Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism

THOMAS E. ANDERSON, BRIAN N. BERSHAD, EDWARD D. LAZOWSKA, and HENRY M. LEVY
University of Washington

Threads are the vehicle for concurrency in many approaches to parallel programming. Threads can be supported either by the operating system kernel or by user-level library code in the application address space, but neither approach has been fully satisfactory.

This paper addresses this dilemma. First, we argue that the performance of kernel threads is *inherently* worse than that of user-level threads, rather than this being an artifact of existing implementations; managing parallelism at the user level is essential to high-performance parallel computing. Next, we argue that the problems encountered in integrating user-level threads with other system services is a consequence of the lack of kernel support for user-level threads provided by contemporary multiprocessor operating systems; kernel threads are the *wrong abstraction* on which to support user-level management of parallelism. Finally, we describe the design, implementation, and performance of a new kernel interface and user-level thread package that together provide the same functionality as kernel threads without compromising the performance and flexibility advantages of user-level management of parallelism.

Categories and Subject Descriptors: D.4.1 [Operating Systems]: Process Management; D.4.4 [Operating Systems]: Communications Management—*input/output*; D.4.7 [Operating Systems]: Organization and Design; D.4.8 [Operating Systems]: Performance
General Terms: Design, Measurement, Performance
Additional Key Words and Phrases: Multiprocessor, thread

1. INTRODUCTION

The effectiveness of parallel computing depends to a great extent on the performance of the primitives that are used to express and control the parallelism within programs. Even a coarse-grained parallel program can

This work was supported in part by the National Science Foundation (grants CCR-5619663, CCR-5703049, and CCR.8907666), the Washington Technology Center, and Digital Equipment Corporation (the Systems Research Center and the External Research Program). Anderson was supported by an IBM Graduate Fellowship and Bershad by an AT&T Ph.D. Scholarship. Anderson is now with the Computer Science Division, University of California at Berkeley; Bershad is now with the School of Computer Science, Carnegie Mellon University.
Authors' address: Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195.

exhibit poor performance if the cost of creating and managing parallelism is high. Even a fine-grained program can achieve good performance if the cost of creating and managing parallelism is low.

One way to construct a parallel program is to share memory between a collection of traditional UNIX-like processes, each consisting of a single address space and a single sequential execution stream within that address space. Unfortunately, because such processes were designed for multiprogramming in a uniprocessor environment, they are simply too inefficient for general-purpose parallel programming; they handle only coarse-grained parallelism well.

The shortcomings of traditional processes for general-purpose parallel programming have led to the use of *threads*. Threads separate the notion of a sequential execution stream from the other aspects of traditional processes such as address spaces and I/O descriptors. This separation of concerns yields a significant performance advantage relative to traditional processes.

1.1 The Problem

Threads can be supported either at user level or in the kernel. Neither approach has been fully satisfactory.

User-level threads are managed by runtime library routines linked into each application so that thread management operations require no kernel intervention. The result can be excellent performance: in systems such as PCR [25] and FastThreads [2], the cost of user-level thread operations is within an order of magnitude of the cost of a procedure call. User-level threads are also flexible; they can be customized to the needs of the language or user without kernel modification.

User-level threads execute within the context of traditional processes; indeed, user-level thread systems are typically built without any modifications to the underlying operating system kernel. The thread package views each process as a "virtual processor," and treats it as a physical processor executing under its control; each virtual processor runs user-level code that pulls threads off the ready list and runs them. In reality, though, these virtual processors are being multiplexed across real, physical processors by the underlying kernel. "Real world" operating system activity, such as multiprogramming, I/O, and page faults, distorts the equivalence between virtual and physical processors; in the presence of these factors, user-level threads built on top of traditional processes can exhibit poor performance or even incorrect behavior.

Multiprocessor operating systems such as Mach [2], Topaz [22], and V [7] provide direct kernel support for multiple threads per address space. Programming with kernel threads avoids the system integration problems exhibited by user-level threads, because the kernel directly schedules each application's threads onto physical processors. Unfortunately, kernel threads, just like traditional UNIX processes, are too heavyweight for use in many parallel programs. The performance of kernel threads, although typically an order of magnitude better than that of traditional processes, has been typically an order of magnitude *worse* than the best-case performance of user-level threads (e.g., in the absence of multiprogramming and I/O). As a result, user-level threads have ultimately been implemented on top of the kernel threads of both Mach (C Threads [8]) and Topaz (WorkCrews [24]). User-level threads are built on top of kernel threads exactly as they are built on top of traditional processes; they have exactly the same performance, and they suffer exactly the same problems.

The parallel programmer, then, has been faced with a difficult dilemma: employ user-level threads, which have good performance and correct behavior provided the application is uniprogrammed and does no I/O, or employ kernel threads, which have worse performance but are not as restricted.

1.2 The Goals of this Work

In this paper we address this dilemma. We describe a kernel interface and a user-level thread package that together combine the functionality of kernel threads with the performance and flexibility of user-level threads. Specifically,

—In the common case when thread operations do not need kernel intervention, our performance is essentially the same as that achieved by the best existing user-level thread management systems (which suffer from poor system integration).

—In the infrequent case when the kernel must be involved, such as on processor reallocation or I/O, our system can mimic the behavior of a kernel thread management system:

  —No processor idles in the presence of ready threads.
  —No high-priority thread waits for a processor while a low-priority thread runs.
  —When a thread traps to the kernel to block (for example, because of a page fault), the processor on which the thread was running can be used to run another thread from the same or from a different address space.

—The user-level part of our system is structured to simplify application-specific customization. It is easy to change the policy for scheduling an application's threads, or even to provide a different concurrency model such as workers [16], Actors [1], or Futures [10].

The difficulty in achieving these goals in a multiprogrammed multiprocessor is that the necessary control and scheduling information is distributed between the kernel and each application's address space. To be able to allocate processors among applications, the kernel needs access to user-level scheduling information (e.g., how much parallelism there is in each address space). To be able to manage the application's parallelism, the user-level support software needs to be aware of kernel events (e.g., processor reallocations and I/O request/completions) that are normally hidden from the application.

1.3 The Approach

Our approach provides each application with a *virtual multiprocessor*, an abstraction of a dedicated physical machine. Each application knows exactly how many (and which) processors have been allocated to it and has complete control over which of its threads are running on those processors. The operating system kernel has complete control over the allocation of processors among address spaces including the ability to change the number of processors assigned to an application during its execution.

To achieve this, the kernel notifies the address space thread scheduler of every kernel event affecting the address space, allowing the application to have complete knowledge of its scheduling state. The thread system in each address space notifies the kernel of the subset of user-level thread operations that can affect processor allocation decisions, preserving good performance for the majority of operations that do not need to be reflected to the kernel.

The kernel mechanism that we use to realize these ideas is called *scheduler activations*. A scheduler activation vectors control from the kernel to the address space thread scheduler on a kernel event; the thread scheduler can use the activation to modify user-level thread data structures, to execute user-level threads, and to make requests of the kernel.

We have implemented a prototype of our design on the DEC SRC Firefly multiprocessor workstation [22]. While the differences between scheduler activations and kernel threads are crucial, the similarities are great enough that the kernel portion of our implementation required only relatively straightforward modifications to the kernel threads of Topaz, the native operating system on the Firefly. Similarly, the user-level portion of our implementation involved relatively straightforward modifications to FastThreads, a user-level thread system originally designed to run on top of Topaz kernel threads.

Since our goal is to demonstrate that the exact functionality of kernel threads can be provided at the user level, the presentation in this paper assumes that user-level threads are the concurrency model used by the programmer or compiler. We emphasize, however, that other concurrency models, when implemented at user level on top of kernel threads or processes, suffer from the same problems as user-level threads—problems that are solved by implementing them on top of scheduler activations.

2. USER-LEVEL THREADS: PERFORMANCE ADVANTAGES AND FUNCTIONALITY LiMITATIONS

In this section we motivate our work by describing the advantages that user-level threads offer relative to kernel threads, and the difficulties that arise when user-level threads are built on top of the interface provided by kernel threads or processes. We argue that the performance of user-level threads is inherently better than that of kernel threads, rather than this being an artifact of existing implementations. User-level threads have an additional advantage of flexibility with respect to programming models and environments. Further, we argue that the lack of system integration exhibited by user-level threads is not inherent in user-level threads themselves, but is a consequence of inadequate kernel support.

2.1 The Case for User-Level Thread Management

It is natural to believe that the performance optimizations found in user-level thread systems could be applied within the kernel, yielding kernel threads that are as efficient as user-level threads without the compromises in functionality. Unfortunately, there are significant inherent costs to managing threads in the kernel:

—*The cost of accessing thread management operations:* With kernel threads, the program must cross an extra protection boundary on every thread operation, even when the processor is being switched between threads in the same address space. This involves not only an extra kernel trap, but the kernel must also copy and check parameters in order to protect itself against buggy or malicious programs. By contrast, invoking user-level thread operations can be quite inexpensive, particularly when compiler techniques are used to expand code inline and perform sophisticated register allocation. Further, safety is not compromised: address space boundaries isolate misuse of a user-level thread system to the program in which it occurs.

—*The cost of generality:* With kernel thread management, a single underlying implementation is used by all applications. To be general-purpose, a kernel thread system must provide any feature needed by any reasonable application; this imposes overhead on those applications that do not use a particular feature. In contrast, the facilities provided by a user-level thread system can be closely matched to the specific needs of the applications that use it, since different applications can be linked with different user-level thread libraries. As an example, most kernel thread systems implement preemptive priority scheduling, even though many parallel applications can use a simpler policy such as first-in-first-out [24].

These factors would not be important if thread management operations were inherently expensive. Kernel trap overhead and priority scheduling, for instance, are not major contributors to the high cost of UNIX-like processes. However, the cost of thread operations can be within an order of magnitude of a procedure call. This implies that any overhead added by a kernel implementation, however small, will be significant, and a well-written user-level thread system will have significantly better performance than a well-written kernel-level thread system.

To illustrate this quantitatively, Table I shows the performance of example implementations of user-level threads, kernel threads, and UNIX-like processes, all running on similar hardware, a CVAX processor. FastThreads and Topaz kernel threads were measured on a CVAX Firefly; Ultrix (DEC's derivative of UNIX) was measured on a CVAX uniprocessor workstation. (Each of these implementations, while good, is not "optimal." Thus, our measurements are illustrative and not definitive).

The two benchmarks are *Null Fork*, the time to create, schedule, execute and complete a process/thread that invokes the null procedure (in other words, the overhead of forking a thread), and Signal-Wait, the time for a

Table I: Thread Operation Latencies (μsec.)

Operation	FastThreads	Topaz threads	Ultrix processes
Null Fork	34	948	11300
Signal-Wait	37	441	1840

process/thread to signal a waiting process/thread, and then wait on a condition (the overhead of synchronizing two threads together). Each benchmark was executed on a single processor, and the results were averaged across multiple repetitions. For comparison, a procedure call takes about 7 μsec. on the Firefly, while a kernel trap takes about 19 μsec.

Table I shows that while there is an order of magnitude difference in cost between Ultrix process management and Topaz kernel thread management, there is yet another order of magnitude difference between Topaz threads and FastThreads. This is despite the fact that the Topaz thread code is highly tuned with much of the critical path written in assembler.

Commonly, a tradeoff arises between performance and flexibility in choosing where to implement system services [26]. User-level threads, however, avoid this tradeoff: they simultaneously improve both performance and flexibility. Flexibility is particularly important in thread systems since there are many parallel programming models, each of which may require specialized support within the thread system. With kernel threads, supporting multiple parallel programming models may require modifying the kernel, which increases complexity, overhead, and the likelihood of errors in the kernel.

2.2 Sources of Poor Integration in User-Level Threads Built on the Traditional Kernel Interface

Unfortunately, it has proven difficult to implement user-level threads that have the same level of integration with system services as is available with kernel threads. This is not inherent in managing parallelism at the user level, but rather is a consequence of the lack of kernel support in existing systems. Kernel threads are the *wrong abstraction* for supporting user-level thread management. There are two related characteristics of kernel threads that cause difficulty:

—Kernel threads block, resume, and are preempted without notification to the user level.

—Kernel threads are scheduled obliviously with respect to the user-level thread state.

These can cause problems even on a uniprogrammed system. A user-level thread system will often create as many kernel threads to serve as "virtual processors" as there are physical processors in the system; each will be used to run user-level threads. When a user-level thread makes a blocking I/O request or takes a page fault, though, the kernel thread serving as its virtual processor also blocks. As a result, the *physical* processor is lost to the address space while the I/O is pending, because there is no kernel thread to run other user-level threads on the just-idled processor.

A plausible solution to this might be to create more kernel threads than physical processors; when one kernel thread blocks because its user-level thread blocks in the kernel, another kernel thread is available to run user-level threads on that processor. However, a difficulty occurs when the I/O completes or the page fault returns: there will be more runnable kernel threads than processors, each kernel thread in the middle of running a user-level thread. In deciding which kernel threads are to be assigned processors, the operating system will implicitly choose which user-level threads are assigned processors.

In a traditional system, when there are more runnable threads than processors, the operating system could employ some kind of time-slicing to ensure each thread makes progress. When user-level threads are running on top of kernel threads, however, time-slicing can lead to problems. For example, a kernel thread could be preempted while its user-level thread is holding a spin-lock; any user-level threads accessing the lock will then spin-wait until the lock holder is rescheduled. Zahorjan et al. [28] have shown that time-slicing in the presence of spin-locks can result in poor performance. As another example, a kernel thread running a user-level thread could be preempted to allow another kernel thread to run that happens to be idling in its user-level scheduler. Or a kernel thread running a high-priority user-level thread could be descheduled in favor of a kernel thread that happens to be running a low-priority user-level thread.

Exactly the same problems occur with multiprogramming as with I/O and page faults. If there is only one job in the system, it can receive all of the machine's processors; if another job enters the system, the operating system should preempt some of the first job's processors to give to the new job [23]. The kernel then is forced to choose which of the first job's kernel threads, and thus implicitly which user-level threads, to run on the remaining processors. The need to preempt processors from an address space also occurs due to variations in parallelism within jobs; Zahorjan and McCann [27] show that the dynamic reallocation of processors among address spaces in response to variations in parallelism is important to achieving high performance.

While a kernel interface can be designed to allow the user level to influence which kernel threads are scheduled when the kernel has a choice [5], this choice is intimately tied to the user-level thread state; the communication of this information between the kernel and the user-level negates many of the performance and flexibility advantages of using user-level threads in the first place.

Finally, ensuring the logical correctness of a user-level thread system built on kernel threads can be difficult. Many applications, particularly those that require coordination among multiple address spaces, are free from deadlock based on the assumption that all runnable threads eventually receive processor time. When kernel threads are used directly by applications, the kernel satisfies this assumption by time-slicing the processors among all of the runnable threads. But when user-level threads are multiplexed across a fixed number of kernel threads, the assumption may no longer hold: because a kernel thread blocks when its user-level thread blocks, an application can run out of kernel threads to serve as execution contexts, even when there are runnable user-level threads and available processors.

3. EFFECTIVE KERNEL SUPPORT FOR THE USER-LEVEL MANAGEMENT OF PARALLELISM

Section 2 described the problems that arise when kernel threads are used by the programmer to express parallelism poor performance and poor flexibility) and when user-level threads are built on top of kernel threads (poor behavior in the presence of multiprogramming and I/O). To address these problems, we have designed a new kernel interface and user-level thread system that together combine the functionality of kernel threads with the performance and flexibility of user-level threads.

The operating system kernel provides each user-level thread system with its own virtual multiprocessor, the abstraction of a dedicated physical machine except that the kernel may change the number of processors in that machine during the execution of the program. There are several aspects to this abstraction:

—The kernel allocates processors to address spaces; the kernel has complete control over how many processors to give each address space's virtual multiprocessor.

—Each address space’s user-level thread system has complete control over which threads to run on its allocated processors, as it would if the application were running on the bare physical machine.

—The kernel notifies the user-level thread system whenever the kernel changes the number of processors assigned to it; the kernel also notifies the thread system whenever a user-level thread blocks or wakes up in the kernel (e.g., on I/O or on a page fault). The kernel's role is to vector events to the appropriate thread scheduler, rather than to interpret these events on its own.

—The user-level thread system notifies the kernel when the application needs more or fewer processors. The kernel uses this information to allocate processors among address spaces. However, the user level notifies the kernel only on those subset of user-level thread operations that might affect processor allocation decisions. As a result, performance is not compromised; the majority of thread operations do not suffer the overhead of communication with the kernel.

—*The application programmer sees no difference, except for performance, from programming directly with kernel threads.* Our user-level thread system manages its virtual multiprocessor transparently to the programmer, providing programmers a normal Topaz thread interface [4]. (The user-level runtime system could easily be adapted, though, to provide a different parallel programming model).

In the remainder of this section we describe how kernel events are vectored to the user-level thread system, what information is provided by the application to allow the kernel to allocate processors among jobs, and how we handle user-level spin-locks.

3.1 Explicit Vectoring of Kernel Events to the User-Level Thread Scheduler

The communication between the kernel processor allocator and the user-level thread system is structured in terms of scheduler activations. The term “scheduler activation” was selected because each vectored event causes the user-level thread system to reconsider its scheduling decision of which threads to run on which processors.

A scheduler activation serves three roles:

—It serves as a vessel, or execution context, for running user-level threads, in exactly the same way that a kernel thread does.

—It notifies the user-level thread system of a kernel event.

—It provides space in the kernel for saving the processor context of the activation's current user-level thread, when the thread is stopped by the kernel (e.g., because the thread blocks in the kernel on I/O or the kernel preempts its processor).

A scheduler activation’s data structures are quite similar to those of a traditional kernel thread. Each scheduler activation has two execution stacks — one mapped into the kernel and one mapped into the application address space. Each user-level thread is allocated its own user-level stack when it starts running [2]; when a user-level thread calls into the kernel, it uses its activation's kernel stack. The user-level thread scheduler runs on the activation's user-level stack. In addition, the kernel maintains an activation control block (akin to a thread control block) to record the state of the scheduler activation's thread when it blocks in the kernel or is preempted; the user-level thread scheduler maintains a record of which user-level thread is running in each scheduler activation.

When a program is started, the kernel creates a scheduler activation, assigns it to a processor, and upcalls into the application address space at a fixed entry point. The user-level thread management system receives the upcall and uses that activation as the context in which to initialize itself and run the main application thread. As the first thread executes, it may create more user threads and request additional processors. In this case, the kernel will create an additional scheduler activation for each processor and use it to upcall into the user level to tell it that the new processor is available. The user level then selects and executes a thread in the context of that activation.

Similarly, when the kernel needs to notify the user level of an event, the kernel creates a scheduler activation, assigns it to a processor, and upcalls into the application address space. Once the upcall is started, the activation is similar to a traditional kernel thread—it can be used to process the event, run user-level threads, and trap into and block within the kernel.

Table II: Scheduler Activation Upcall Points

Add this processor (processor #) *Execute a runnable user-level thread.*
Processor has been preempted (preempted activation # and its machine state) *Return to the ready list the user-level thread that was executing in the context of the preempted scheduler activation.*
Scheduler activation has blocked (blocked activation #) *The blocked scheduler activation is no longer using its processor.*
Scheduler activation has unblocked (unblocked activation # and its machine state) *Return to the ready list the user-level thread that was executing in the context of the blocked scheduler activation.*

The crucial distinction between scheduler activations and kernel threads is that once an activation's user-level thread is stopped by the kernel, the thread is never directly resumed by the kernel. Instead, a new scheduler activation is created to notify the user-level thread system that the thread has been stopped. The user-level thread system then removes the state of the thread from the old activation, tells the kernel that the old activation can be reused, and finally decides which thread to run on the processor. By contrast, in a traditional system, when the kernel stops a kernel thread, even one running a user-level thread in its context, the kernel never notifies the user level of the event. Later, the kernel directly resumes the kernel thread (and by implication, its user-level thread), again without notification. By using scheduler activations, the kernel is able to maintain the invariant that there are always exactly as many running scheduler activations (vessels for running user-level threads) as there are processors assigned to the address space.

Table II lists the events that the kernel vectors to the user level using scheduler activations; the parameters to each upcall are in parentheses, and the action taken by the user-level thread system is italicized. Note that events are vectored at exactly the points where the kernel would otherwise be forced to make a scheduling decision. In practice, these events occur in combinations; when this occurs, a single upcall is made that passes all of the events that need to be handled.

As one example of the use of scheduler activations, Figure 1 illustrates what happens on an I/O request/completion. Note that this is the uncommon case; in normal operation, threads can be created, run, and completed, all without kernel intervention. Each pane in Figure 1 reflects a different time step. Straight arrows represent scheduler activations, s-shaped arrows represent user-level threads, and the cluster of user-level threads to the right of each pane represents the ready list.

At time T1, the kernel allocates the application two processors. On each processor, the kernel upcalls to user-level code that removes a thread from the ready list and starts running it. At time T2, one of the user-level threads (thread 1) blocks in the kernel. To notify the user level of this

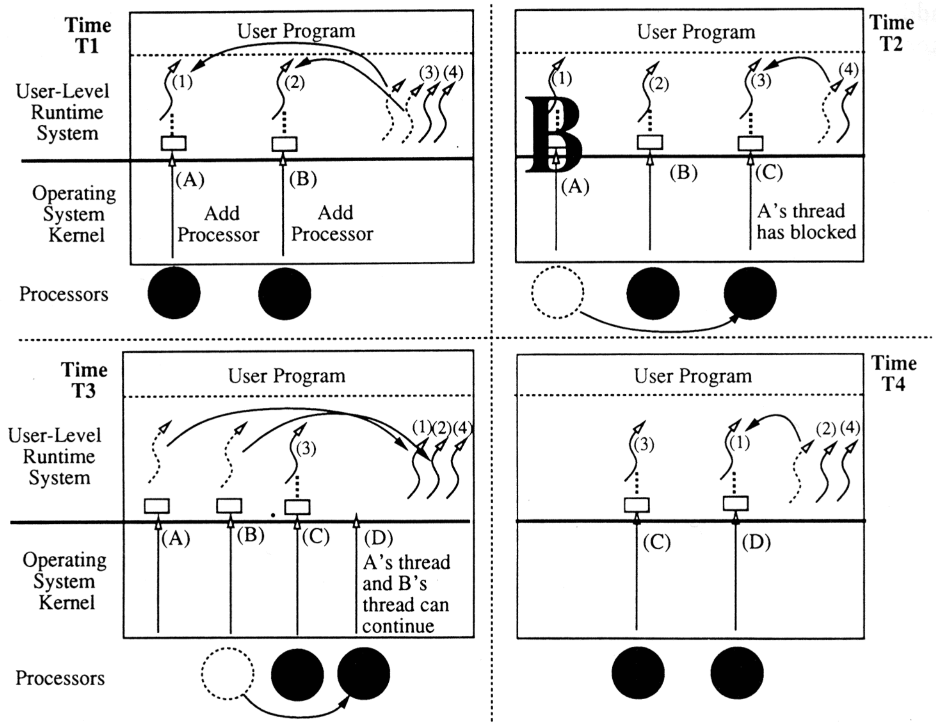

Fig. 1. Example: I/O request/completion.

event, the kernel takes the processor that had been running thread 1 and performs an upcall in the context of a fresh scheduler activation. The user-level thread scheduler can then use the processor to take another thread off the ready list and start running it.

At time T3, the I/O completes. Again, the kernel must notify the user-level thread system of the event, but this notification requires a processor. The kernel preempts one of the processors running in the address space and uses it to do the upcall. (If there are no processors assigned to the address space when the I/O completes, the upcall must wait until the kernel allocates one). This upcall notifies the user level of two things: the I/O completion and the preemption. The upcall invokes code in the user-level thread system that (1) puts the thread that had been blocked on the ready list and (2) puts the thread that was preempted on the ready list. At this point, scheduler activations A and B can be discarded. Finally, at time T4, the upcall takes a thread off the ready list and starts running it.

When a user level thread blocks in the kernel or is preempted, most of the state needed to resume it is already at the user level—namely, the thread's stack and control block. The thread's register state, however, is saved by low-level kernel routines, such as the interrupt and page fault handlers; the kernel passes this state to the user level as part of the upcall notifying the address space of the preemption and/or I/O completion.

We use exactly the same mechanism to reallocate a processor from one address space to another due to multiprogramming. For example, suppose the kernel decides to take a processor away from one address space and give it to another. The kernel does this by sending the processor an interrupt, stopping the old activation, and then using the processor to do an upcall into the new address space with a fresh activation. The kernel need not obtain permission in advance from the old address space to steal its processor; to do so would violate the semantics of address space priorities (e.g., the new address space could have higher priority than the old address space). However, the old address space must still be notified that the preemption occurred. The kernel does this by doing *another* preemption on a different processor still running in the old address space. The second processor is used to make an upcall into the old address space using a fresh scheduler activation, notifying the address space that *two* user-level threads have been stopped. The user-level thread scheduler then has full control over which of these threads should be run on its remaining processors. (When the last processor is preempted from an address space, we could simply skip notifying the address space of the preemption, but instead, we delay the notification until the kernel eventually reallocates it as a processor. Notification allows the user level to know *which* processors it has been assigned, in case it is explicitly managing cache locality).

The above description is over-simplified in several minor respects. First, if threads have priorities, an additional preemption may have to take place beyond the ones described above. In the example in Figure 1, suppose thread 3 is lower priority than both threads 1 and 2. In that case, the user-level thread system can ask the kernel to preempt thread 3's processor. The kernel will then use that processor to do an upcall, allowing the user-level thread system to put thread 3 on the ready list and run thread 2 instead. The user level can know to ask for the additional preemption because it knows exactly which thread is running on each of its processors.

Second, while we described the kernel as stopping and saving the context of user-level threads, the kernel's interaction with the application is entirely in terms of scheduler activations. The application is free to build any other concurrency model on top of scheduler activations; the kernel's behavior is exactly the same in every case. In particular, the kernel needs no knowledge of the data structures used to represent parallelism at the user level.

Third, scheduler activations work properly even when a preemption or a page fault occurs in the user-level thread manager when no user-level thread is running. In this case, it is the thread manager whose state is saved by the kernel. The subsequent upcall, in a new activation with its own stack, allows the (reentrant) thread manager to recover in one way if a user-level thread is running, and in a different way if not. For example, if a preempted processor was in the idle loop, no action is necessary; if it was handling an event during an upcall, a user-level context switch can be made to continue processing the event. The only added complication for the kernel is that an upcall to notify the program of a page fault may in turn page fault on the same location; the kernel must check for this, and when it occurs, delay the subsequent upcall until the page fault completes.

Finally, a user-level thread that has blocked in the kernel may still need to execute further in kernel mode when the I/O completes. If so, the kernel resumes the thread temporarily, until it either blocks again or reaches the point where it would leave the kernel. It is when the latter occurs that the kernel notifies the user level, passing the user-level thread's register state as part of the upcall.

3.2 Notifying the Kernel of User-Level Events Affecting Processor Allocation

The mechanism described in the last subsection is independent of the policy used by the kernel for allocating processors among address spaces. Reasonable allocation policies, however, must be based on the available parallelism in each address space. In this subsection, we show that this information can be efficiently communicated for policies that both respect priorities and guarantee that processors do not idle if runnable threads exist. These constraints are met by most kernel thread systems; as far as we know, they are not met by any user-level thread system built on top of kernel threads.

The key observation is that the user level thread system need not tell the kernel about every thread operation, but only about the small subset that can affect the kernel's processor allocation decision. By contrast, when kernel threads are used directly for parallelism, a processor traps to the kernel even when the best thread for it to run next—a thread that respects priorities while minimizing overhead and preserving cache context—is within the same address space.

In our system, an address space notifies the kernel whenever it makes a transition to a state where it has more runnable threads than processors, or more processors than runnable threads. Provided an application has extra threads to run and the processor allocator has not reassigned it additional processors, then all processors in the system must be busy. Creating more parallelism cannot violate the constraints. Similarly, if an application has notified the kernel that it has idle processors and the kernel has not taken them away, then there must be no other work in the system. The kernel need not be notified of additional idle processors. (An extension to this approach handles the situation where threads, rather than address spaces, have globally meaningful priorities).

Table III lists the kernel calls made by an address space on these state transitions. For example, when an address space notifies the kernel that it needs more processors, the kernel searches for an address space that has registered that has idle processors. If none are found, nothing happens, but the address space may eventually get a processor if one becomes idle in the future. These notifications are only hints: if the kernel gives an address space a processor that is no longer needed by the time it gets there, the address space simply returns the processor to the kernel with the updated information. Of course, the user-level thread system must serialize its notifications to the kernel, since ordering matters.

An apparent drawback to this approach is that applications may not be honest in reporting their parallelism to the operating system. This problem is not unique to multiprocessors: a dishonest or misbehaving program can

Table III: Scheduler Activation Upcall Points

Add more processors (additional # of processors needed)	
	Allocate more processors to this address space and start them running scheduler activations.
This processor is idle ()	
	Preempt this processor if another address space needs it.

consume an unfair proportion of resources on a multiprogrammed uniprocessor as well. In either kernel-level or user-level thread systems, multi-level feedback can be used to encourage applications to provide honest information for processor allocation decisions. The processor allocator can favor address spaces that use fewer processors and penalize those that use more. This encourages address spaces to give up processors when they are needed elsewhere, since the priorities imply that it is likely that the processors will be returned when they are needed. On the other hand, if overall the system has fewer threads than processors, the idle processors should be left in the address spaces most likely to create work in the near future, to avoid the overhead of processor reallocation when the work is created.

Many production uniprocessor operating systems do something similar. Average response time, and especially interactive performance, is improved by favoring jobs with the least remaining service, often approximated by reducing the priority of jobs as they accumulate service time. We expect a similar policy to be used in multiprogrammed multiprocessors to achieve the same goal; this policy could easily be adapted to encourage honest reporting of idle processors.

3.3 Critical Sections

One issue we have not yet addressed is that a user-level thread could be executing in a critical section at the instant when it is blocked or preempted.[1] There are two possible ill effects: poor performance (e.g., because other threads continue to test an application-level spin-lock held by the pre-empted thread) [28], and deadlock (e.g., the preempted thread could be holding the ready list lock; if so, deadlock would occur if the upcall attempted to place the preempted thread onto the ready list). Problems can occur even when critical sections are not protected by a lock. For example, FastThreads uses unlocked per-processor (really, per-activation) free lists of thread control blocks to improve latency [2]; accesses to these free lists also must be done atomically. *Prevention* and *recovery* are two approaches to dealing with the problem of inopportune preemption. With prevention, inopportune preemptions are avoided through the use of a scheduling and locking protocol between the kernel

[1] The need for critical sections would be avoided if we were to use wait-free synchronization [11]. Many commercial architectures, however, do not provide the required hardware support (we assume only an atomic test-and-set instruction); in addition, the overhead of wait-free synchronization can be prohibitive for protecting anything but very small data structures.

and the user level. Prevention has a number of serious drawbacks, particularly in a multiprogrammed environment. Prevention requires the kernel to yield control over processor allocation (at least temporarily) to the user-level, violating the semantics of address space priorities. Prevention is inconsistent with the efficient implementation of critical sections that we will describe in Section 4.3. Finally, in the presence of page faults, prevention requires "pinning" to physical memory all virtual pages that might be touched while in a critical section; identifying these pages can be cumbersome.

Instead, we adopt a solution based on recovery. When an upcall informs the user-level thread system that a thread has been preempted or unblocked, the thread system checks if the thread was executing in a critical section. (Of course, this check must be made before acquiring any locks). if so, the thread is continued temporarily via a user-level context switch. When the continued thread exits the critical section, it relinquishes control back to the original upcall, again via a user-level context switch. At this point, it is safe to place the user-level thread back on the ready list. We use the same mechanism to continue an activation if it was preempted in the middle of processing a kernel event.

This technique is free from deadlock. By continuing the lock holder, we ensure that once a lock is acquired, it is always eventually released, even in the presence of processor preemptions or page faults. Further, this technique supports arbitrary user-level spin-locks, since the user-level thread system is always notified when a preemption occurs, allowing it to continue the spinlock holder. Although correctness is not affected, processor time may be wasted spin-waiting when a spin-lock holder takes a page fault; a solution to this is to relinquish the processor after spinning for a while [4].

4. IMPLEMENTATION

We have implemented the design described in Section 3 by modifying Topaz, the native operating system for the DEC SRC Firefly multiprocessor workstation, and FastThreads, a user-level thread package.

We modified the Topaz kernel thread management routines to implement scheduler activations. Where Topaz formerly blocked, resumed, or preempted a thread, it now performs upcalls to allow the user level to take these actions (see Table II). In addition, we modified Topaz to do explicit allocation of processors to address spaces; formerly, Topaz scheduled threads obliviously to the address spaces to which they belonged. We also maintained object code compatibility; existing Topaz (and therefore UNIX) applications still run as before.

FastThreads was modified to process upcalls, to resume interrupted critical sections, and to provide Topaz with the information needed for its processor allocation decisions (see Table III).

In all, we added a few hundred lines of code to FastThreads and about 1200 lines to Topaz. (For comparison, the original Topaz implementation of kernel threads was over 4000 lines of code). The majority of the new Topaz code was concerned with implementing the processor allocation policy (discussed below), and not with scheduler activations *per se*.

Our design is "neutral" on the choice of policies for allocating processors to address spaces and for scheduling threads onto processors. Of course, some pair of policies had to be selected for our prototype implementation; we briefly describe these, as well as some performance enhancements and debugging considerations, in the subsections that follow.

4.1 Processor Allocation Policy

The processor allocation policy we chose is similar to the dynamic policy of Zahorjan and McCann [27]. The policy "space-shares" processors while respecting priorities and guaranteeing that no processor idles if there is work to do. Processors are divided evenly among the highest priority address spaces; if some address spaces do not need all of the processors in their share, those processors are divided evenly among the remainder. Space-sharing reduces the number of processor reallocations; processors are time-sliced only if the number of available processors is not an integer multiple of the number of address spaces (at the same priority) that want them.

Our implementation makes it possible for an address space to use kernel threads, rather than requiring that every address space use scheduler activations. Continuing to support Topaz kernel threads was necessary to preserve binary compatibility with existing possibly sequential) Topaz applications. In our implementation, address spaces that use kernel threads compete for processors in the same way as applications that use scheduler activations. The kernel processor allocator only needs to know whether each address space could use more processors or has some processors that are idle. (An application can be in neither state; for instance, if it has asked for a processor, received it, and has not asked for another processor yet). The interface described in Section 3.2 provides this information for address spaces that use scheduler activations; internal kernel data structures provide it for address spaces that use kernel threads directly. Processors assigned to address spaces using scheduler activations are handed to the user-level thread scheduler via upcalls; processors assigned to address spaces using kernel threads are handed to the original Topaz thread scheduler. As a result, there is no need for static partitioning of processors.

4.2 Thread Scheduling Policy

An important aspect of our design is that the kernel has no knowledge of an application's concurrency model or scheduling policy, or of the data structures used to manage parallelism at the user level. Each application is completely free to choose these as appropriate; they can be tuned to fit the application's needs. The default policy in FastThreads uses per-processor ready lists accessed by each processor in last-in-first-out order to improve cache locality; a processor scans for work if its own ready list is empty. This is essentially the policy used by Multilisp [10].

In addition, our implementation includes hysteresis to avoid unnecessary processor reallocations; an idle processor spins for a short period before notifying the kernel that it is available for reallocation.

4.3 Performance Enhancements

While the design as just described is sufficient to provide user-level functionality equivalent to that of kernel threads, there are some additional considerations that are important for performance.

The most significant of these relates to critical sections, described in Section 3.3 In order to provide temporary continuation of critical sections when a user-level thread is preempted (or when it blocks in the kernel and can be resumed), the user-level thread system must be able to check whether the thread was holding a lock. One way to do this is for the thread to set a flag when it enters a critical section, clear the flag when it leaves, and then check to see if it is being continued. The check is needed so that the thread being temporarily continued will relinquish the processor to the original upcall when it reaches a safe place. Unfortunately, this imposes overhead on lock acquisition and release whether or not a preemption or page fault occurs, even though these events are infrequent. Latency is particularly important since we use these continuable critical sections in building our user-level thread system.

We adopt a different solution that imposes no overhead in the common case; a related technique was used on a uniprocessor in the Trellis/Owl garbage collector [17]. We make an exact copy of every low-level critical section. We do this by delimiting, with special assembler labels, each critical section in the C source code for the user-level thread package; we then post-process the compiler-generated assembly code to make the copy. This would also be straightforward to do given language and compiler support. At the end of the copy, but not the original version of the critical section, we place code to yield the processor back to the resumer. Normal execution uses the original code. When a preemption occurs, the kernel starts a new scheduler activation to notify the user-level thread system; this activation checks the preempted thread's program counter to see if it was in one of these critical sections, and if so, continues the thread at the corresponding place in the copy of the critical section. The copy relinquishes control back to the original upcall at the end of the critical section. Because normal execution uses the original code, and this code is exactly the same as it would be if we were not concerned about preemptions, there is *no* impact on lock latency in the common case. (In our implementation, occasionally a procedure call must be made from within a critical section. In this case, we bracket the call, but not the straight line path, with the setting and clearing of an explicit flag).

A second significant performance enhancement relates to the management of scheduler activations. Logically, a new scheduler activation is created for each upcall. Creating a new scheduler activation is not free, however, because it requires data structures to be allocated and initialized. Instead, discarded scheduler activations can be cached for eventual reuse. The user-level thread system can recycle an old scheduler activation by returning it to the kernel as soon as the user-level thread it had been running is removed from its context: in the case of preemption, after processing the upcall that notifies the user level of the preemption; in the case of blocking in the kernel, after processing the upcall that notifies the user level that resumption is possible. A similar optimization is used in many kernel thread implementations: kernel threads, once created, can be cached when destroyed to speed future thread creations [13].

Further, discarded scheduler activations can be collected and returned to the kernel in bulk, instead of being returned one at a time. Ignoring the occasional bulk deposit of discards, our system makes the same number of application-kernel boundary crossings on I/O or processor preemption as a traditional kernel thread system. In a kernel thread system, one crossing is needed to start an I/O, and another is needed when the I/O completes. The same kernel boundary crossings occur in our system.

4.4 Debugging Considerations

We have integrated scheduler activations with the Firefly Topaz debugger. There are two separate environments, each with their own needs: debugging the user-level thread system and debugging application code running on top of the thread system.

Transparency is crucial to debugging—the debugger should have as little effect as possible on the sequence of instructions being debugged. The kernel support we have described informs the user-level thread system of the state of each of its physical processors, but this is inappropriate when the thread system itself is being debugged. Instead, the kernel assigns each scheduler activation being debugged a *logical* processor; when the debugger stops or single-steps a scheduler activation, these events do not cause upcalls into the user-level thread system.

Assuming the user-level thread system is working correctly, the debugger can use the facilities of the thread system to stop and examine the state of application code running in the context of a user-level thread [18].

5. PERFORMANCE

The goal of our research is to combine the functionality of kernel threads with the performance and flexibility advantages of managing parallelism at the user level within each application address space. The functionality and flexibility issues have been addressed in previous sections. In terms of performance, we consider three questions. First, what is the cost of user-level thread operations (e.g., fork, block and yield) in our system? Second, what is the cost of communication between the kernel and the user level (specifically, of upcalls)? Third, what is the overall effect on the performance of applications?

5.1 Thread Performance

The cost of user-level thread operations in our system is essentially the same as those of the FastThreads package running on the Firefly *prior* to our work—that is, running on top of Topaz kernel threads, with the associated poor system integration. Table IV adds the performance of our system to the data for original FastThreads, Topaz kernel threads, and Ultrix processes contained in Table I. Our system preserves the order of

Table IV. Thread Operation Latencies (μsec.)

Operation	FastThreads on Topaz Threads	FastThreads on Scheduler Activations	Topaz threads	Ultrix processes
Null Fork	34	37	948	11300
Signal-Wait	37	42	441	1840

magnitude advantage that user-level threads offer over kernel threads. There is a 3 μsec. degradation in Null Fork relative to original FastThreads, which is due to incrementing and decrementing the number of busy threads and determining whether the kernel must be notified. (This could be eliminated for a program running on a uniprogrammed machine or running with sufficient parallelism that it can inform the kernel that it always wants as many processors as are available). There is a 5 μsec. degradation in Signal-Wait, which is due to this factor plus the cost of checking whether a preempted thread is being resumed (in which case extra work must be done to restore the condition codes). Although still an order of magnitude better than kernel threads, our performance would be significantly worse without a zero-overhead way of marking when a lock is held (see Section 4.3). Removing this optimization from FastThreads yielded a Null Fork time of 49 μsec. and a Signal-Wait time of 48 μsec. (The Null Fork benchmark has more critical sections in its execution path than does Signal-Wait.)

5.2 Upcall Performance

Thread performance (Section 5.1) characterizes the frequent case when kernel involvement is not necessary. Upcall performance—the infrequent case—is important, though, for several reasons. First, it helps determine the “break-even” point, the ratio of thread operations that can be done at user level to those that require kernel intervention, needed for user-level threads to begin to outperform kernel threads. if the cost of blocking or preempting a user-level thread in the kernel using scheduler activations is similar to the cost of blocking or preempting a kernel thread, then scheduler activations could be practical even on a uniprocessor. Further, the latency between when a thread is preempted and when the upcall reschedules it determines how long other threads running in the application may have to wait for a critical resource held by the preempted thread.

When we began our implementation, we expected our upcall performance to be commensurate with the overhead of Topaz kernel thread operations. Our implementation is considerably slower than that. One measure of upcall performance is the time for two user-level threads to signal and wait through the kernel; this is analogous to the Signal-Wait test in Table IV, except that the synchronization is forced to be in the kernel. This approximates the overhead added by the scheduler activation machinery of making and completing an I/O request or a page fault. The Signal-Wait time is 2.4 milliseconds, a factor of five worse than Topaz threads.

We see nothing inherent in scheduler activations that is responsible for this difference, which we attribute to two implementation issues. First, because we built scheduler activations as a quick modification to the existing implementation of the Topaz kernel thread system, we must maintain more state, and thus have more overhead, than if we had designed that portion of the kernel from scratch. As importantly, much of the Topaz thread system is written in carefully tuned assembler; our kernel implementation is entirely in Modula-2+. For comparison, Schroeder and Burrows [19] reduced SRC RPC processing costs by over a factor of four by recoding Modula-2+ in assembler. Thus, we expect that, if tuned, our upcall performance would be commensurate with Topaz kernel thread performance. As a result, the application performance measurements in the next section are somewhat worse than what might be achieved in a production scheduler activation implementation.

5.3 Application Performance

To illustrate the effect of our system on application performance, we measured the same parallel application using Topaz kernel threads, original FastThreads built on top of Topaz threads, and modified FastThreads running on scheduler activations. The application we measured was an O(N log N) solution to the N-body problem [3]. The algorithm constructs a tree representing the center of mass of each portion of space and then traverses portions of the tree to compute the force on each body. The force exerted by a cluster of distant masses can be approximated by the force that they would exert if they were all at the center of mass of the cluster.

Depending on the relative ratio of processor speed to available memory, this application can be either compute or I/O bound. We modified the application to manage a part of its memory explicitly as a buffer cache for the application's data. This allowed us to control the amount of memory used by the application; a small enough problem size was chosen so that the buffer cache always fit in our Firefly's physical memory. As a further simplification, threads that miss in the cache simply block in the kernel for 50 msec.; cache misses would normally cause a disk access. (Our measurements were qualitatively similar when we took contention for the disk into account; because the Firefly's floating point performance and physical memory size are orders of magnitude less than current generation systems, our measurements are intended to be only illustrative.) All tests were run on a six processor CVAX Firefly.

First, we demonstrate that when the application makes minimal use of kernel services, it runs as quickly on our system as on original FastThreads and much faster than if Topaz threads were used. Figure 2 graphs the application's speedup versus the number of processors for each of the three systems when the system has enough memory so that there is negligible I/O and there are no other applications running. (Speedup is relative to a sequential implementation of the algorithm).

With one processor, all three systems perform worse than the sequential implementation, because of the added overhead of creating and synchronizing threads to parallelize the application. This overhead is greater for Topaz kernel threads than for either user-level thread system.

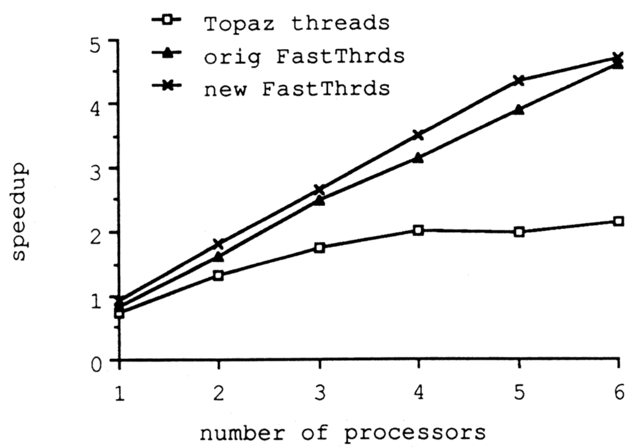

Fig. 2. Speedup of N-Body application versus number of processors, 100% of memory available.

As processors are added, the performance with Topaz kernel threads initially improves and then flattens out. In Topaz, a thread can acquire and release an application lock on a critical section without trapping to the kernel, provided there is no contention for the lock. If a thread tries to acquire a busy lock, however, the thread will block in the kernel and be rescheduled only when the lock is released. Thus, Topaz lock overhead is much greater in the presence of contention. The good speedup attained by both user-level thread systems shows that the application has enough parallelism; it is the overhead of kernel threads that prevents good performance. We might be able to improve the performance of the application when using kernel threads by restructuring it so that its critical sections are less of a bottleneck or perhaps by spinning for a short time at user level if the lock is busy before trapping to the kernel [12]; these optimizations are less crucial if the application is built with user-level threads.

The performance of original FastThreads and our system diverges slightly with four or five processors. Even though no other applications were running during our tests, the Topaz operating system has several daemon threads which wake up periodically, execute for a short time, and then go back to sleep. Because our system explicitly allocates processors to address spaces, these daemon threads cause preemptions only when there are no idle processors available; this is not true with the native Topaz scheduler, which controls the kernel threads used as virtual processors by original FastThreads. When the application tries to use all of the processors of the machine (in this case, six processors), the number of preemptions for both user-level thread systems is similar. (The preemptions have only a small impact on the performance of original FastThreads because of their short duration).

Next, we show that when the application requires kernel involvement because it does I/O, our system performs better than either original Fast-

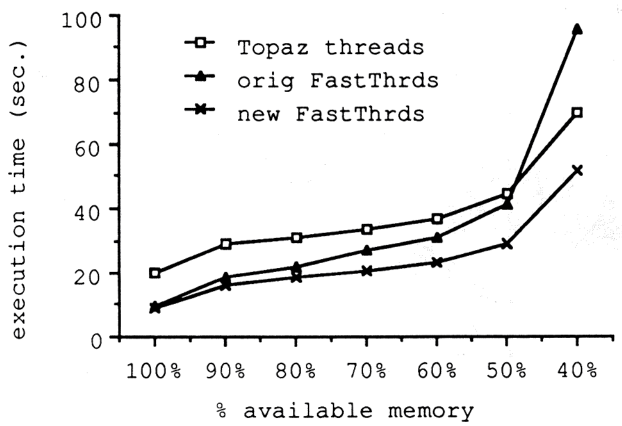

Fig. 3. Execution time of N-Body application versus amount of available memory, 6 processors.

Threads or Topaz threads. Figure 3 graphs the application's execution time on six processors as a function of the amount of available memory.

For all three systems, performance degrades slowly at first, and then more sharply once the application's working set does not fit in memory. However, application performance with original FastThreads degrades more quickly than with the other two systems. This is because when a user-level thread blocks in the kernel, the kernel thread serving as its virtual processor also blocks, and thus the application loses that physical processor for the duration of the I/O. The curves for modified FastThreads and for Topaz threads parallel each other because both systems are able to exploit the parallelism of the application to overlap some of the I/O latency with useful computation. As in Figure 2, though, application performance is better with modified FastThreads than with Topaz because most thread operations can be implemented without kernel involvement.

Finally, while Figure 3 shows the effect on performance of application-induced kernel events, multiprogramming causes system-induced kernel events that result in our system having better performance than either original FastThreads or Topaz threads. To test this, we ran two copies of the N-body application at the same time on a six processor Firefly and then averaged their execution times. Table V lists the resulting speedups for each system; note that a speedup of three would be the maximum possible.

Table V shows that application performance with modified FastThreads is good even in a multiprogrammed environment; the speedup is within 5% of that obtained when the application ran uniprogrammed on three processors. This small degradation is about what we would expect from bus contention and the need to donate a processor periodically to run a kernel daemon thread. In contrast multiprogrammed performance is much worse with either original FastThreads or Topaz threads, although for different reasons. When applications using original FastThreads are

Table V. Speedup of N-Body application, Multiprogramming Level = 2, 6 Processors, 100% of Memory Available

Topaz Threads	Original FastThreads	New FastThreads
1.29	1.26	2.45

multiprogrammed, the operating system time-slices the kernel threads serving as virtual processors; this can result in physical processors idling waiting for a lock to be released while the lock holder is descheduled. Performance is worse with Topaz threads than with our system because common thread operations are more expensive. In addition, because Topaz does not do explicit processor allocation, it may end up scheduling more kernel threads from one address space than from the other; Figure 2 shows, however, that performance flattens out for Topaz threads when more than three processors are assigned to the application.

While the Firefly is an excellent vehicle for constructing proof-of-concept prototypes, its limited number of processors makes it less than ideal for experimenting with significantly parallel applications or with multiple, multiprogrammed parallel applications. For this reason, we are implementing scheduler activations in C Threads and Mach; we are also porting Amber [6], a programming system for a network of multiprocessors, onto our Firefly implementation.

6. RELATED iDEAS

The two systems with goals most closely related to our own—achieving properly integrated user-level threads through improved kernel support—are Psyche [20] and Symunix [9]. Both have support for NUMA multiprocessors as a primary goal: Symunix in a high-performance parallel UNIX implementation, and Psyche in the context of a new operating system.

Psyche and Symunix provide "virtual processors" as described in Sections 1 and 2, and augment these virtual processors by defining software interrupts that notify the user level of some kernel events. (Software interrupts are like upcalls, except that all interrupts on the same processor use the same stack and thus are not reentrant). Psyche has also explored the notion of multimodel parallel programming in which user-defined threads of various kinds, in different address spaces, can synchronize while sharing code and data.

While Psyche, Symunix, and our own work share similar goals, the approaches taken to achieve these goals differ in several important ways. Unlike our work, neither Psyche nor Symunix provides the exact functionality of kernel threads with respect to I/O, page faults and multiprogramming; further, the performance of their user-level thread operations can be compromised. We discussed some of the reasons for this in Section 2: these systems notify the user level of *some but not all* of the kernel events that affect the address space. For example, neither Psyche nor Symunix notify the user level when a preempted virtual processor is rescheduled. As a result, the user-level thread system does not know how many processors it has or what user threads are running on those processors.

Both Psyche and Symunix provide shared writable memory between the kernel and each application, but neither system provides an efficient mechanism for the user-level thread system to notify the kernel when its processor allocation needs to be reconsidered. The number of processors needed by each application could be written into this shared memory, but that would give no efficient way for an application that needs more processors to know that some other application has idle processors.

Applications in both Psyche and Symunix share synchronization state with the kernel in order to avoid preemption at inopportune moments (e.g., while spin-locks are being held). In Symunix, the application sets and later clears a variable shared with the kernel to indicate that it is in a critical section; in Psyche, the application checks for an imminent preemption before starting a critical section. The setting, clearing, and checking of these bits adds to lock latency, which constitutes a large portion of the overhead when doing high-performance user-level thread management [2]. By contrast, our system has no effect on lock latency unless a preemption actually occurs. Furthermore, in these other systems the kernel notifies the application of its intention to preempt a processor *before* the preemption actually occurs; based on this notification, the application can choose to place a thread in a "safe" state and voluntarily relinquish a processor. This mechanism violates the constraint that higher priority threads are always run in place of lower priority threads.

Gupta et al. [9a] share our goal of maintaining a one-to-one correspondence between physical processors and execution contexts for running user-level threads. When a processor preemption or I/O completion results in there being more contexts than processors, Gupta et al.'s kernel time-slices contexts until the application reaches a point where it is safe to suspend a context. Our kernel eliminates the need for time-slicing by notifying the application thread system of the event while keeping the number of contexts constant.

Some systems provide asynchronous kernel I/O as a mechanism to solve some of the problems with user-level thread management on multiprocessors [9, 25]. Indeed, our work has the flavor of an asynchronous I/O system: when an I/O request is made, the processor is returned to the application, and later, when the I/O completes, the application is notified. There are two major differences between our work and traditional asynchronous I/O systems, though. First, and most important, scheduler activations provide a single uniform mechanism to address the problems of processor preemption, I/O, and page faults. Relative to asynchronous I/O, our approach derives conceptual simplicity from the fact that all interaction with the kernel is synchronous from the perspective of a single scheduler activation. A scheduler activation that blocks in the kernel is replaced with a new scheduler activation when the awaited event occurs. Second, while asynchronous I/O schemes may require significant changes to both application and kernel code, our scheme leaves the structure of both the user-level thread system and the kernel largely unchanged.

Finally, parts of our scheme are related in some ways to Hydra [26], one of the earliest multiprocessor operating systems, in which scheduling policy was moved out of the kernel. However, in Hydra, this separation came at a performance cost because policy decisions required communication through the kernel to a scheduling policy server, and then back to the kernel to implement a context switch. In our system, an application can set its own policy for scheduling its threads onto its processors, and can implement this policy without trapping to the kernel. Longer-term processor allocation decisions in our system are the kernel's responsibility, although as in Hydra, this could be delegated to a distinguished application-level server.

7. SUMMARY

Managing parallelism at the user level is essential to high-performance parallel computing, but kernel threads or processes, as provided in many operating systems, are a poor abstraction on which to support this. We have described the design, implementation and performance of a kernel interface and a user-level thread package that together combine the performance of user-level threads (in the common case of thread operations that can be implemented entirely at user level) with the functionality of kernel threads (correct behavior in the infrequent case when the kernel must be involved). Our approach is based on providing each application address space with a virtual multiprocessor in which the application knows exactly how many processors it has and exactly which of its threads are running on those processors. Responsibilities are divided between the kernel and each application address space:

—Processor allocation (the allocation of processors to address spaces) is done by the kernel.

—Thread scheduling (the assignment of an address space's threads to its processors) is done by each address space.

—The kernel notifies the address space thread scheduler of every event affecting the address space.

—The address space notifies the kernel of the subset of user-level events that can affect processor allocation decisions.

The kernel mechanism that we use to implement these ideas is called *scheduler activations*. A scheduler activation is the execution context for vectoring control from the kernel to the address space on a kernel event. The address space thread scheduler uses this context to handle the event, e.g., to modify user-level thread data structures, to execute user-level threads, and to make requests of the kernel. While our prototype implements threads as the concurrency abstraction supported at the user level, scheduler activations are not linked to any particular model; scheduler activations can support any user-level concurrency model because the kernel has no knowledge of user-level data structures.

ACKNOWLEDGMENTS

We would like to thank Andrew Black, Mike Burrows, Jan Edler, Mike Jones, Butler Lampson, Tom LeBlanc, Kai Li, Brian Marsh, Sape Mullender, Dave Redell, Michael Scott, Garret Swart, and John Zahorjan for their helpful comments. We would also like to thank the DEC Systems Research Center for providing us with their Firefly hardware and software.

REFERENCES

1. AGHA, G. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, Mass. 1986.
2. ANDERSON, T., LAZOWSKA, E., AND LEVY, H. The performance implications of thread management alternatives for shared memory multiprocessors. *IEEE Trans. Comput. 38*, 12 (Dec. 1989), 1631–1644. Also appeared in *Proceedings of the 1989 ACM SIGMETRICS and Performance '59 Conference on Measurement and Modeling of Computer Systems* (Oakland, Calif., May 1989), pp. 49–60.
3. BARNES, J., AND HUT, P. A hierarchical O(N log N) force-calculation algorithm. *Nature 324* (1986), 446–449.
4. BIRRELL, A., GUTTAG, J., HORNING, J., AND LEVIN, R. Synchronization primitives for a multiprocessor: A formal specification. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles* (Austin, Tex., Nov.1987), pp. 94–102.
5. BLACK, D. Scheduling support for concurrency and parallelism in the Mach operating system. *IEEE Comput. Mag. 23*, 5 (May 1990), 35–43.
6. CHASE, J., AMADOR, F., LAZOWSKA, E., LEVY, H., AND LITTLEFIELD, R. The Amber system: Parallel programming on a network of multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles* (Litchfield Park, Ariz., Dec.1989), pp. 147–158.
7. CHERITON, D. The V distributed system. *Commun. ACM. 31*, 3 (Mar. 1988), 314–333.
8. DRAVES, R., AND COOPER, E. C Threads. Tech. Rep. CMU-CS-88-154, School of Computer Science, Carnegie Mellon Univ., June 1988.
9. EDLER, J., LIPKIS, J., AND SCHONBERG, E. Process management for highly parallel UNIX systems. In *Proceedings of the USENIX workshop on UNIX and Supercomputers* (Sept. 1988), pp. 1–17

9A. GUPTA, A., TUCKER, A., AND STEVENS, L. Making effective use of shared-memory multiprocessors: The process control approach. Tech. Rep. CSL-TR-91-475A, Computer Systems Laboratory, Stanford Univ., July 1991.

10. HALSTEAD, R. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst. 7*, 4 (Oct. 1985), 501–538.
11. HERLIHY, M. A methodology for implementing highly concurrent data structures. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Seattle, Wash., Mar. 1990), pp. 197–206.
12. KARLIN, A., LI, K., MANASSE, M., AND OWICKI, S. Empirical studies of competitive spinning for a shared-memory multiprocessor. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles* (Pacific Grove, Calif., Oct. 1991), pp. 41–55.
13. LAMPSON, B., AND REDELL, D. Experiences with processes and monitors in Mesa. *Commun. ACM. 23*, 2 (Feb. 1980), 104–117.
14. LO, S.-P., AND GLIGOR, V. A comparative analysis of multiprocessor scheduling algorithms. In *Proceedings of the 7th International Conference on Distributed Computing Systems* (Sept. 1987), pp. 356–363.
15. MARSH, B., SCOTT, M., LEBLANC, T., AND MARKATOS, E. First-class user-level threads. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles* (Pacific Grove, Calif., Oct. 1991), pp. 110–121.
16. MOELLER-NIELSEN, P., AND STAUNSTRUP, J. Problem-heap: A paradigm for multiprocessor algorithms. *Parallel Comput. 4*, 1 (Feb. 1987), 63–74.
17. MOSS, J., AND KOHLER, W. Concurrency features for the Trellis/Owl language. In *Proceedings of European Conference on Object-Oriented Programming 1987*

(ECOOP 87) (June 1987), pp. 171–180.
18. REDELL, D. Experience with Topaz teledebugging. In *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging* (Madison, Wisc., May 1988), pp. 35–44.
19. SCHROEDER, M., AND BURROWS, M. Performance of Firefly RPC. *ACM Trans. Comput. Syst. 8*, 1 (Feb. 1990), 1–17.
20. SCOTT, M., LEBLANC, T., AND MARSH, B. Multi-model parallel programming in Psyche. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Mar. 1990), pp. 70–78.
21. TEVANIAN, A., RASHID, R., GOLUB, D., BLACK, D., COOPER, E., AND YOUNG, M. Mach Threads and the Unix Kernel: The battle for control. In *Proceedings of the 1987 USENIX Summer Conference* (1987), pp. 185–197.
22. THACKER, C., STEWART, L., AND SATTERTHWAITE, JR., E. Firefly: A multiprocessor workstation. *IEEE Trans. Comput. 37*, 8 (Aug. 1988), 909–920.
23. TUCKER, A., AND GUPTA, A. Process control and scheduling issues for multiprogrammed shared memory multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles* (Litchfield Park, Ariz., Dec. 1989), pp. 159–166.
24. VANDEVOORDE, M., AND ROBERTS, E. WorkCrews: An abstraction for controlling parallelism. *Int. J. Parallel Program. 17*, 4 (Aug. 1988), 347–366.
25. WEISER, M., DEMERS, A., AND HAUSER, C. The portable common runtime approach to interoperability. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles* (Litchfield Park, Ariz., Dec. 1989), pp. 114–122.
26. WULF, W., LEVIN, R., AND HARBISON, S. *Hydra/C.mmp: An Experimental Computer System*. McGraw-Hill, New York, 1981.
27. ZAHORJAN, J., AND MCCANN, C. Processor scheduling in shared memory multiprocessors. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Boulder, Colo., May 1990), pp. 214–225.
28. ZAHORJAN, J., LAZOWSKA, E., AND EAGER, D. The effect of scheduling discipline on spin overhead in shared memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst. 2*, 2 (Apr. 1991), 180–198.

Received June 1991; revised August 1991; accepted September 1991