TECHNICAL REPORT 94-02

A Framework for Minimizing Thread Management Overhead Based on Asynchronous Cooperation between User and Kernel Schedulers

Shigekazu Inohara and Takashi Masuda

January 19, 1994

DEPARTMENT OF INFORMATION SCIENCE
FACULTY OF SCIENCE, UNIVERSITY OF TOKYO

7–3–1 HONGO, BUNKYO-KU TOKYO, 113 JAPAN

TITLE
A Framework for Minimizing Thread Management Overhead Based on Asynchronous Cooperation between User and Kernel Schedulers

AUTHORS
Shigekazu Inohara and Takashi Masuda

KEY WORDS AND PHRASES
multiprogrammed multiprocessor system, shared-memory multiprocessor, operating system, thread management, thread switching overhead

ABSTRACT
The performance of thread mechanism is dominated primarily by two kinds of thread-switching overheads: vertical switching (user/kernel domain switching) and horizontal switching (context switching between threads). Ideally, vertical switchings should occur only when system calls or external interrupts are invoked. Horizontal switchings within a user address space should occur only when threads synchronize with one another. Horizontal switchings in the kernel should occur only when processors are redistributed among user address spaces. Existing thread mechanisms, however, do not function ideally in a multiprogrammed system because (1) user-level programs determine how many threads to use, (2) the kernel does not inform the user level of which nor how many threads are actually being assigned processors, and/or (3) interaction between the kernel and user-level schedulers is synchronous. This paper presents a thread mechanism that minimizes the thread-switching overhead by avoiding these three problems; the kernel scheduler determines how many threads to use in each user address space, the kernel scheduler lets user-level schedulers know which threads are actually being assigned processors, and all interaction between the kernel scheduler and user-level schedulers is asynchronous. Employing these three ideas, the proposed mechanism minimizes thread switchings *both in the kernel and in the user level*, under the assumption that threads are managed without off-line information on applications to run.

REPORT DATE January 19, 1994	**WRITTEN LANGUAGE** English
TOTAL NO. OF PAGES 24	**NO. OF REFERENCES** 22

ANY OTHER IDENTIFYING INFORMATION OF THIS REPORT
Submitted to IEEE Transactions on Parallel and Distributed Systems

DISTRIBUTION STATEMENT
First Issue 40 copies

SUPPLEMENTARY NOTES

DEPARTMENT OF INFORMATION SCIENCE
Faculty of Science, University of Tokyo
7-3-1 Hongo, Bunkyo-ku Tokyo, 113 Japan

A Framework for Minimizing Thread Management Overhead Based on Asynchronous Cooperation between User and Kernel Schedulers

Shigekazu Inohara and Takashi Masuda

January 20, 1994

Abstract

The performance of thread mechanism is dominated primarily by two kinds of thread-switching overheads: vertical switching (user/kernel domain switching) and horizontal switching (context switching between threads). Ideally, vertical switchings should occur only when system calls or external interrupts are invoked. Horizontal switchings within a user address space should occur only when threads synchronize with one another. Horizontal switchings in the kernel should occur only when processors are redistributed among user address spaces. Existing thread mechanisms, however, do not function ideally in a multiprogrammed system because (1) user-level programs determine how many threads to use, (2) the kernel does not inform the user level of which nor how many threads are actually being assigned processors, and/or (3) interaction between the kernel and user-level schedulers is synchronous. This paper presents a thread mechanism that minimizes the thread-switching overhead by avoiding these three problems; the kernel scheduler determines how many threads to use in each user address space, the kernel scheduler lets user-level schedulers know which threads are actually being assigned processors, and all interaction between the kernel scheduler and user-level schedulers is asynchronous. Employing these three ideas, the proposed mechanism minimizes thread switchings both in the kernel and in the user level*, under the assumption that threads are managed without off-line information on applications to run.*

Index Terms: asynchronous interaction, multiprogrammed systems, operating systems, parallel computing, shared memory multi-processors, scheduler activations, thread management.

1 Introduction

The primary role of a thread mechanism is to virtualize the processors of a shared-memory multiprocessor system and provide user address spaces with logical multiprocessors (multiple threads of control). With this system, parallel applications in user address spaces become independent of the actual number of processors in the system. A thread mechanism is thus especially important on a multiprogrammed system, where a single application is not allowed to monopolize all the processors in the system. In such an environment, implementing a low-overhead thread mechanism has a large impact on the performance of all the applications in the system.

A thread management system has three sources of overhead: thread creation, thread switching, and thread termination. Among the three, thread-switching overhead dominates the overall overhead of thread management. This is because thread creation and thread termination are implemented as manipulations (allocation or deallocation) of a data structure called thread control block (TCB) followed by thread switching, where the manipulation of TCB can be implemented with almost no overhead if we have sufficient capacity of the TCB table (this technique is known as "thread buffering").

Thread switchings are classified in two different types: vertical switching and horizontal switching. The first is the switching of a processor's protection domain between a user address space from/to the operating system kernel. Horizontal switching is the context switching of a processor from one thread to another. Ideally, vertical switchings should occur only when system calls or external interrupts are invoked. Horizontal switchings within a user address space should occur only when threads synchronize with one another. Horizontal switchings in the kernel should occur only when processors are redistributed among user address spaces.

Existing thread mechanisms, however, do not function ideally in a multiprogrammed system. Most existing thread mechanisms implement threads solely with a scheduler in the kernel [1, 2, 3, 4] or in user-level schedulers on top of the kernel scheduler [5, 6, 7, 8, 9, 10, 11, 12, 13] (let us call threads implemented by the kernel scheduler *k-threads*, and those implemented by user-level schedulers *u-threads* in the rest of this paper). Regardless of their different implementation schemes, they share three characteristics that cause unnecessary vertical and horizontal switchings on multiprogrammed systems: (1) user-level programs determine how many k-threads to use, (2) the kernel does not inform the user level of which nor how many k-threads are actually being assigned processors, and (3) interaction between the kernel and user-level schedulers is synchronous. The combination of (1) and (2) implies that there is no mechanism for properly maintaining the number of k-threads. The number of runnable k-threads is hence larger than the number of processors (otherwise some processors remain idle), which results in extra horizontal and vertical switchings when u-threads are horizontally switched. An uncommunicative kernel (characteristic (2)) also forces user-level programs to use sleeping locks in place of spin locks for synchronizing u-threads, resulting in extra horizontal switchings at the user level.

Extra switchings occur even in a system with "scheduler activations," which is a mechanism that properly maintains the number of k-threads [14]. In this system, the characteristic (3) still holds for two cases. First, information on the activities of k-threads is conveyed from the kernel schedulers to the user-level schedulers through a synchronous interface, which causes extra vertical switchings. Second, user-level schedulers react immediately to an update of the information sent from the kernel scheduler, which sometimes causes extra horizontal switchings in the user level.

This paper presents a thread mechanism that minimizes the thread-switching overhead by avoiding these three problems. First, the kernel scheduler determines how many k-threads to use in each user address space. The kernel scheduler is allowed to create, suspend, resume, and terminate k-threads at will, thereby maintaining equal numbers of runnable k-threads and of processors in each user address space. This minimizes the number of horizontal switchings in the kernel scheduler. Second, the kernel scheduler lets user-level schedulers know which k-threads are actually being assigned processors. With this information on the scheduling status of k-threads, user-level schedulers can avoid extra switchings in user address spaces, and can implement conventional u-threads on top of our rather extraordinary k-threads. Third, all interaction between the kernel scheduler and user-level schedulers is asynchronous. More specifically, the information on the scheduling status of k-threads is passed through an asynchronous interface using a shared memory area between user address spaces and the kernel. In addition, user-level schedulers react to the update of the information in a lazy way. The *asynchronous interaction between the kernel scheduler and user-level schedulers* is the most important concept behind the design of our thread mechanism, because synchronization between the kernel and user-level schedulers necessarily involves vertical and horizontal switchings of threads. Employing these three ideas avoids all unnecessary horizontal switchings in the user-level scheduler as well as all unnecessary vertical switchings.

As a result, thread switchings are minimized *both in the kernel and in the user level*, under the assumption that the thread mechanism manages threads without off-line information on applications to run. That is, horizontal switchings in the kernel scheduler occur only when the processor resources is redistributed among user address spaces. Horizontal switchings in user-level schedulers occur only when they are requested by application programs. Vertical switchings occur only when application programs invoke system calls and when interrupts arrive. These are the minimum switchings, which have not been accomplished by any other existing thread mechanism. At the same time, our scheme does not incur the cost of each switching as we show in Section 3.

In the next section, we present the structure of our thread mechanism in detail. Section 3 shows that the proposed thread mechanism minimizes the overheads of vertical and horizontal switchings. Section 4 presents the result of experiments that compare the performance of our thread mechanism with other systems, and we conclude this paper in Section 5.

2 Framework of Cooperation between User and Kernel

The three ideas for minimizing thread management overhead described above, k-threads created and terminated by the kernel scheduler, the passing of information on activities of k-threads, and asynchronous interaction between the kernel scheduler and user-level schedulers, are implemented as the following mechanisms.

Automatic creation and termination of k-threads: The kernel scheduler controls when and in which user address space k-threads are created, suspended, resumed, and terminated. This is opposite to conventional systems, where user-level programs must control k-threads explicitly with system calls that increase the number of vertical switchings.

Shared memory between the kernel and user address spaces: Each user-level scheduler shares a part of its address space with the kernel. The primary purpose of the shared memory area, called communication area or c-area, is to enable user-level programs to monitor and control the activities of k-threads without any vertical switchings.

Asynchronous user-level schedulers: An application program and a user-level scheduler coexist in each user address space. The user-level schedulers in our thread mechanism are, using the information provided through the c-area, responsible for recovering u-threads whose k-threads are suspended or terminated automatically by the kernel scheduler. The user-level schedulers refer to the c-area asynchronously to the updates of the c-area when they require the information.

Event notifiers: Some applications inherently require synchronous interaction with the kernel. For example, some applications require the notification of kernel-level events such as "I/O done" and "timer expired." Notifiers are provided for this purpose. Using the same implementation scheme as k-threads above, the number of vertical switchings for event notification is minimized.

2.1 Automatic Creation and Termination of K-threads

In conventional thread mechanisms, k-threads are created one by one through explicit system calls issued by user-level schedulers. Similarly, the k-threads are terminated through explicit system calls. The k-threads in our thread mechanism, on the other hand, are created and terminated by the kernel scheduler. What a user-level scheduler does is only to tell (through the c-area, as we describe below) the kernel scheduler how many u-threads can run, and the kernel scheduler responds to the request by supplying k-threads afterwards, *i.e.*, asynchronously. No system call is thus involved in controlling the action of our k-threads [1]. Another notable difference between the two k-thread mechanisms is that the proposed one does *not* assure any fairness in the scheduling of k-threads, while conventional thread mechanisms assure some fairness. A schematic view of the lives of conventional k-threads and our k-threads is shown in Fig. 1.

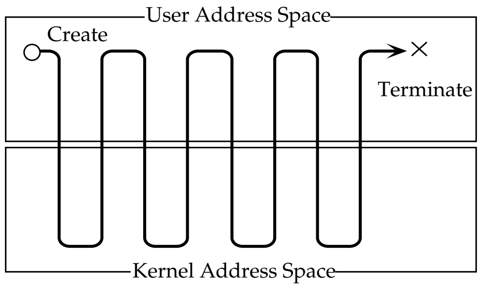
The design of our k-threads allows the kernel scheduler to schedule only as many runnable k-threads as processors so that the horizontal switching of k-threads in the kernel scheduler occurs only to balance the usage of the processor resources among user address spaces, and never occurs within an individual user address space. This is accomplished by coupling the creation, suspension, resumption, and termination of k-threads with the allocation and deallocation of processors. A k-thread in a user address space is created or resumed from the suspended state when and only when the kernel scheduler allocates a processor to the user address space. A k-thread is suspended or terminated when and only when the kernel deallocates a processor from the user address space.

2.2 Asynchronous Interaction between the Kernel and User-level Schedulers

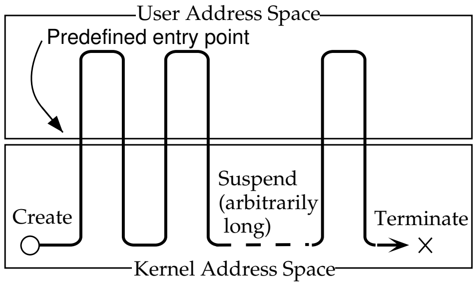
Since user-level schedulers cannot predict when automatic creation, suspension, resumption, and termination of k-threads will occur, the implementation of u-threads on top of our k-threads is not a simple task. We thus provide the c-area (Fig. 2), through which user-level schedulers can implement conventional u-threads by monitoring and controlling the activities of k-threads.

The kernel scheduler passes the current scheduling status of k-threads to user-level schedulers: which k-threads are currently running, what the final states of suspended k-threads are, and which k-threads have been terminated. Each element of the array c_kthread corresponds to a k-thread. The c_registers field holds the final processor state of the k-thread, and c_status indicates the current scheduling status of

[1] When the parallelism of a user-level program suddenly drops below the number of k-threads allocated, the user-level program can voluntarily relinquish unnecessary k-threads with kt_terminate(), which is the only k-thread-related system call.



(a) Conventional k-threads



(b) Proposed k-threads

Fig. 1. The life of a k-thread.

```
struct {
  struct {
    struct registers_t                  c_registers;      /* kernel-->user */
    enum { RUNNING, BLOCKED, SUSPENDED,
           RECOVERING, RECOVERED }      c_status;         /* kernel<->user */
  } c_kthread[MAX_KTHREAD];

  int c_max_parallelism;                                  /* kernel<--user */
} c;
```

Fig. 2. The c-area.

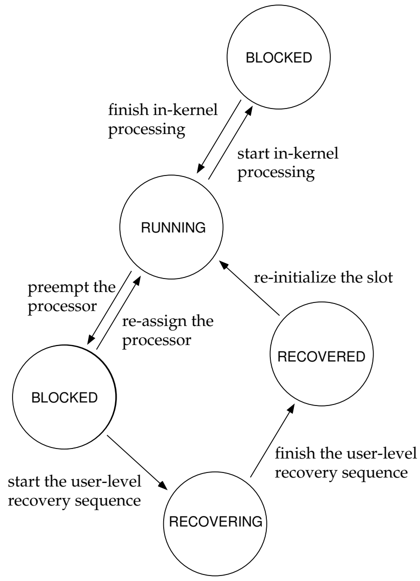


Fig. 3. Five states of k-threads.

the k-thread (either RUNNING, BLOCKED, SUSPENDED, RECOVERING, or RECOVERED). From user-level schedulers to the kernel scheduler, the c_max_parallelism field conveys the number of runnable u-threads. A single principle is employed for accessing the c-area to make the interaction between the user-level schedulers and kernel scheduler asynchronous: the sender of information writes to the c-area as soon as the information is available, while the receiver reads the c-area only when it requires the information.

Sending information from the user level to the kernel In the case of c_max_parallelism, the senders of information are user-level schedulers and the receiver is the kernel scheduler. Each user-level scheduler updates the c_max_parallelism field in its c-area whenever the number of runnable u-threads changes. When the kernel scheduler tries to redistribute the processor resources among user address spaces, it refers to the fields for all user address spaces in turn, examines the balance between the number of runnable u-threads and the number of processors assigned, and changes the assignment whenever necessary. The new assignment is chosen so that the number of processors does not exceed the number of runnable u-threads in each user address space. Although malicious user-level schedulers can fool the kernel scheduler and obtain a large amount of processor time in the short term, they will be reined in eventually through the kernel's medium-term processor scheduling.

Sending information from the kernel to the user level In the case of c_status and c_registers, the sender of information is the kernel scheduler and the receiver is a user-level scheduler. The kernel scheduler updates c_status whenever a k-thread changes scheduling states. When a k-thread is created, its corresponding c_status field is set to RUNNING. This means that the corresponding k-thread is assigned a processor and is actually running. The c_status field takes the value BLOCKED when the kernel scheduler suspends the k-thread at a point where it is not recoverable by a user-level scheduler (suspension due to page faults, I/O operations, etc.). The kernel scheduler sets the c_status field of a k-thread to SUSPENDED when the k-thread has lost its processor due to the redistribution of the processor resources. At the same time, the kernel scheduler updates the c_registers field with the latest processor context of the k-thread. In addition, k-threads in the BLOCKED and SUSPENDED states may return to the RUNNING state when they are unblocked or they regain processors.

The user-level scheduler tries to find SUSPENDED k-threads when a k-thread becomes idle within the user address space. Upon finding one, if that scheduler wants to immediately resume the execution of the u-thread that is corresponding to the k-thread, the user-level scheduler can recover the u-thread by loading the contents of the c_registers to a running k-thread. An important implementation issue for the recovery sequence is that the entire sequence must be atomic in terms of another update of c_registers by the kernel scheduler. Also neither the kernel nor the user-level schedulers should wait for the other in order to realize asynchronous interaction between the two layers. To satisfy these two requirements, the user-level scheduler uses RECOVERING and RECOVERED states. When the user-level scheduler starts the recovery sequence, it changes the value of c_status from SUSPENDED to RECOVERING, thereby requesting the kernel scheduler not to update c_registers any more[2]. After the user-level scheduler finishes copying the c_registers, it sets the c_status to RECOVERED. The kernel scheduler can reclaim elements of the c-area of this state whenever necessary. Alternatively, the kernel scheduler may wish to resume the execution of k-thread when its state is RECOVERING. In this case, the k-thread should not continue execution from the point of suspension because the u-thread executed by the k-thread has already been recovering at the user level. Instead, the k-thread scheduler re-initializes the k-thread and starts execution as if it were newly created. Note that the kernel scheduler does not wait for user-level schedulers, and vice versa.

2.3 Notifiers

Although the above-mentioned components are sufficient for executing most applications, some applications require notification of the occurrence of specific kernel-level events for their correct and/or efficient execution. For example, applications whose u-threads must be scheduled preemptively require the notification of "timer expired" events. Real-time applications require information on kernel-level processing such as "I/O done" events and "page fault occurred" events. A notifier, used to convey such information to the user level, is essentially a k-thread that is created when an application-specified event happens. The difference between a notifier and a k-thread is that several processor registers of a notifier carry information on an event. After the notifier enters the user level and performs user-defined processing to handle the event, the user-level scheduler can use the notifier to schedule u-threads in the same way as with ordinary k-threads.

3 Minimizing Thread-Switching Overhead

In this section, we show that our thread mechanism minimizes the overhead of vertical switchings and horizontal switchings in both kernel and user address spaces. Let us start the discussion with the ideal

[2] The update of c_status should itself be an atomic compare-and-set.

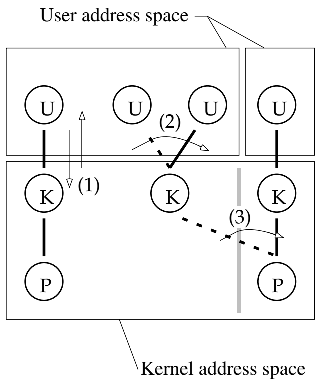


Fig. 4. Horizontal and vertical switchings in an ideal situation.

situation. Three patterns of switching are indispensable even in the ideal situation. First, a horizontal switching occurs within the kernel scheduler when the kernel redistributes the processor resources among user address spaces. Since multiple applications share the processor resources in a multiprogrammed multiprocessor system, redistribution of the processor resources is unavoidable. Another pattern is a pair of vertical switchings that occur when a system call is invoked (one for entering the kernel and the other for exiting). The third one is a horizontal switching within a user-level scheduler that occurs when a rendezvous of two u-threads takes place. Fig. 4 illustrates these three switching patterns. Existing thread mechanisms cannot realize this ideal situation because they involve extra horizontal switchings in the kernel scheduler, extra vertical switchings, and/or extra horizontal switchings in the user-level schedulers. In the following three subsections, we compare these extra switchings in three thread management schemes: a conventional two-level thread management scheme (user-level schedulers are built on top of the kernel-level scheduler without specialized interfaces for cooperation), scheduler activations scheme (user-level schedulers are built on top of the kernel-level scheduler with synchronous interfaces for cooperation), and the proposed scheme (user-level schedulers are built on top of the kernel-level scheduler with asynchronous interfaces for cooperation).

3.1 Extra Horizontal Switchings in the Kernel

In the conventional two-level scheme, extra horizontal switchings in the kernel scheduler occur because the number of runnable k-threads and the number of processors do not match (see Fig. 5). In this situation, a horizontal switching at the user level sometimes involves an extra horizontal switching of k-threads as well as a pair of extra vertical switchings. As presented in two examples in the introduction, both the kernel scheduler and the user-level schedulers initiate the extra switchings of this pattern; they occur when the kernel schedules k-threads and when u-threads are synchronized with each other.

In scheduler activations and in the proposed scheme, extra horizontal switchings in the kernel never occur because these schemes control k-threads so that the number of runnable k-threads equals the number of processors in each user address space.

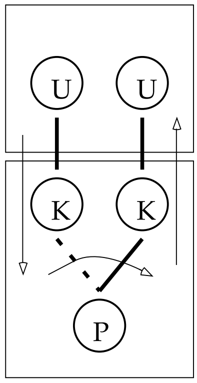


Fig. 5. Extra horizontal switchings in the conventional two-level scheme.

3.2 Extra Vertical Switchings

In addition to the extra horizontal switchings just described, the conventional two-level scheme causes extra vertical switchings. Suppose two u-threads A and B are about to make a barrier synchronization. When A reaches the synchronization point sooner than B, A can take one of two actions, depending on the state of B: A should sleep if B has already been assigned a processor, and A should switch its processor to B if B is not assigned a processor. In the conventional two-level scheme, A should issue a system call (a pair of vertical switchings) to monitor the scheduling state of B. In our scheme, on the other hand, the vertical switchings are not necessary because the c-area supplies the necessary information.

In scheduler activations, every change in scheduling status of k-threads is reported to the user level synchronously through an upcall [15], that is, a vertical switching of a k-thread from the kernel scheduler to a user-level scheduler. In our scheme, since all interactions between the kernel scheduler and user-level schedulers are asynchronous and on-demand, no vertical switching takes place in reporting the changes of scheduling status. Vertical switchings in our scheme occur only when applications issue system calls and when external interrupts arrive.

3.3 Extra Horizontal Switchings in the User Level

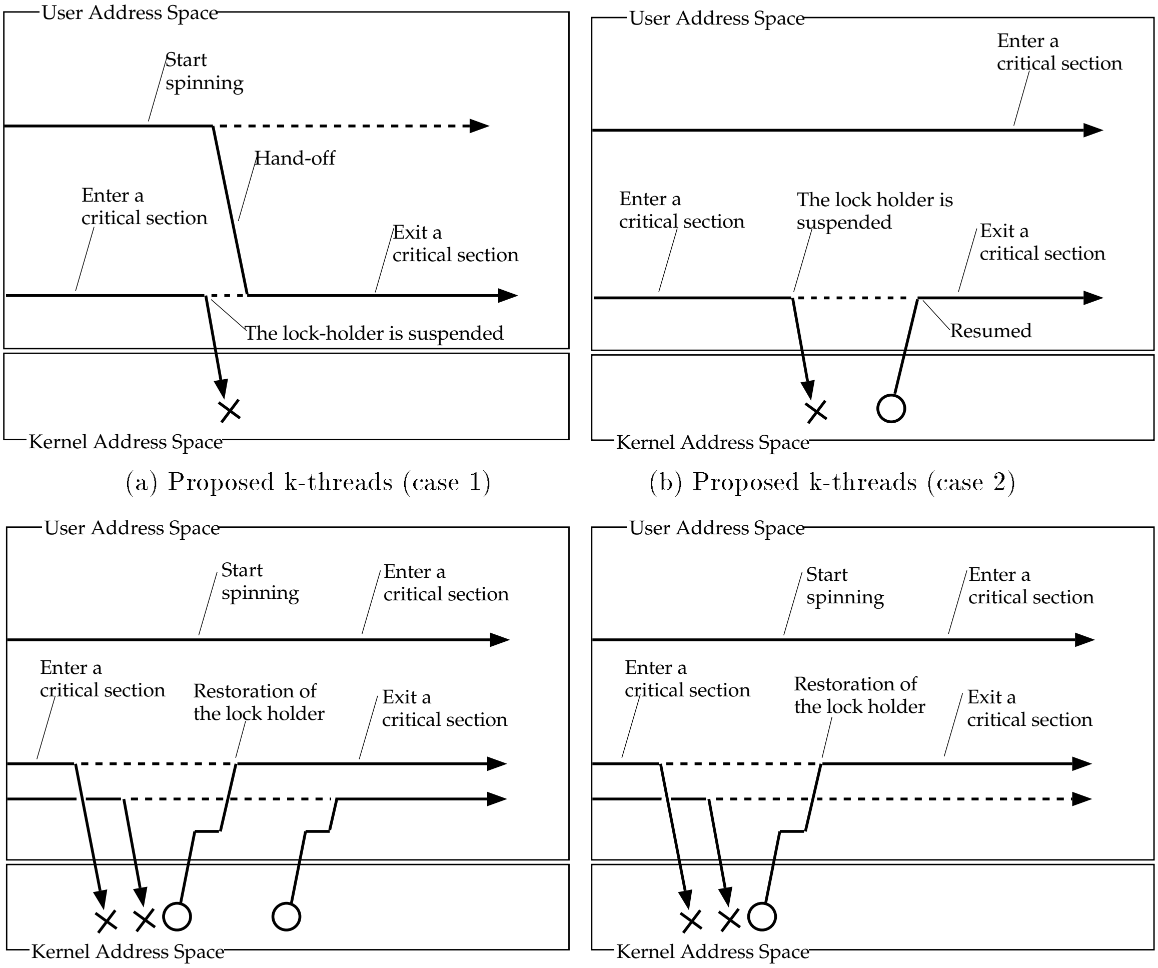
It is well-known that in the conventional two-level scheme, spin locks sometimes waste large amounts of the processor resources; if a u-thread in a critical section is preempted, other u-threads that want to enter the critical section are forced to spin for quite a long time. Synchronization among u-threads is hence achieved with sleeping locks, which cause horizontal switchings that spin locks do not require.

The scheduler activations scheme allows the use of spin locks by having the kernel scheduler notify user-level schedulers every change in the scheduling status of k-threads. When notified of a preemption of a u-thread that is holding a lock, the user-level scheduler restores the execution of the lock-holding u-thread. Although this mechanism achieves an efficient spin locks in a multiprogrammed environment, an extra horizontal switching occurs at the user level during this restoration sequence as well as the above-mentioned extra vertical switchings. This is because scheduler activations scheme has no interface for continuing a suspended k-thread, as a consequence of its design that user-level schedulers in scheduler activations are expected to react to upcalls from the kernel scheduler immediately.

In our scheme, user-level schedulers are not expected to react to updates of the c-area immediately, hence the kernel scheduler is allowed to restore the execution of k-threads that are once suspended but are not recovered by user-level schedulers. Taking this into account, spin locks are implemented efficiently by recovering u-threads on an on-demand basis. When a u-thread is spinning to acquire a lock for a critical region, it periodically examines the c_status of the k-thread that holds the lock. The u-thread can continue spinning without wasting too much processor time as long as the lock-holder is running. If, on the other hand, the lock-holder is suspended, the waiting u-thread hands off its k-thread to the lock-holder, thereby letting the lock-holder exit from the current critical region as soon as possible. Note that horizontal switching related to spin locks occur only when a u-thread is suspended within a critical section *and* another u-thread starts spinning to enter the same critical section. Please refer to the appendix for a pseudo-code the description of our lock/unlock implementation.

Fig. 6 illustrates the difference between our scheme [(a) and (b)] and scheduler activations [(c) and (d)] when preemption occurs during a critical section. In case (a), a u-thread was suspended inside a critical section, but was resumed by the kernel scheduler before other u-threads become aware of the suspension. This sequence involves two vertical switchings and no horizontal switchings. In case (b), during the suspension of a u-thread inside a critical section, another u-thread tried to enter the same critical section. The waiting u-thread, being aware of the suspension, performs a hand-off scheduling. This sequence involves one vertical switching and a horizontal switching. For case (c) with the scheduler activations scheme, notice of the suspension of a k-thread is immediately sent to the user address space using another k-thread in the same address space. This sequence involves four vertical switchings and two horizontal switchings. Since case (c) corresponds to case (a) in our scheme, the scheduler activations scheme involves two extra vertical switchings and two extra horizontal switchings. Case (d) involves three vertical switchings and a horizontal switching, that is, two extra vertical switchings are included.

In summary, due to the proposed kernel interface, all switchings in our thread mechanism follow the three patterns in Fig. 4, therefore the number of switchings is minimized. In addition, both vertical and horizontal switchings are implemented with the same cost as those of the conventional two-level scheme, even though in our scheme, the c-area must be updated on each vertical switching. This is achieved by implementing the c-area as follows: we divide the contents of the c-area into a kernel-writable part (c_registers) and user-writable part (c_status and c_max_parallelism), and store them on different virtual memory pages. By mapping the kernel-writable part as read-only to user address spaces, we can use the c-area as the repository of the k-thread contexts, in place of ordinary thread control blocks (TCB) in the kernel address space. Since updates of c-area simply replace updates of TCB, there is no additional cost for manipulating c-area. Our thread mechanism therefore achieves the minimum overhead in thread management.



(a) Proposed k-threads (case 1)

(b) Proposed k-threads (case 2)

(c) Scheduler activations (case 1)

(d) Scheduler activations (case 2)

Fig. 6. Example: Spin lock with preemption.

Table 1. Cost of basic operations (inst./operation).

Operation	U-thread	K-thread
Create	92	560
Suspend	13	60
Resume	46	82
Switch	40	260
Address Space Switch	—	300
Terminate	46	513

4 Performance

To ensure the effectiveness of our proposal, we investigated the performance of our thread mechanism. The thread mechanism has been implemented as a part of our research operating system XERO [16, 17, 18]. Table 1 presents the number of instructions per operation for all thread-related operations in XERO on Motorola 68030-based machines, obtained through the instruction-level tracing of a processor.

Assuming these figures for fundamental thread management costs, we built an emulator to investigate the performance characteristics of our thread mechanism under various multiprogramming situations. Inside this emulator, processors, k-threads, and u-threads work concurrently in accordance with the virtual time. The system has eight processors that share the whole primary memory. The effects of caches are not considered.

4.1 Compared Thread Management Schemes

For comparison, we measured thread switching costs of the following four different thread management schemes that we have discussed throughout the paper. The *k-only* scheme implements threads solely in the kernel. That is, a k-thread is created each time applications create a u-thread. The k-threads are scheduled by the kernel. Each operation on u-threads (creation, suspension, resumption, and termination) thus involves at least one system call. The *conventional two-level* scheme involves user-level schedulers on top of the kernel-level threads without special kernel support (the "u/k two level" scheme in the figures below). Each application creates k-threads, but less than or equal to the number of processors in the system. *Scheduler activations* maintains the number of runnable k-threads in the system through synchronous interaction between the kernel scheduler and user-level schedulers. The *proposed scheme* maintains the number of runnable k-threads in the system through an asynchronous interface using c-area.

4.2 Applications

Three parallel applications are used for generating multiprogrammed workloads. "MULT" multiplies two 256x256 arrays. By partitioning the arrays into blocks of 64x64, this application invokes sixteen u-threads almost from the beginning to the end. In the final part of the program, all u-threads make a barrier synchronization. "FORCE" implements an algorithm for N-body force calculation [19]. This divide-and-conquer algorithm has two phases, before each of which all u-threads should synchronize.

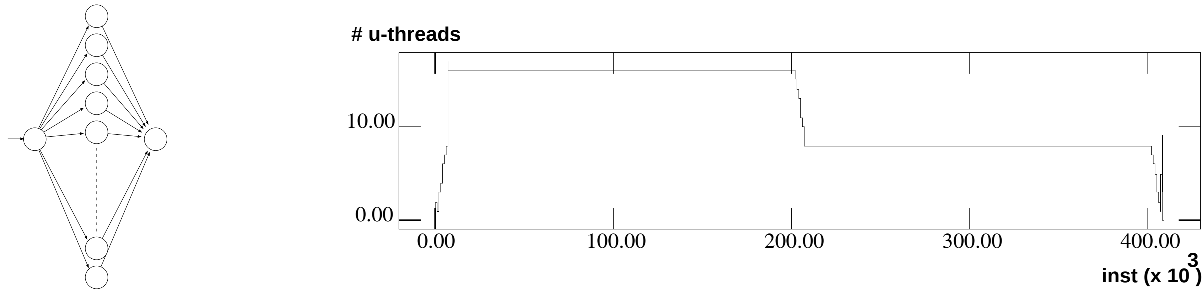
In this implementation, sixteen u-threads are used for each of the two phases. The third application program, “WAVE,” is a wave-front computation. Each u-thread connected in a 10x10 mesh topology obtains data from one or two neighbors and passes it to other neighbors after local processing.

We examined the primary characteristics of the three applications; Fig. 7 shows the synchronization schemes and the change in logical parallelism (the number of runnable u-threads) during an execution for each. Here, each application was executed using the conventional two-level scheme in a uniprogrammed environment. In MULT, synchronization among threads occurs only at the beginning and the end of the program after about 200,000 instructions are executed. The execution trace in Fig. 7(a) shows this. In the first half of the program execution, all u-threads are active. Since the system has only eight processors, only eight u-threads finish their execution after 200,000 instructions of execution time; the remaining eight, in the second 200,000 instructions. In FORCE, sixteen threads all synchronize once every 10,000 instructions. Since a barrier synchronization is performed at every 10,000 instructions with sixteen u-threads, the effective interval between two barrier synchronizations on eight processors is about 20,000 instructions [Fig. 7(b)]. In WAVE, each thread synchronizes with its neighbors every 10,000 instructions, changing the overall parallelism between one and ten. Synchronization of a small number of u-threads makes a number of small pulses. The average consumption of processors for MULT, FORCE, and WAVE are 7.87, 7.94, and 4.99, respectively.

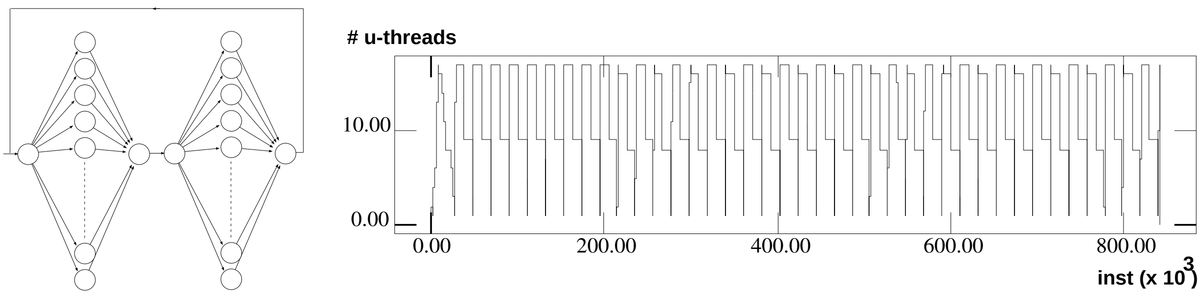
4.3 Results

First, fundamental properties of the four thread management schemes are examined by taking a close look at a workload. Fig. 8 and Fig. 9 depict the change in the number of runnable u-threads and k-threads when two processes running FORCE are executed in parallel under the four thread management schemes. We can identify from Fig. 8 that the pulses (the period between two barrier synchronizations) are larger in the k-only scheme and the conventional two-level scheme than in scheduler activations and the proposed scheme. This indicates that scheduler activations and our scheme achieve faster thread operations. Fig. 9 reveals that the number of runnable k-threads and the change in their number is completely different in the four schemes. In the k-only scheme, thirty-two k-threads are used because each FORCE process uses sixteen k-threads. In the conventional two-level scheme, sixteen k-threads are used because each FORCE allocates a maximum of eight k-threads, that is, the number equal to that of processors in the system. In scheduler activations and the proposed scheme, the number of runnable k-threads is kept to less than or equal to eight, and the number of suspensions and resumptions of k-threads is quite small.

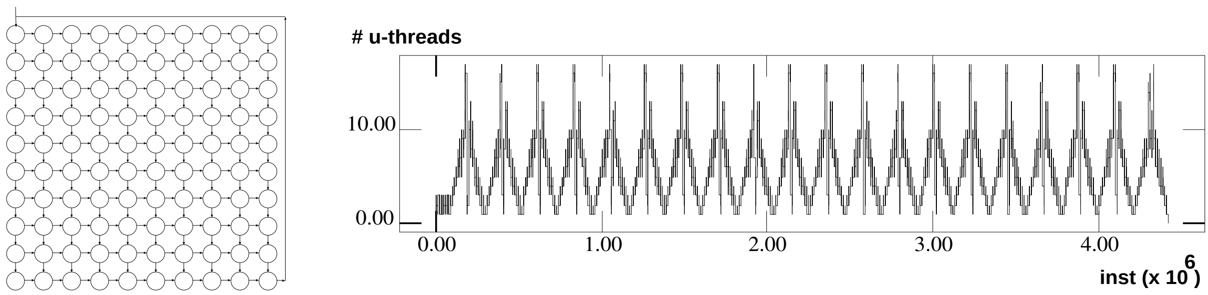
The next set of experiments for comparing the four schemes changes the overall load of the system (that is, the number of processes in the system). We used seven different workloads, mixing MULTs, FORCEs, and WAVEs. Fig. 10 presents the cases when multiple processes of a single application are executed in parallel, while Fig. 11 shows cases when two or three applications are executed in parallel. As expected from the discussion in Section 3, our scheme always performs the best of the four. The k-only scheme and conventional two-level scheme incur high overhead for high system loads. The primary source of additional overhead over our scheme is extra vertical switchings, which are initiated by the



(a) MULT: array multiplication.

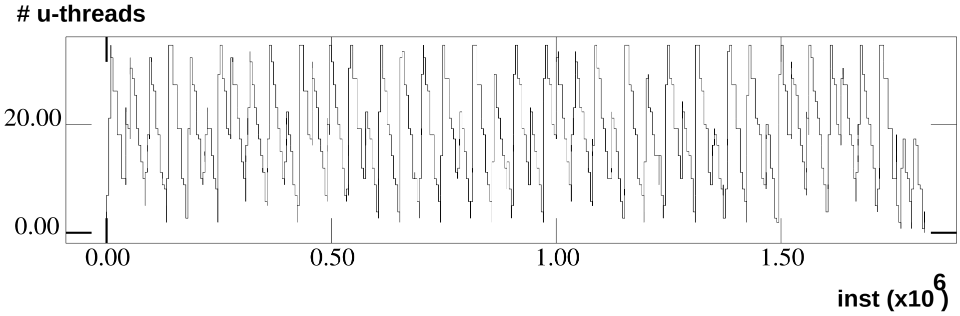


(b) FORCE: force calculation.

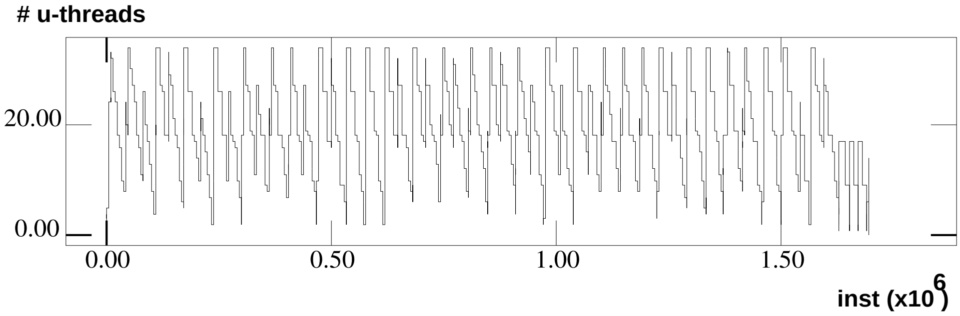


(c) WAVE: wave-front algorithm.

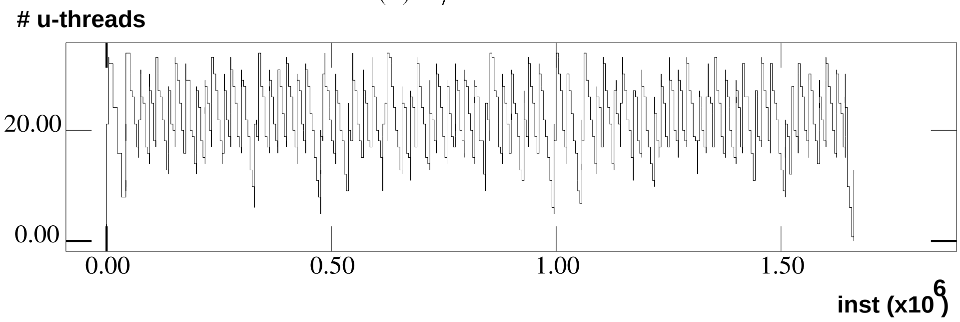
Fig. 7. Characteristics of MULT, FORCE, and WAVE.



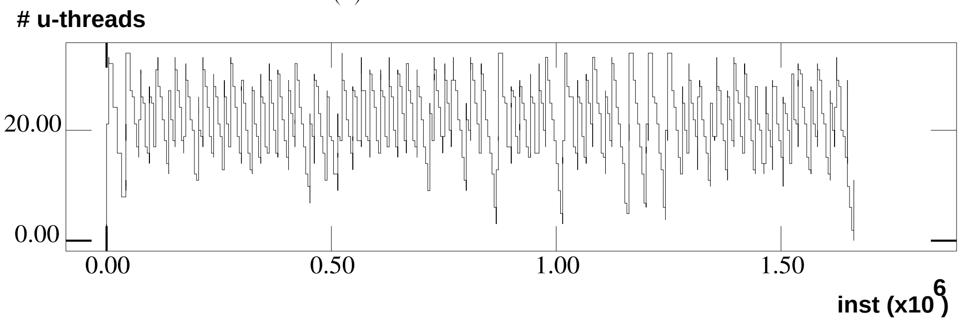
(a) K-only.



(b) U/k two-level.

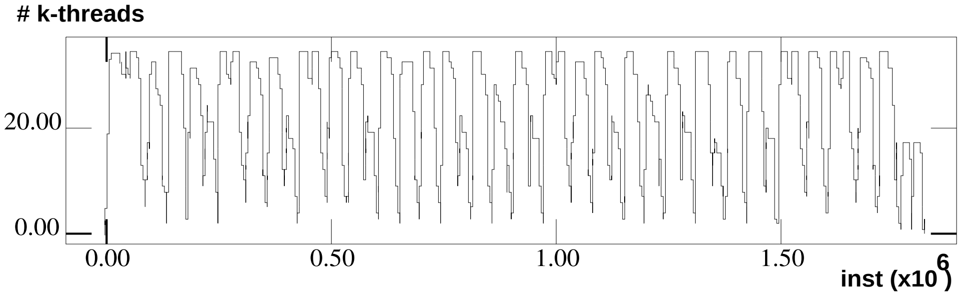


(c) Scheduler Activations.

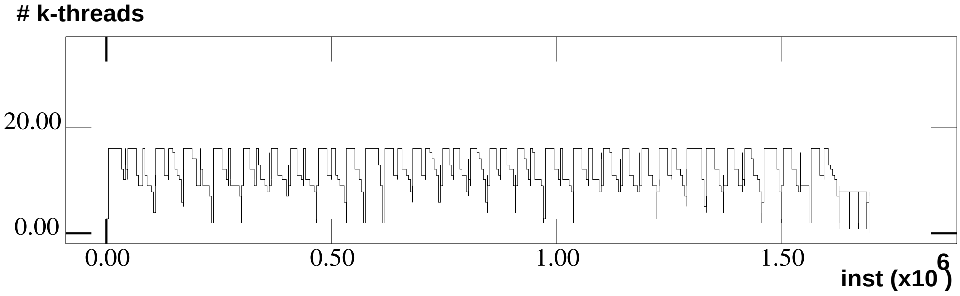


(d) Proposed scheme.

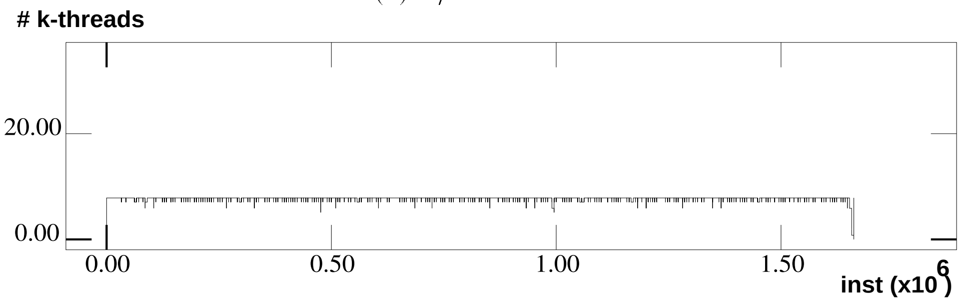
Fig. 8. The number of u-threads during the execution of two FORCE.



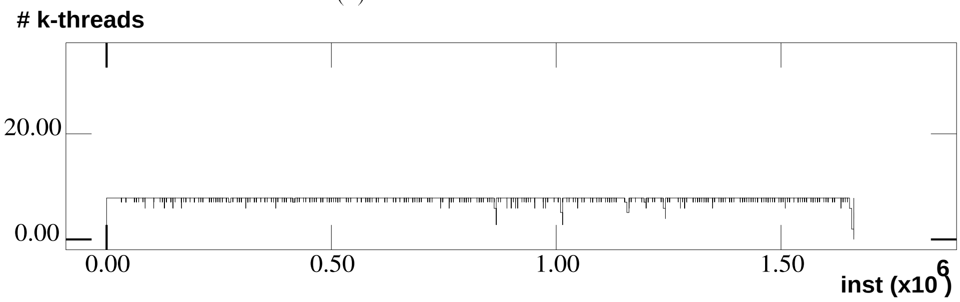
(a) K-only.



(b) U/K two-level.



(c) Scheduler activations.



(d) Proposed scheme.

Fig. 9. The number of k-threads during the execution of two FORCE.

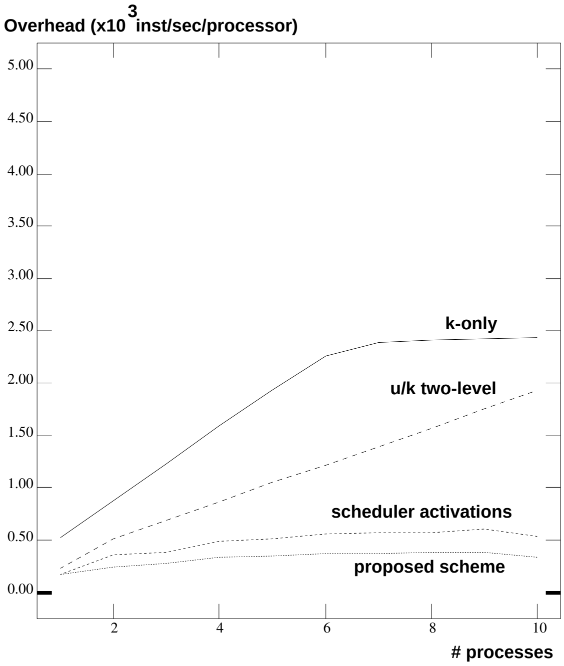
kernel scheduler for scheduling more than one k-thread with one processor. The scheduler activations scheme and our scheme both control the number of runnable k-threads automatically, thus the overhead of thread management is little affected by the system load. The difference in overhead between our scheme and the scheduler activation is due to the difference in style of interaction between the kernel and user-level schedulers; the synchronous interaction in scheduler activations causes more overhead than the asynchronous interaction in our scheme. The thread management overhead of our scheme is, on average, less than 27.5% overhead of the k-only scheme, less than 40.5% overhead of the conventional two-level scheme, and less than 81.0% overhead of scheduler activations.

In order to make sure that the above results are not greatly affected by the characteristics of different applications, we conducted another set of experiments. We clarified the effects of communication frequency of applications on the overheads of the four thread management schemes. We changed the computation-communication ratio of the applications in the range of 1/16 to 16 times their original ratios. The results for cases of two MULTs, two FORCEs, and two WAVEs are shown in Fig. 12 (a), (b), and (c), respectively. These results confirm the qualitative performance advantage of our proposed scheme over existing thread management schemes, even when applications with fine-grained threads (less than one thousand instructions per synchronization) are executed. The results agree reasonably with the theoretical result presented in Section 3; the proposed thread mechanism minimizes the overhead of thread switching.

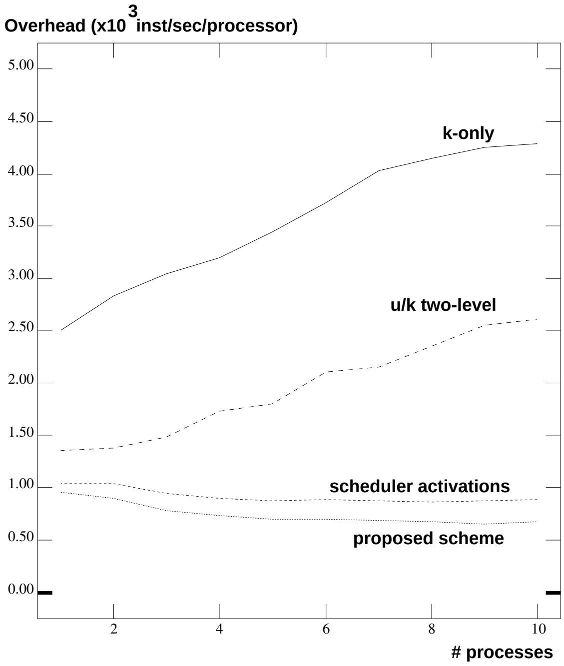
5 Conclusions

This work enhanced our previous work [16, 18] so that we can minimize the thread-switching overhead as a whole, that is, minimize thread switching both in user address spaces and in the kernel. The most important contribution of this work is thus not that we achieved better performance than existing thread management systems, but that we finally know how the thread management system can be constructed for *best* performance after years of reconstructing thread management system architecture. The proposed scheme is a software architecture for thread management whose performance we cannot further improve except through tune-up of individual thread operations.

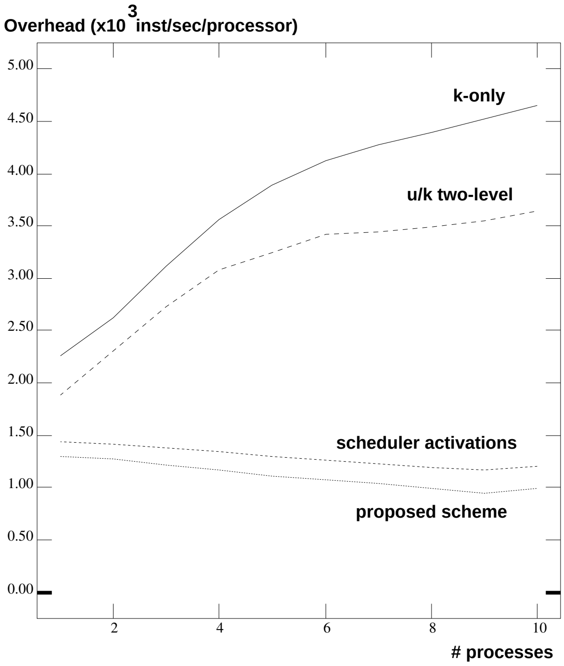
There remain several open questions. First, we would like to know which processor allocation policy best complements the proposed thread management scheme. C. McCann et al. [20] thoroughly investigated the effects of processor allocation policies on the performance of multiprogrammed systems. Since they use scheduler activations for their dynamic thread management schemes, they assume that the kernel-level schedulers and user-level schedulers interact synchronously. How asynchronous interaction affects the choice of processor allocation policy must be investigated. Another attractive future work is to extend the proposed scheme for distributed shared-memory multiprocessor systems like Dash [21]. In distributed shared-memory machines, we must consider the difference in costs for local and remote memory references, or cache affinity [22], for processor redistribution.



(a) MULT.



(b) FORCE.



(c) WAVE.

Fig. 10. Effect of system load on thread overhead (single application).

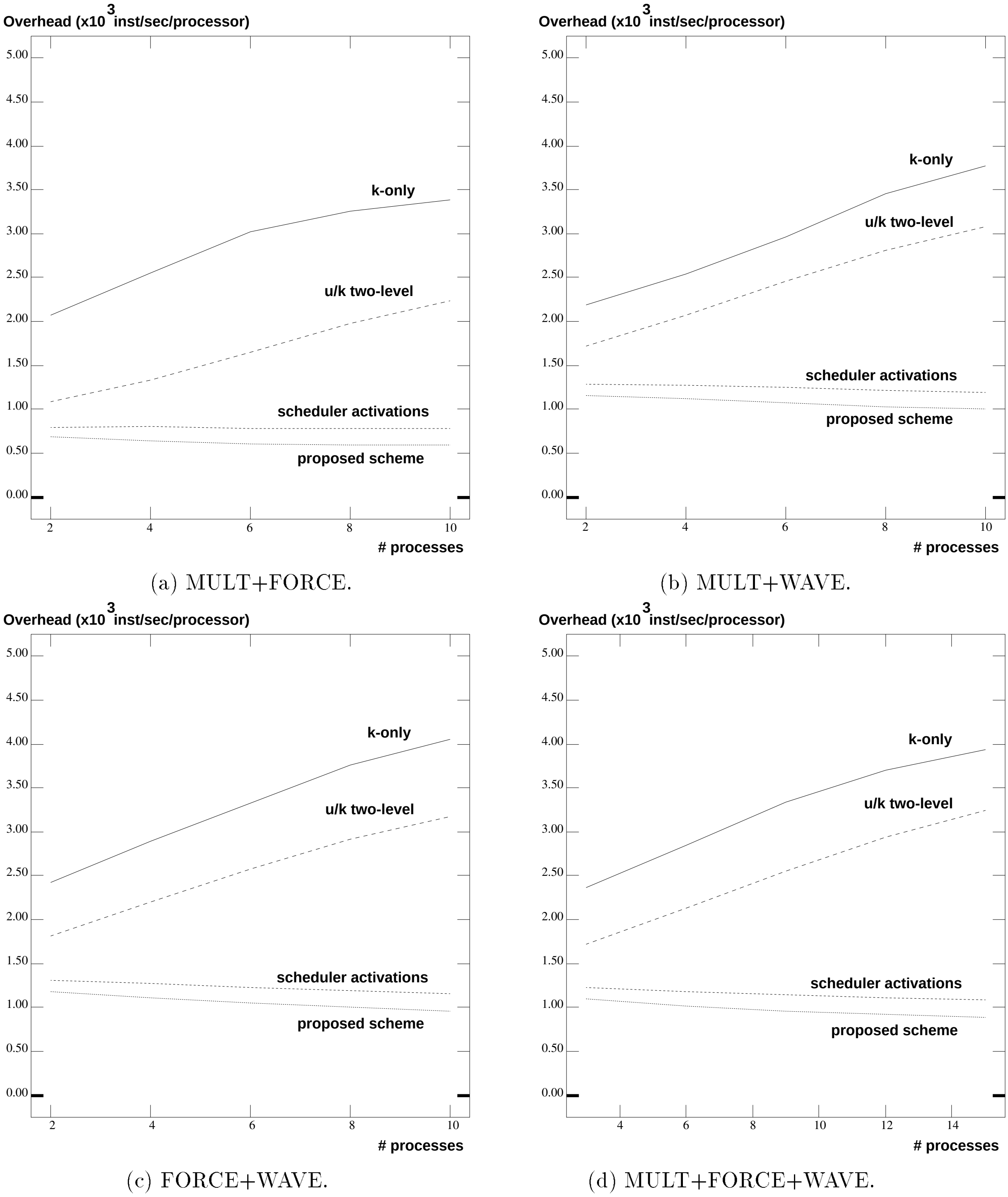
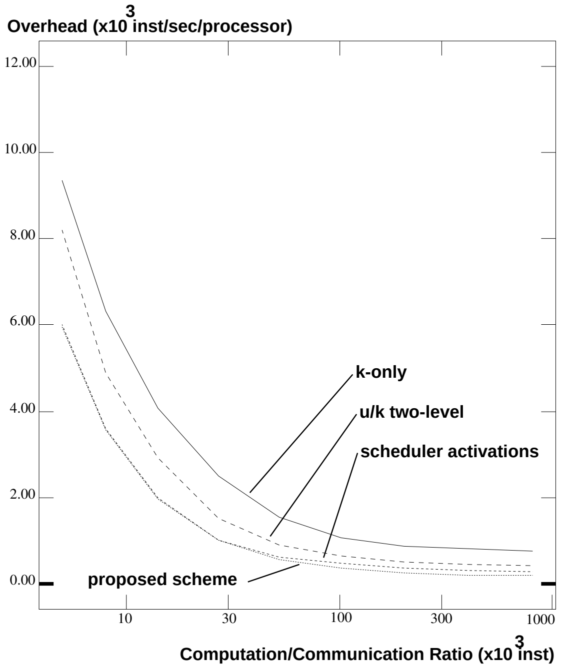
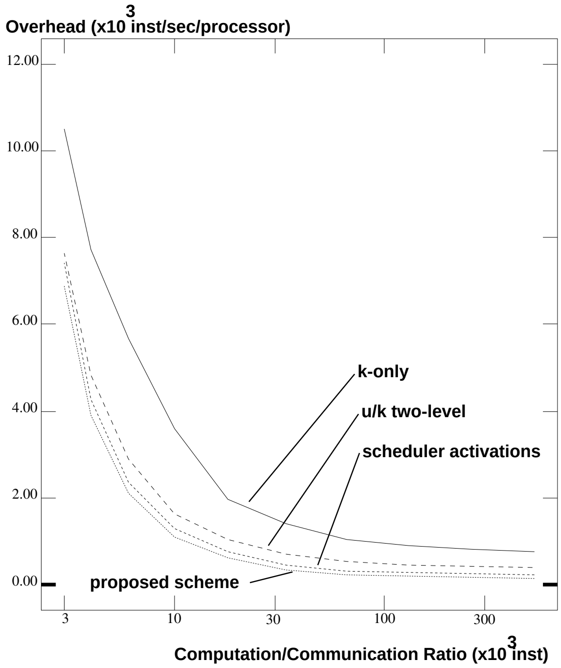


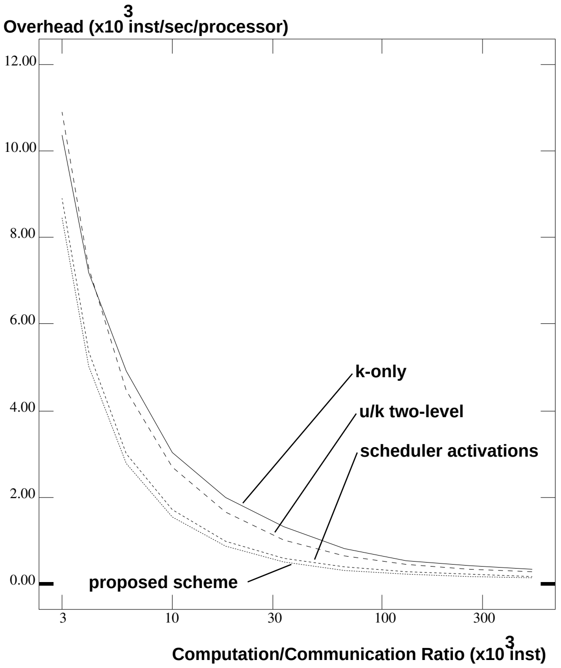
Fig. 11. Effect of system load on thread overhead (mixed applications).



(a) MULT.



(b) FORCE.



(c) WAVE.

Fig. 12. Effect of communication frequency on thread overhead.

```
uthread_schedule()
{
 loop:
        if (ut = uthread_remove_readyq()){
                context_restore(ut);
                /*NOTREACHED*/
        } else if (kt = search_for_suspended_kthread()) {
                ut =  the u-thread that has been executed on kt;
                context_copy_from_c_area(kt, ut);
                uthread_set_readyq(ut);
                goto loop;
        } else {
                KThread_Terminate();
                /*NOTREACHED*/
        }
}
```

Fig. 13. Entry point for new k-threads.

Appendix User-level Scheduling and Synchronization Code

A user-level scheduler can implement non-preemptive scheduling of u-threads only with our k-threads and the c-area. The core of the implementation is a procedure to conceal the unstable nature of k-threads. Fig. 13 presents the procedure uthread_schedule for this purpose. This procedure acts as the entry point for new k-threads. The uthread_schedule procedure first tries to execute a runnable u-thread in the ready queue with uthread_remove_readyq and context_restore. If this attempt fails, uthread_schedule calls search_for_suspended_kthread to search for a suspended k-thread by examining c_status. The context_copy_from_c_area procedure then copies the contents of c_registers to the data structure that represents the suspended u-thread, which is returned to the ready queue with uthread_set_readyq. Finally, if there exists no u-thread to execute, the current k-thread is terminated with the kthread_terminate system call.

In addition to uthread_schedule, code for synchronizing u-threads should also be implemented carefully. Due to the unstable nature of k-threads, a u-thread may wait infinitely for another u-thread whose k-thread has been suspended by the kernel. The lock and unlock procedures shown in Fig. 14 that avoid the problem. Instead of a simple while loop that tries a test-and-set instruction, this lock procedure watches for the scheduling status of the k-thread associated with the u-thread that holds the lock. The c_status field in the c-area makes it possible to do this at quite low cost. If the k-thread is suspended or blocked, u-threads waiting for the lock block themselves.

When a u-thread tries to enter a critical section by executing lock, it registers itself in the FIFO queue, in which the first element represents the lock-holder thread, and the rest wait for the lock. Within the while loop, the waiting u-threads check the c-area, and if the lock-holder is blocked or suspended, they yield their processors. This is achieved through four procedures: uthread_sleep, uthread_wait, uthread_unsleep, and uthread_wakeup. The uthread_sleep procedure declares which event to wait for, and uthread_wait actually blocks the current u-thread. The uthread_wakeup procedure unblocks a blocked u-thread, and uthread_unsleep clears the effect of uthread_sleep. When the lock-holder thread exits the

```
lock(Lock) {
        uthread_sleep(Lock);
        FIFO_set(Lock.queue, self_ut);
        while (FIFO_head(Lock.queue) != self_ut) {
                /* check the c-area */
                switch (c.c_kthread[FIFO_head(Lock.queue)->id].c_status) {
                  case RUNNING:
                  case RECOVERED:
                        continue;           /* spin again */
                  case BLOCKED:
                  case SUSPENDED:
                  case RECOVERING:
                        uthread_wait();
                        break;
                 }
        }
        uthread_unsleep();
}

unlock(Lock)
{
        FIFO_remove(Lock.queue);
        if (next_holder = FIFO_head(Lock.queue) != NULL)
                uthread_wakeup(&Lock, next_holder);
}
```

Fig. 14. Mutual exclusion.

critical section, it wakes up a waiting thread. In this way, u-threads avoid wasting processor resources.

References

[1] A. Tevanian Jr., R. Rashid, D. B. Golub, D. L. Black, E. Cooper, and M. W. Young, "Mach threads and the Unix kernel: The battle for control," in *Proceedings of the USENIX 1987 Summer Conference*, pp. 185–197, Summer 1987.

[2] D. R. Cheriton, "The V distributed system," *Communications of the ACM*, vol. 31, pp. 314–333, March 1988.

[3] C. P. Thacker and L. C. Stewart, "Firefly: A multiprocessor workstation," *IEEE Transactions on Computers*, vol. 37, pp. 909–920, August 1988.

[4] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse, and H. van Staveren, "Amoeba: A distributed operating system for the 1990s," *IEEE Computer*, vol. 23, pp. 44–53, May 1990.

[5] J. Edler, J. Lipkis, and E. Schonberg, "Process management for highly parallel UNIX systems," in *Proceedings of the USENIX Workshop on UNIX and Supercomputers*, pp. 1–17, September 1988.

[6] T. E. Anderson, E. D. Lazowska, and H. M. Levy, "The performance implications of thread management alternatives for shared-memory multiprocessors," *IEEE Transactions on Computers*, vol. 38, pp. 1631–1643, December 1989.

[7] M. L. Scott, T. J. LeBlanc, and B. D. Marsh, "Multi-model parallel programming in Psyche," in *Proceedings of the second ACM Symposium on Principles and Practices of Parallel Programming*, pp. 70–78, March 1990.

[8] J. E. Faust and H. M. Levy, "The performance of an object-oriented threads package," in *Proceedings of Conference on Object-Oriented Programming: Systems, Languages, and Applications and European Conference on Object-Oriented Programming*, pp. 278–288, October 1990.

[9] T. E. Anderson, "The performance of spin lock alternatives for shared-memory multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, pp. 6–16, January 1990.

[10] D. Golub, R. Dean, A. Forin, and R. Rashid, "UNIX as an application program," in *Proceedings of the USENIX 1990 Summer Conference*, pp. 87–95, June 1990.

[11] B. D. Marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos, "First-class user-level threads," in *Proceedings of the thirteenth ACM Symposium on Operating Systems Principles*, pp. 110–121, 1991.

[12] R. Govindan and D. P. Anderson, "Scheduling and IPC mechanisms for continuous media," in *Proceedings of the thirteenth ACM Symposium on Operating Systems Principles*, pp. 68–80, October 1991.

[13] M. L. Powell, S. R. Klieman, S. Barton, D. Shar, D. Stein, and M. Weeks, "SunOS multi-thread architecture," in *Proceedings of the USENIX 1991 Winter Conference*, pp. 65–79, Winter 1991.

[14] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy, "Scheduler activations: Effective kernel support for the user-level management of parallelism," *ACM Transactions on Computer Systems*, vol. 10, pp. 53–79, February 1992.

[15] D. Clark, "The structure of systems using upcalls," in *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pp. 128–140, March 1988.

[16] S. Inohara, K. Kato, A. Narita, and T. Masuda, "Thread facility based on user/kernel cooperation in the XERO operating system," in *Proceedings of the fifteenth IEEE International Computer Systems and Applications Conference*, pp. 398–405, IEEE Computer Society, September 1991.

[17] K. Kato, S. Inohara, A. Narita, S. Chiba, and T. Masuda, "Design of the XERO open distributed operating system," *JIP – Information Processing Society of Japan*, vol. 14, pp. 384–397, April 1992.

[18] S. Inohara, K. Kato, and T. Masuda, "'Unstable Threads' kernel interface for minimizing the overhead of thread switching," in *Proceedings of the 7th IEEE International Parallel Processing Symposium*, pp. 149–155, IEEE Computer Society, April 1993.

[19] J. Barnes and P. Hut, "A hierarchical o($n \log n$) force-calculation algorithm," *Nature*, vol. 324, pp. 446–449, 1986.

[20] C. McCann, R. Vaswani, and J. Zahorjan, "A dynamic processor allocation policy for multi-programmed shared-memory multiprocessors," *ACM Transactions on Computer Systems*, vol. 11, pp. 146–178, May 1993.

[21] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, "The directory-based cache coherence protocol for the DASH multiprocessor," in *Proceedings of the 17th Annual Symposium on Computer Architecture*, (New York), pp. 148–160, IEEE, May 1990.

[22] R. Vaswani and J. Zahorjan, "The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors," in *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pp. 26–40, Association for Computing Machinery SIGOPS, October 1991.