delay center. Service demands were directly measured, then the part of each service demand due to bus accesses was inflated by the bus response time of the low level model. As in the low level model, each processor is represented as a single customer in a closed class. By solving this model, we can determine the average amount of time each processor spends spin-waiting for a lock versus executing thread operations or user code. This ratio is then used as an input to the low level model. (Note that it is a simple matter to add queueing centers if the application-level code does further locking.)

If the time between thread operations is deterministic, our model is pessimistic at moderate loads. As for the bus, if two processors collide at a lock, the effect of deterministic processing times is to reduce the likelihood that they will collide at the next visit. Figure 3.2 shows this effect. The curves are similar in shape to asymptotic optimistic bounds, since the processing time to do each thread operation is deterministic. Figure 3.3 does not show this effect, since the user computation for each thread was randomly chosen from a uniform distribution.

Our model does not explicitly represent an application's distribution of parallelism, although Figure 3.4 shows that this affects performance. We chose not to include this in our model since the distribution and the effect of lock queueing delay on that distribution are almost always application-dependent.

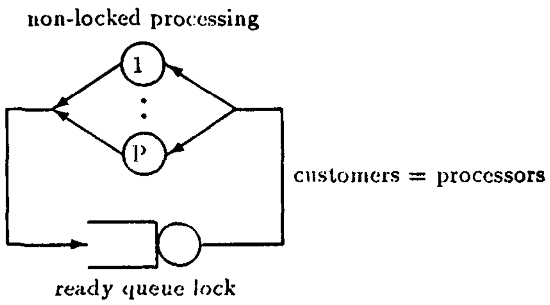

**Diagram 5.2: High level model of lock contention for the local freelist alternative**

Given the distribution, the model could be evaluated separately for each population of threads; these separate evaluations could then be averaged, weighted by the proportion of time for that population. The population of the high level model should be the minimum between the number of processors and the number of threads, reflecting the number of active processors. The population of the low level model should be set similarly, except that since idle processors consume bus resources, a second class should be added to represent them.

This method of separate evaluations ignores the fact that lock contention can only occur when the parallelism is being incremented or decremented; we believe that any distortion introduced by the adaptive nature of the mechanism will be outweighed by the effects of lock and bus contention. Ni and Wu [1985] also discuss this issue.

## 5.3. Comparison with measured results, and projections

Figure 5.1 compares our model results with our measurement results previously reported in Figure 3.3. We modelled two alternatives: per-processor ready queues (local readyq) and per-processor free lists with a central ready queue (local freelist). Our model agrees well with the measurements, within 5% except for the central ready queue with 18 processors. The model predicts the shape of the curve, but is somewhat optimistic; this appears to be due to underestimating the bus demand, which is important in determining the effective size of the critical section. The model does capture the difference between the alternatives.

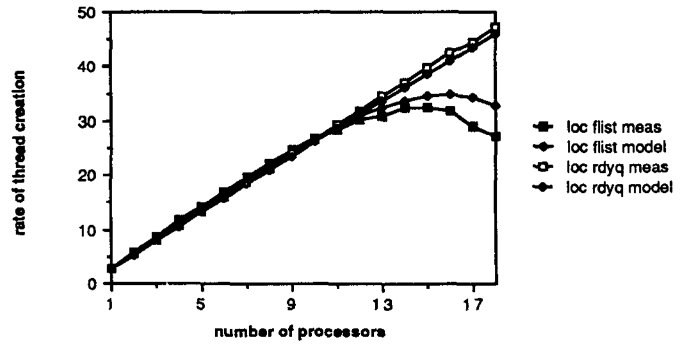

**Figure 5.1: Comparison of analytic and measured results from Figure 3.3**

Having validated our model, we used it to investigate the effect of varying key parameters. Figure 5.2 shows throughput with 20 processors as a function of the amount of user computation per thread. As we would expect, as an application uses finer-grained parallelism (smaller amounts of computation per thread), the central lock on the ready queue becomes a bottleneck. For sufficiently coarse-grained parallelism, the performance of the thread package ceases to matter. In the limit, even DYNIX processes could be used.

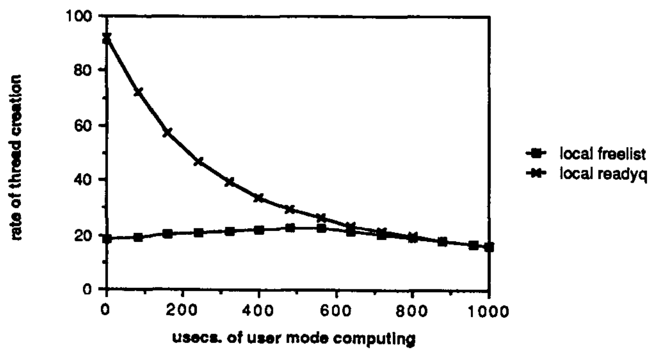

**Figure 5.2: Thread creation rate vs. μsec. of user computation per thread, 20 processors, bus load = 5% (analytic)**

Contention for the bus can also reduce the difference between the alternatives. Figure 5.3 shows throughput as a function of the percentage usage of the bus by each thread. As the bus usage increases, the bus limits the throughput with local ready queues, but it also limits the throughput with the central ready queue, since bus contention inflates the critical section time.

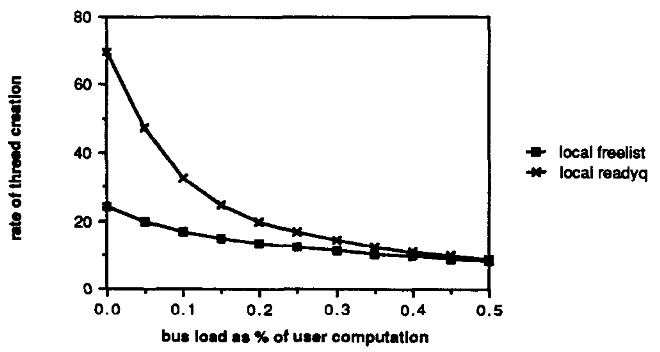

**Figure 5.3: Thread creation rate vs. bus load, user work = 200 μsec., 20 processors (analytic)**

On the other hand, the central ready queue lock can again limit throughput even for more coarsely-grained parallelism, given a sufficient number of processors. Figure 5.4 shows the throughput as a function of the number of processors when threads each compute for 2 milliseconds. The sharp dropoff for the central ready queue alternative shows the inherent instability of a system where spinning processors consume resources.

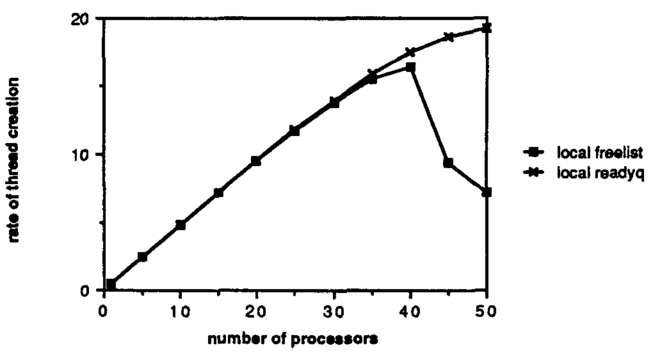

**Figure 5.4: Thread creation rate vs. number of processors, user work = 2 msec. (analytic)**

## 6. Conclusions

Threads have become a common element of new languages and operating systems. Efficient thread management is critical to achieving good performance from parallel applications. We have studied the performance implications of several thread management and locking alternatives. We showed that:

- It is possible to implement a fast thread package. Simplicity is crucial for this.
- For fine-grained parallelism, small changes in data structures and locking have a large effect on both latency and throughput.
- Per-processor data structures can be used to improve throughput; if a resource is not scarce, localizing data can avoid locking, improving latency as well.
- Spin-waiting can delay not only the processor waiting for a lock, but other processors doing work. This appears to be independent of the cache coherency protocol.
- An Ethernet-style backoff algorithm can reduce the cost of spin-waiting.
- A simple queueing model can accurately predict the effect of a combination of factors on the performance of shared-memory multiprocessors.

An area of future research is to determine the extent to which our results, developed in the context of thread management systems, also apply to application programs that exploit fine-grained parallelism on shared-memory multiprocessors.

## Acknowledgements

We would like to thank Dave Wagner for suggesting that an Ethernet-style algorithm might solve the spin-waiting problem.

## References

[Accetta et al. 1986]
M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation For UNIX Development. *Proc. Summer 1986 USENIX Technical Conference and Exhibition*, June 1986, pp. 93-112.

[Archibald & Baer 1986]
J. Archibald and J.L. Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, vol. 4, no. 4, Nov. 1986.

[Bach & Buroff 1984]
M.J. Bach and S.J. Buroff. Multiprocessor UNIX Operating Systems. *AT&T Bell Laboratories Technical Journal*, vol. 63, no. 8, Oct. 1984, pp. 1733-1749.

[Bershad et al. 1988a]
Brian Bershad, Edward Lazowska, and Henry Levy. PRESTO: A System for Object-Oriented Parallel Programming. *Software: Practice and Experience*, vol. 18, no. 8, Aug. 1988, pp. 713-732.

[Bershad et al. 1988b]
Brian Bershad, Edward Lazowska, Henry Levy, and David Wagner. An Open Environment for Building Parallel Programming Systems. *Proc. ACM/SIGPLAN PPEALS 1988*, pp. 1-9.

[Dritz & Boyle 1987]
Kenneth W. Dritz and James M. Boyle. Beyond "Speedup": Performance Analysis of Parallel Programs. Technical Report ANL-87-7, Mathematics and Computer Science Division, Argonne National Laboratory, Feb. 1987.

[Eager et al. 1986]
Derek Eager, Edward Lazowska, and John Zahorjan. Adaptive Load Sharing in Homogeneous Distributed Systems. *IEEE Transactions on Software Engineering*, vol. 12, no. 5, May 1986, pp. 662-675.

[Edler et al. 1988]
Jan Edler, Jim Lipkis, and Edith Schonberg. Process Management for Highly Parallel UNIX Systems. Ultracomputer Note #136, April 1988.

[Hoare 1978]
C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, vol. 21, no. 8, Aug. 1978, pp. 666-677.

[Holt 1982]
R. Holt. A Short Introduction to Concurrent Euclid. *SIGPLAN Notices*, vol. 17, May 1982, pp. 60-79.

[Jul et al. 1988]
Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, vol. 6, no. 1, Feb. 1988, pp. 109-133.

[Kumar & Gonsalves 1977]
B. Kumar and Timothy Gonsalves. Modelling and Analysis of Distributed Software Systems. *Proc. 7th ACM Symposium on Operating Systems Principles*, Dec. 1977, pp. 2-8.

[Lampson & Redell 1980]
B.W. Lampson and D.D. Redell. Experiences with Processes and Monitors in Mesa. *Communications of the ACM*, vol. 23, no. 2, Feb. 1980, pp. 104-117.

[Lazowska et al. 1984]
Edward Lazowska, John Zahorjan, G. Scott Graham, and Kenneth Sevcik. Quantitative System Performance. Prentice-Hall, 1984.

[Lovett & Thakkar 1988]
Tom Lovett and Shreekant Thakkar. The Symmetry Multiprocessor System. *Proc. 1988 International Conference on Parallel Processing*, pp. 303-310.

[Metcalfe & Boggs 1976]
Robert Metcalfe and David Boggs. Ethernet: Distributed Packet Switching for Local Computer Networks. *Communications of the ACM*, vol. 19, no. 7, July 1976, pp. 395-404.

[Mundie & Fisher 1985]
D.A. Mundie and D.A. Fisher. Parallel Processing in Ada. *IEEE Computer*, Aug. 1985, pp. 20-25.

[Ni & Wu 1985]
Lionel Ni and Ching-Fern Wu. Design Trade-offs for Process Scheduling in Tightly Coupled Multiprocessor Systems. *Proc. 1985 International Conference on Parallel Processing*, pp. 63-70.

[Scott et al. 1988]
Michael Scott, Thomas LeBlanc, and Brian Marsh. Design Rationale for Psyche, a General Purpose Multiprocessor Operating System. *Proc. 1988 International Conference on Parallel Processing*, August, 1988.

[Sequent 1988]
Sequent Computer Systems, Inc. Symmetry Technical Summary.

[Thacker et al. 1988]
Charles Thacker, Lawrence Stewart, and Edward Satterthwaite Jr. Firefly: A Multiprocessor Workstation. IEEE Transactions on Computers, vol. 37, no. 8, Aug. 1988, pp. 909-920.

[Vandevoorde & Roberts 1988]
Mark Vandevoorde and Eric Roberts. WorkCrews: An Abstraction for Controlling Parallelism. Digital Equipment Corporation Systems Research Center, 1988.

[Wagner et al. 1989]
David Wagner, Edward Lazowska, and Brian Bershad. Techniques for Efficient Shared-Memory Parallel Simulation. *Proc. Performance '89 / ACM SIGMETRICS 1989*.

[Zahorjan et al. 1988]
John Zahorjan, Edward Lazowska, and Derek Eager. Spinning Versus Blocking in Parallel Systems with Uncertainty. *Proc. International Seminar on the Performance of Distributed and Parallel Systems*, North Holland, Dec. 1988.