ThreadMon
A Tool for Monitoring Multithreaded Program Performance

Bryan M. Cantrill and Thomas W. Doeppner Jr.†
Department of Computer Science
Brown University
Providence, RI 02912-1910

Abstract

This paper describes ThreadMon, a monitoring tool for improving the performance of multithreaded programs. Traditional tools, e.g., gprof, thread analyzer, etc., present overall indications of performance, such as the total number of calls to each procedure, the time spent in each function, who called whom how often, etc. This information is not correlated with the passage of time. For time-critical programs, however, it is crucial to relate performance data to the events to which the program is responding: it is not enough to know *where* a program has spent its time, we must know *when* the program spent that time. In addition, to take full advantage of a multiprocessor, we need to know *when* the program is spending its time on *which* processors. Rather than present such information after the program has executed, we present it while the program is running, thus making it easy to relate our interactions with the subject program to the performance data being displayed.

1 Introduction

Even seemingly simple multithreaded programs may have performance problems whose causes are far from obvious. A program employing a number of independent threads may show no speedup when moved from a uniprocessor to a multiprocessor. One's carefully designed synchronization strategy may inexplicably result in miserable performance. Programs that run quite well most of the time may behave poorly at key moments.

Determining the causes of such problems using traditional tools can be challenging. Most tools (such as gprof and thread analyzer) depend on the availability of source code. However, while programmer-supplied code might appear faultless, library code, for which no source code is available, might make one's threads perform unanticipated (and expensive) actions. User and kernel schedulers might make conflicting decisions about which threads should be running.

Traditional tools present overall indications of performance, such as the total number of calls to each procedure, the time spent in each function, who called whom how often, etc. This information is not correlated with the passage of time. For time-critical programs, however, it is crucial to relate performance data to the events to which the program is responding: it is not enough to know *where* a program has spent its time, we must know *when* the program spent that time. In addition, to take full advantage of a multiprocessor, we need to know *when* the program is spending its time on *which* processors. Rather than present such information after the program has executed, we present it while the program is running, thus making it easy to relate our interactions with the subject program to the performance data being displayed.

†This work was supported by a grant from Sun Microsystems.

This paper describes ThreadMon, a monitoring tool for improving the performance of multithreaded programs. We describe the design of the tool, explain what we have learned from its use, and discuss a number of areas in which it has proved useful:

1) *Bottleneck analysis*: concurrent programs consist of a number of threads, each executing instructions independently and competing for various resources. Contention for these resources hinders performance—thus its minimization is an important goal. By interposing itself between the application and the threads package, ThreadMon can monitor a program's resource usage and display the extent of contention, not only for individual resources but for aggregates of resources. Compounding this resource-contention problem is that many library routines cause contention for resources that the application programmer may not even know exist. Our tool identifies and shows the conflicts for these resources, providing further valuable information to the programmer.
2) *Processor-utilization analysis*: an important concern to the user of a multiprocessor workstation is whether all processors are being effectively utilized. If so, could adding more processors yield performance gains? By showing what the program's threads and the workstation's processors are doing, ThreadMon gives the programmer sufficient information to handle these concerns: it does not solve performance problems, but points out that there are problems and provides feedback on the effectiveness of the programmer's solutions.
3) *Studying the effectiveness of two-level threads-implementation strategies*: Most thread packages provide simple, easy-to-use threads abstractions to the application programmer. Hidden behind many of these packages, however, is a two-level implementation model (also known as the *many-to-many model*) in which the user-level library schedules *user threads* on *kernel threads* and the kernel schedules *kernel threads* on *processors*. Potential programmer concerns when using this model include insuring adequate concurrency (e.g., making certain that threads can execute when they are ready and processors are available) and minimizing overhead in managing user and kernel threads. We demonstrate that without knowledge of both the implementation model and its runtime behavior with respect to one's application, programmers can unknowingly encounter performance problems. ThreadMon is being used to help the programmer discover these problems and develop tactics for overcoming them. We discuss what we have learned about the two-level implementation model and compare it with other implementation models, such as Scheduler Activations.

2 Implementation Models

The performance of a multithreaded program is strongly dependent on the underlying implementation of the threads package. Though for many applications the programmer need not be concerned with the implementation strategy of the threads package, for some applications, particularly compute-intensive applications on multiprocessors, the programmer must take into account various crucial aspects of the threads package. In this section we summarize the commonly used implementation strategies in preparation for our subsequent discussion on the impact of the strategy on the application.

2.1 Many-to-one Model

For kernels that do not support multiple threads of control, multithreading can be implemented entirely as a user-level library. These libraries, without the kernel's knowledge, schedule multiple threads of control onto the process's single kernel thread. Thus, just as a uniprocessor provides the illusion of parallelism by multiplexing multiple processes on a single CPU, user-level threads packages provide the illusion of parallelism by multiplexing multiple user threads on a single kernel thread; this is referred to as the *many-to-one* model [Kleiman et al. 1996]. There are several advantages to this model:

- *Cheap synchronization*. When a user thread wishes to perform synchronization, the user-level thread library checks to see if the thread needs to block. If it does, then the library enqueues the thread on the synchronization primitive, dequeues a user thread from the library's run queue, and switches the active thread. If it does not need to block, then the active thread continues to run. No system calls are required in either case.
- *Cheap thread creation*. To create a new thread, the threads library need only create a context (i.e. a stack and registers) for the new thread and enqueue it in the user-level run queue.
- *Resource efficiency*. Kernel memory isn't wasted on a stack for each user thread. This allows as many threads as virtual memory permits.
- *Portability*. Because user-level threads packages are implemented entirely with standard UNIX and POSIX library calls (e.g. with *getcontext* and *setcontext*), they are often quite portable.

However, the many-to-one model does not come without a price. Specifically:

- *Single-threaded OS interface*. Since there is only one kernel thread, if a user thread executes a blocking system call, the entire process blocks, since no other user thread can execute until the kernel thread (which is blocked in the system call) becomes available. While it adds significantly to implementation complexity, the library can circumvent this problem where nonblocking variants of system calls exist [Doeppner 1987].
- *No parallelism*. Multithreaded programs under the many-to-one model will run no faster on multiprocessors than they run on uniprocessors. The single kernel thread acts as a bottleneck, preventing optimal use of the multiprocessor.

Despite substantial disadvantages, the relative ease of implementation of many-to-one threads packages has made it the most popular model to date. For example, the current implementations of Netscape and Java achieve their multithreading strictly through user-level threads packages.

2.2 One-to-One Model

An obvious alternative to the many-to-one model is that every user thread have its own kernel thread (i.e., that there be a *one-to-one correspondence* between user threads and kernel threads). This provides several advantages:

- *Scalable parallelism*. Because each kernel thread is actually a different kernel-schedulable entity, multiple threads can run concurrently on different processors. Thus, multithreaded programs written under the one-to-one model can achieve significant speedups when migrated from uniprocessors to multiprocessors.
- *Multithreaded OS interface*. Unlike the many-to-one model, threads blocking in the kernel do not impede process progress under the one-to-one model. When one user thread and its kernel thread block, the other user threads can continue to execute since their kernel threads are unaffected.

While the one-to-one model can yield a major performance win, it too is not without its costs. Most of the benefits of the many-to-one model do not carry over to the one-to-one model:

- *Expensive synchronization*. Because kernel threads require kernel involvement to be descheduled, kernel-thread synchronization will require a system call if the lock is not immediately acquired.† Estimates vary, but if a trap is required, synchronization will be from three to ten times more costly than for the many-to-one case [Powell et al. 1991, Vahalia 1996].

- *Expensive creation*. Under the one-to-one model, every thread creation requires explicit kernel involvement and consumes kernel resources. The difference in creation cost depends on the specific implementation, but creating a kernel thread is generally between three and ten times more expensive than creating a user thread [Vahalia 1996].
- *Resource inefficiency*. Every thread created by the user requires kernel memory for a stack, as well as some sort of kernel data structure to keep track of it. Many parts of many kernels cannot be paged out to disk; the presence of kernel threads is likely to displace physical memory for applications.

2.3 Many-to-Many Model

In an attempt to combine these two models, some operating systems, notably Mach 3.0 [Golub et al. 1990], SVR4/MP, Solaris 2.x[Powell et al. 1991], and Digital UNIX 4.0, give the programmer both user-level threads and kernel threads to the programmer. User-level threads are multiplexed on top of kernel-level threads, which in turn are scheduled on top of processors. The kernel knows only about the kernel-level threads; it does not know of the multiplexing performed by the user-level scheduler. Due to the many-to-many relationship between user threads and kernel threads, this is called the many-to-many model [Kleiman et al. 1996] (it is also referred to as the two-level model [Catanzaro 1994], the split model [Vahalia 1996] and the LWP model). By taking a hybrid approach, this model aims to combine the advantages of both the many-to-one model and the one-to-one model, while minimizing these models' disadvantages.

The major advantage of the many-to-many model is that large numbers of threads can be supported relatively cheaply. As with the many-to-one model, the creation of a user thread does not necessarily require the (relatively expensive) creation of a kernel thread. Thus one can create a large number of user threads, but have the overhead of creating only a small number of kernel threads. Synchronization can also be inexpensive: the implementation of synchronization primitives involves primarily user-level code. A user thread that must block on a synchronization primitive (such as a mutex) is queued on a wait queue and the underlying kernel thread finds and runs another user thread on the user-level run queue. Only if no runnable user thread is available does the kernel thread make a system call and block (or *park*) in the kernel. Thus the cost of a context switch from one thread to another can be no worse than the cost of a few subroutine calls—a system call is often not necessary.

User threads in the many-to-many model normally "float" among kernel threads—they may run on whatever kernel thread is available when they become runnable. However, in some cases it may be necessary to associate a user thread permanently with a kernel thread, i.e., to *bind* the user thread to the kernel thread. Such bound threads behave as threads do in the one-to-one model—their creation requires the creation of a kernel thread, and synchronization operations requires system calls (to park the kernel thread in the kernel when the user thread is blocked and to unpark it when the user thread is released). Deciding when to use bound threads is an issue we discuss extensively in this paper.

---

† This system call can sometimes be avoided on a multiprocessor by spinning for some small period before giving up and trapping.

2.4 Scheduler Activations

The many-to-many model employs two schedulers: one in the kernel and one in the user threads library. It is not immediately obvious how the kernel scheduler can cooperate with the user scheduler. For example, say the user scheduler has a high-priority thread to schedule, so it preempts the execution of a lower-priority thread, reassigning its kernel thread to the high-priority user thread. But, at the same time, the kernel scheduler decides that the kernel thread's time slice has expired and reassigns the processor to another kernel thread, perhaps one that has been assigned by our user-level scheduler to a lower-priority thread. Thus the thread deemed the most important by the user-level scheduler is denied immediate use of the processor by the kernel scheduler in favor of a less important thread.

Another problem is the number of kernel threads. How many kernel threads should be created to support a particular process? If there are too few, then the available concurrency will not be realized—user threads that are ready to run will stand idle, even though there may also be idle processors. If there are too many, then the kernel may needlessly be multiplexing a number of kernel threads on a smaller number of processors, wasting time doing the context switching, even though the application has no need for such time slicing.

One might be tempted to give a process just as many kernel threads as there are processors. But if a user thread executes a blocking system call (such as reading from an I/O device) or suffers a page fault, then its underlying kernel thread also blocks—another user thread may be ready to execute, but no kernel thread is available to be assigned to it.

An elegant solution to both problems, not yet appearing in a commercial system, is *scheduler activations*, an approach devised at the University of Washington [Anderson et al. 1992]. This variant of the many-to-many model provides an explicit means for the user-level and kernel schedulers to cooperate. The kernel assigns processors to processes and the user-level scheduler assigns these processors to user threads. The user-level scheduler keeps the kernel apprised of how many processors it needs; the kernel scheduler notifies the user-level scheduler of all processor-related events that affect the user process, such as when processors become available to it or are taken away from it. This model appears to solve many of the problems of the many-to-many family of models; its greatest drawback is perhaps the frequent crossings of the user-kernel boundary.

3 Solaris Implementation of the Many-to-Many Model

3.1 Overview

Solaris 2.5 provides an implementation of the many-to-many model [Powell et al. 1991] and introduces a new set of vocabulary: a kernel thread in Solaris is referred to as a *lightweight process* (LWP), while a user thread is simply a *thread*. The Solaris threads package is intended to isolate the programmer as much as possible from the notion of LWPs.

3.2 User-level Thread Scheduling

3.2.1 User-level Thread States

Unbound threads in Solaris may be in one five states: **Stopped**, **Blocked**, **Run queue**, **Dispatchable** or **On LWP**. A thread that has been suspended is **Stopped**, while a thread blocked on a synchronization primitive is **Blocked**. If a thread is runnable but is not running on an LWP, then it is either on the **Run queue** or, if an LWP has been found to run the thread, it is **Dispatchable**. Once a runnable thread is picked up by an LWP, it is **On LWP**. While a thread cannot be actually running on a CPU unless it is **On LWP**, being **On LWP** does not imply that a thread is running on a CPU; the underlying LWP itself could be sleeping, waiting for a processor, etc.

3.2.2 Thread-LWP Interaction

Solaris implements the multiplexing of user-level threads onto LWPs by maintaining a *pool* of LWPs. Any unbound thread may run on any LWP in the pool; when a thread is ready to run (i.e. in the user-level run queue), the user-level scheduler takes an LWP out of the pool and assigns it to run the newly runnable thread (changing the thread's state to **On LWP**). This LWP will continue to run the thread until either a thread at a higher priority becomes runnable or the thread blocks on a synchronization primitive. Thus, the user-level threads library is *nonpreemptable* when all threads have the same priority.

When an LWP is idle (i.e. the LWP is in the pool and no threads are runnable), the user-level scheduler *parks* it in the kernel. If a thread becomes runnable while LWPs are parked, the user-level scheduler *unparks* one of the LWPs. Once an LWP is unparked, it dequeues and runs a user thread from the user-level run queue.

3.3 LWP Pool Management

The size of the LWP pool has a critical impact on the performance of the many-to-many model: if the number of LWPs in the pool is nearly equal to the number of threads, the implementation will act much like the one-to-one model. Conversely, if there are very few LWPs in the pool, the implementation will act like the many-to-one model.

Of particular concern is the risk of deadlock with an excessively small pool: one thread may block on a resource in the kernel and go to sleep, and by so doing block the LWP needed to run the resource-holder. To solve this problem, the threads package makes a minimal guarantee to the threads programmer: progress will always be made. This is implemented through the use of the *SIGWAITING* signal. When the kernel realizes that all of a process's LWPs are blocked at the kernel level, it drops a *SIGWAITING* on the process. Upon receipt of the signal, the user-level threads package decides whether or not to create a new LWP, on the basis of the number of runnable threads. The *SIGWAITING* mechanism makes no guarantees about optimal use of LWPs on a multiprocessor. Specifically, a process may have many more runnable user-level threads than it has LWPs, but it does not receive a *SIGWAITING* until all LWPs are blocked. Thus, even if there are processors available and work to be done, the *SIGWAITING* mechanism does not guarantee that there is a sufficient number of LWPs to run the user threads on the available processors. If the programmer wishes to use unbound threads and take advantage of all available processors, he or she is required to advise the library on the number of LWPs required.

4 ThreadMon

ThreadMon is our tool for monitoring multithreaded programs. In this section we give a brief description of its implementation, then discuss its use in subsequent sections.

4.1 Existing Tools

Traditional performance debuggers (e.g. call profilers) offer little help; simply knowing *where* a thread spent its time does not aid in analysis of the model. While postmortem tracing tools such as *tnfview*† allow some performance analysis of specific programs, they offer little insight into the effectiveness of the model itself. Moreover, the sheer volume of data generated makes it difficult to spot detrimental anomalous performance behavior.

While existing tools make the model difficult to evaluate on a uniprocessor, they make its evaluation on a multiprogrammed multiprocessor virtually impossible. To perform this kind of

†Available at *ftp://opcom.sun.ca/pub/tnf*.

analysis, *runtime* correlation of thread, LWP and CPU behavior is required. To this end, we implemented *ThreadMon*, a tool which graphically displays the runtime interactions in the Solaris implementation of the many-to-many threads model.

4.2 ThreadMon Overview

ThreadMon displays runtime information for each user-level thread, LWP and CPU. It provides not only the *state information* for each of these computational elements, but also the *mappings* between them: which thread is running on which LWP, and which LWP is running on which CPU. Thus, to a large degree, one can watch the scheduling decisions made by both the user-level threads package and the kernel, and view how those decisions affect thread state, LWP state, and most importantly, CPU usage. We have been able to use this tool effectively to analyze the decisions made by the many-to-many model.

4.3 Features

As shown in Figures 1 through 6, ThreadMon can display a variety of information about a multi-threaded program. Figure 1 shows a display of threads in a program, giving the percentage time they have been blocked averaged over the last three seconds and the last thirty seconds. Figure 2 gives more detail about what each thread is doing, showing the percentage time each thread spends in the various user-thread states. Figure 3 shows the percentage of time each LWP spends in the various LWP (kernel) states. Figure 4 lists the synchronization primitives discovered in a program's three modules (*atexit*, *main*, and *drand48*). In Figure 5 we have asked for more detail and it has given us a listing of synchronization primitives first used in the routines *fopen* and *drand48*. The programmer has source code for neither of these routines, so no names are available for their synchronization primitives. Finally, Figure 6 shows the mapping of threads to LWPs at one moment of a program's execution.

4.4 Implementation Details

To minimize probe effects, we did not want to display runtime data on the same machine as the monitored program. Thus, ThreadMon consists of two discrete parts: a *library side* that gathers data in the monitored program, and a remote *display side* that presents the data graphically.

In order to allow monitoring of arbitrary binaries, the library side is implemented as a shared library. Thus, to monitor a program, the user sets the *LD_PRELOAD* environment variable to point to the ThreadMon library. This forces ThreadMon to be loaded before other shared libraries. Once loaded, ThreadMon connects to the remote display side and goes on with the program. As the program continues, ThreadMon wakes up every 10 milliseconds,[†] gathers data, and forwards that data to the display side. The gathering of data at the 10-millisecond rate requires approximately ten percent of one CPU on a four-processor 40-MHz SparcStation 10. In practice, we have found that this probe effect is not significant enough to drastically change a program's performance characteristics. However, for the skeptical, a nice fringe benefit of ThreadMon is its ability to monitor itself: by examining the thread and LWP which ThreadMon uses, the probe effect can be measured.

ThreadMon uses several OS services to perform data gathering:

- *Interpositioning*. The most important data is gathered by the library by *interpositioning* between the user-level threads library and itself. That is, ThreadMon redefines many of

[†]The monitor runs as a separate thread bound to its own LWP.

the functions that the user-level threads library uses internally to change the state of threads and LWPs.

- *Process file system* [Faulkner and Gomes 1991]. The */proc* file system offers a wealth of performance information. Specifically, *PIOCLUSAGE* is used to determine LWP states.
- *Kernel statistics interface*. The *kstat* interface is used to obtain CPU usage statistics.
- *Trace Normal Form*. Unfortunately, there is no existing operating-system service to determine the mappings between LWPs and CPUs. To get this information, we used the TNF kernel probes present in Solaris 2.5 and extrapolated the mapping information. For a variety of reasons, this extrapolation is extremely expensive. The TNF monitoring is off by default; when it is turned on, ThreadMon typically consumes fifty percent of one CPU on a four-processor SparcStation 10.
- *Mmap*ing *of /dev/kmem*. For some statistics, we have found it significantly faster to delve straight into kernel memory.

5 Sample Problems

We have used ThreadMon to analyze a number of programs. In this section we discuss three programs whose performance problems are representative of the sorts of problems we have been able to diagnose and fix with ThreadMon. The first of these, called *mutex loop*, was intended to be a test of the performance of mutexes. The second, *flow*, simulates and displays the flow of air across a model of the space shuttle. The third, *matrix mult*, multiplies two matrices using one thread per processor.

5.1 Mutex Loop

This program was intended to be a means for modeling and testing how mutexes behave in "real" programs. It maintained a set of mutexes. Each thread randomly selects a mutex from the set, locks it, loops for a small (specified) period of time, unlocks the mutex, loops for a period of time randomly selected over a particular interval, and then repeats (for a specified number of iterations). Our expectation was that there would be few collisions; i.e., when a thread attempted to lock a mutex, it was unlikely that it would find the mutex locked by another thread and have to wait. This would mean that the running time of the program would improve almost linearly with the number of processors employed, up to the number of threads being used. However, when we performed the experiment, we discovered that there was no such linear speedup—the program did speed up as more processors were added, but not nearly as quickly as expected.

Examination of the program with ThreadMon showed the cause of the lackluster speedup to be a mutex inside of the library implementation of *erand48*, the random-number generator we were employing. The use of a mutex in this routine was totally unexpected, since there was no apparent reason for its existence—since all state information used by *erand48* is supplied in an argument, it has no need to access data that is potentially shared with other threads. We substituted a different random-number generator that we verified lacked internal mutexes, and finally achieved the speedups we had been expecting.

5.2 Flow

Flow, written by Thomas W. Meyer and discussed in [Meyer 1996], is a *time-critical* (or *soft real-time*) program, in that it must maintain a relatively constant performance level. It displays the flow of air around the Space Shuttle. One can manipulate the shuttle as well as any number of rakes that emit streamlines (simulated smoke) showing the flow. The program displays a new frame every fraction of a second, a rate which must be kept fairly steady. The display of each frame requires, for each of a number of objects, the computation of the object itself and then the rendering of the computed object. Since the graphics pipeline is single-threaded, all rendering must be done by the same thread. However, since the computations can be done by any number of threads, we have one compute thread per processor. Scheduling the computations, subject to the constraints that each object must be computed before it is rendered and that the combined time for computing and rendering all objects must be less than the time between frames, is an inherently difficult problem. The scheduler computes an initial feasible schedule and then refines it as time permits. A poor job of scheduling makes the frame rate vary; good scheduling results in a steady and relatively fast frame rate. Due to the time limitations of maintaining the desired frame rate, not all objects are computed and rendered in each frame—the scheduler determines which objects are to appear.

The program is driven by the desired frame rate. The more time available to compute, the more detailed is the frame produced. The time required for each of the compute and rendering tasks is known; the scheduler's job is to ensure that enough computation tasks are performed that sufficient detail is available for each frame, but that these tasks are done in the time allotted to produce a frame. To obtain the best use of a multiprocessor, the scheduler should distribute the computing tasks evenly over all of the processors—any idle time on a processor is time that could have been spent performing a task assigned to another processor (and thus speeding up the computation) or that could have been used performing an additional task (and thus adding more detail to the frame).

Debugging the scheduler with conventional tools was difficult—it was not easy, for example, to verify that its determination of the running times of the various tasks was correct and that the schedule produced resulted in the balanced use of the processors. When we first applied ThreadMon to the program we found that there were unsuspected bugs in the scheduler and that the processor usage was, indeed, unbalanced—this was clearly indicated by both the activity displays of the compute threads and the activity displays of the synchronization variables. Once the problem was identified, it was easily fixed and the solution verified by monitoring the program with ThreadMon.

5.3 Matrix Mult

This program, taken from a recent book on multithreaded programming [Kleiman et al. 1996], uses a straightforward algorithm for multiplying two matrices: each element of the product matrix is computed as an inner product of a row and a column of the multiplier and multiplicand matrices. These inner products are divided evenly among the threads of the program, each of which computes its set of inner products and then waits for the others at a barrier. The intent of this strategy is to insure that all processors are employed in computing the solution. The intuitive view of most programmers is that each thread represents a processor. Thus, if one has n processors, one should use n threads. However, the number of processors really utilized can depend very strongly on the implementation model. With the many-to-one model, only one kernel thread is used and thus one processor is used; with the one-to-one model, n kernel threads are used and thus, at the kernel scheduler's discretion, up to n processors are used; with the many-to-many model, N kernel threads are used, where N is some number less than or equal to n (which one would have to set to be n) and thus up to N processors are used. Even with the scheduler-activations model, one is still at the mercy of the kernel scheduler as to how many processors are actually used.

With all models, certainly no more processors can be in use at any one time than the number of runnable threads. [Kleiman et al. 1996] suggest that one use two more threads than processors, so that, for example, if one thread blocks for some reason, another thread is available to run on the processor that has just been made available.

We ran the code on a four-processor machine, and thus used six threads. To study the behavior of this program on the many-to-many model (using Solaris threads), we varied the number of kernel threads. First we used one kernel thread, taking time 6T (where T is the time required for one thread to compute its portion of the computation). We then used two kernel threads: not surprisingly, this ran in time 3T. We got the expected performance when increasing the number of kernel threads to three: 2T. However, when we increased the number of kernel threads to four, the running time was still 2T, even though this meant we were able to use four processors rather than three. By increasing the number of kernel threads to five the running time actually increased to approximately 2.25T. Only by increasing the number of kernel threads to six did we get the four-fold speedup of 1.5T.

The subsequent analysis obtained with ThreadMon showed what, in hindsight, was the obvious problem here. With one kernel thread, we employed only one processor and thus there was no parallelism. In fact, since there is no time-slicing in the Solaris user-level scheduler, each thread ran to completion (i.e., until it reached the barrier) before the next thread started. With two kernel threads, two threads could run simultaneously, and thus the running time was reduced by a factor of two. With three kernel threads, the running time was reduced proportionally more. However, with four kernel threads, first four user threads ran in parallel and reached the barrier at more or less the same time. But then two user threads remained to run—though four processors were available to run them, only two could be used. Thus four threads ran at once followed by two threads running at once, which produced the same overall running time as the three-kernel-thread solution (three threads running at once followed by three threads running at once). With five kernel threads, first five user threads ran (multiplexed) on four processors, requiring time 1.25T, then the final thread ran, increasing the total time to 2.25T. With six kernel threads, all six user threads ran on four processors, requiring a total time of 1.5T.

6 The Many-to-Many Model Vs. the One-to-One Model

The many-to-many model appears to have all the advantages of both the many-to-one model and the one-to-one model. This is certainly the case in those applications that are ideally suited for it, ones in which there are sufficient number of active user threads so that kernel threads rarely have to "park." But what if the application is not so ideally suited? Will the many-to-many model always perform at least as well as the one-to-one model? By binding a user thread to a kernel thread, we can get the effects of the one-to-one model within the many-to-many model. This is useful both for comparison purposes and for performance, should there be situations in which the one-to-one model is superior to the many-to-many model.

We focussed our attention on compute-intensive applications on multiprocessors. We took as the archetypical compute-intensive application the matrix-mult example of Section 5.3 and reduced it to a program whose threads make successive iterations of an arbitrary computation followed by synchronization at a barrier [Chen et al. 1990]. The amount of computation per iteration was made a parameter, so that we could adjust the granularity of the synchronization.

Figure 7 gives log-log plots of the performance of a fine-grained barrier on a four-processor machine with four bound threads and the performance of four unbound threads with an LWP pool of size four on the same machine. The number of iterations made by each thread varies along the x-axis. Figure 8 is derived from Figure 7 and represents the percentage penalty for using four unbound threads rather than four bound threads.

It is not immediately clear to most programmers what the significance is of being bound. The common wisdom in Solaris has been that it is important to bind a thread to an LWP if one wants the thread to be scheduled by the operating system at a high priority—since the operating system schedules only LWPs, one must bind the thread to its LWP to take advantage of the LWP's priority (otherwise the LWP might be switched to running some other thread). As we discuss below, however, there are other reasons to consider using bound threads.

6.1 Synchronization Overhead

On a multiprocessor machine with n processors, we felt that a multithreaded program with n bound threads should perform no differently from a program with n unbound threads and an LWP pool of size n, i.e., the performance of the many-to-many model should be no worse than the performance of the one-to-one model. To test this hypothesis, we ran our barrier code on a four-processor machine[†] with four unbound threads and an LWP-pool size of four (set with *thr_setconcurrency*) and again with four bound threads. Our results were surprising: when the granularity was fine, the program using unbound threads ran slower than the one using bound threads (Figure 7). Moreover, the difference was highly dependent upon the amount of work done between barrier synchronization operations; as more work was done, the difference became increasingly small.

Before discussing the monitoring results, we note how to interpret the screen shots: in all of the examples, threads one through three are blocked for various reasons. This is entirely normal; thread one is the main thread, which spawns the worker threads and then goes to sleep. Thread two is the thread assigned to handling callouts, and is thus bound to an LWP; it spends virtually all of its life blocked in the kernel. Thread three is the thread assigned to the dynamic creation of LWPs (i.e. it handles *SIGWAITING*); it spends its life blocked at user-level. Finally, thread four is the bound thread that the monitor itself uses for reporting to the display side. Thus, all interesting behavior is seen by examining threads five and higher.

ThreadMon showed (see Figures 9 and 10) that, as expected, the working threads were dividing their time between being blocked in the barrier (in the **Blocked** state) and running their computation (in the **On LWP** state). It revealed, however, that in the unbound version threads were also spending a significant amount of time in the **Dispatchable** state (Figure 9). In the bound version, on the other hand, threads did not spend time in this state (Figure 10).

Synchronization is supposed to be cheap under the many-to-many model, in part because blocking a thread does not necessarily imply blocking the LWP; threads that need to block are separated from their underlying LWP, which then attempts to find other work. As mentioned in Section 3.2.2, if there is no work for an LWP to do, the user-level scheduler parks it in the kernel. When a thread is made runnable, the user-level scheduler marks the thread **Dispatchable** and unparks an LWP. When the LWP returns from parking, it finds the **Dispatchable** thread, changes its state to **On LWP**, and runs it.

If there are always user threads in the **Dispatchable** state, then this scenario would never occur—there would be no need to park an LWP, since there would always be a thread for it to run. However, the need to park an LWP can certainly occur, and there is a cost associated with unparking an LWP and getting it to find a dispatchable thread. Our experiment shows that much of this cost can be avoided by binding threads to LWPs, i.e., by moving to the one-to-one model. On the other hand, the synchronization of bound threads always involves system calls—there is no potential for eliminating system calls as there is with unbound threads, in which an LWP running one thread can quickly switch from a blocked thread to a dispatchable thread [Powell et al. 1991, Catanzaro 1994, Vahalia 1996, Kleiman et al. 1996]. The programmer must decide which strategy will be the most efficient—ThreadMon provides the means for doing so.

[†] SparcStation10/404ZX running Solaris 2.5.

6.2 Time Slicing

As was clear in our matrix-mult example (Section 5.3), the execution of user threads on LWPs is not time-sliced in the Solaris implementation of the many-to-many model—each thread runs until it blocks, terminates, or is preempted by a higher-priority thread. This lack of time-slicing is not a performance problem, but it is different from what many programmers expect. If it is important that threads be time-sliced, they should be bound to their LWPs. Since it might not be readily apparent to the programmer that time slicing is important, ThreadMon is an important aid to discovering one's need for time slicing.

7 Conclusions and Future Work

ThreadMon has proven to be a valuable tool for performance debugging. It has solved a number of mysteries for us—performance problems that we would have had a difficult time analyzing with other tools. Like many bugs, our problems often ended up having simple explanations and simple solutions. We have shown three classes of problems for which ThreadMon has been of great help:

1) Unexpected interaction with library routines: as shown in our mutex-loop example (Section 5.1), library routines can contain unsuspected synchronization and other code that interferes with a multithreaded program.
2) Debugging of control logic: as shown in our flow example (Section 5.2), it can be difficult to debug code that controls when threads execute. The timing issues involved are not easily observed with conventional tools.
3) Taking advantage of (and avoiding disadvantages of) the underlying thread model: as shown in our matrix-mult example (Section 5.3) and discussed in Section 6, it is important to make certain that one is using the underlying threads model to its best advantage. There can be a number of nonintuitive effects can be made apparent with ThreadMon.

We are very much interested in applying ThreadMon to a system employing scheduler activations. The first two classes of problems will still be important. What is unknown are what, if any, problems there will be of the third class.

The largest program to which we have applied ThreadMon has been Flow (Section 5.2), which is several thousand lines in length, although with relatively simple synchronization. We are currently working on a new version with fewer system dependencies that we plan to distribute to a number of people working on a variety of large multithreaded applications. Among the issues we will address is coping with hundreds, if not thousands of synchronization primitives and threads, presenting useful information without overwhelming the programmer. Our current approach for this, as seen in Figures 4 and 5, is to display summary information about aggregates, allowing programmers to focus on more detailed views when things look interesting. It is not clear how well this will scale to large aggregates—we need large programs to experiment with.

Despite the recent popularity of multithreaded programming, we have been having a difficult time obtaining nontrivial test programs. This is perhaps indicative of the perceived difficulty of writing multithreaded code. We hope that with research into tools such as ThreadMon, some of the mystery behind multithreaded programs will go away and people will be more willing to take advantage of this useful paradigm.

8 References

ANDERSON, T.E., BERSHAD, B.N., LAZOWSKA, AND LEVY, H.M. 1992. Scheduler activations: effective kernel support for the user-level management of parallelism. *ACM Trans. on Comp. Syst. 10*, 4 (Feb), 53–70

CATANZARO, B. 1994. *Multiprocessor System Architectures.* SunSoft Press.

CHEN, D.K., SU, H.H., AND YEW, P.C. 1990. The impact of synchronization and granularity in parallel systems. In *Proceedings of the 17th Annual International Symposium on Computer Architecture.* 239–248.

FAULKNER, R. AND GOMES, R. 1991. The process file system and process model in UNIX System V. In *Winter USENIX Conference Proceedings* (Jan). USENIX Assoc, El Cerrito, Calif., 243–252.

GOLUB, D., DEAN, R., FORIN, A., AND RASHID, R. 1990. UNIX as an application program. In *Summer USENIX Conference Proceedings* (June). USENIX Assoc, El Cerrito, Calif., 87–96.

DOEPPNER, T. W. 1987. Threads: a system for the support of concurrent programming. Technical Report CS-87-11, Brown University, Department of Computer Science, Providence, RI (June).

KLEIMAN, S.R., SHAH, D., SMAALDERS, B. 1996. *Programming with Threads*. SunSoft Press.

MEYER, T.W. 1996. *Scheduling time-critical graphics on multiple processors*. Master's Thesis, Brown University, Department of Computer Science, Providence, RI.

POWELL, M.L., KLEIMAN, S.R., BARTON, S., SHAH, D., STEIN, D., WEEKS, M. 1991. SunOS multithread architecture. In *Winter USENIX Conference Proceedings* (Jan). USENIX Assoc, El Cerrito, Calif., 65–80.

VAHALIA, U. 1996. *UNIX internals: the new frontiers.* Prentice Hall.