ABSTRACT: Parallel applications on multiprocessors achieve better performance when they run on simpler microkernel scheduling mechanisms with appropriated user level scheduling policies. Our purpose is to offer to application programmers a set of new and simple primitives to get more control over the user-level thread scheduling. This paper presents a new library scheduling approach, based in the CThreads package, running on top of the Mach 3.0 microkernel. To measure and evaluate the proposed primitives, we present experimental results of a multi-threaded producer-consumer benchmark.

The proposed scheduling primitives reduce the application execution time from 10 to 20% depending on the number of kernel threads and physical processors.

Keywords: thread packages, user-level scheduling, microkernels, shared memory multiprocessors, parallel applications.

1. Introduction

We present an experimental proposal of a thread package enhancement to support parallel applications running on shared memory multiprocessors. Our main goal is to allow scheduling decisions to be taken at the right place, both at kernel and user level. Applications know how many execution flows may run, which synchronization mechanisms are more suitable and which user-level threads have to run at every moment on the virtual processors given by the kernel. Instead, the operating system manages efficiently physical processors and memory.

The experience has been done on a multiprocessor running the Mach microkernel. This system design model involves new layers in scheduling decisions. The applications have to deal with many entities (kernel, subsystems and libraries). Up to now, the kernel was responsible for thread scheduling. Nowadays there are also libraries that help the user level scheduling. This control, when done independently in both layers, may end in a global low performance. We have exploited and extended the user-level context switch of the CThreads package.

Beside this work, we have also ported and adapted a new event driven scheduling policy of kernel threads based on ESCHED proposals [1][2].

2. User-level parallelism

CThreads is a library provided with Mach which includes primitives to manipulate user-level threads of control and to ease multithreaded programming [3]: primitives to fork and join threads, and primitives for handling mutual exclusion and synchronization based on mutex variables, spin locks and condition variables. All global and static variables are shared among all threads.

The total amount of threads needed by an application is potentially much larger than the number of kernel threads that can be reasonable dedicated to it. The CThreads package allows a pool of threads to run on a specified amount of kernel threads. Some of the services offered are:
• Creation of new cthreads: cthread_fork (function, argument).
• Termination of cthreads: cthread_exit (status).
• Joining the execution of two cthreads: cthread_join (cthread).
• Detaching cthreads to avoid the requirement of joining: cthread_detach (cthread).

This work has been supported by the Ministry of Education of Spain (CICYT) under contract TIC 94-439
• To give the cthread virtual processor to another cthread. If no other application cthread can run, do a context switch at kernel level (possibly to another application): `cthread_yield()`.
• To access to critical regions through the spin locks and mutex locks: `spin_lock(slock)`, `spin_unlock(slock)`, `mutex_lock(mlock)` and `mutex_unlock(mlock)`.
• To synchronize on condition variables: `condition_wait(cond_var, lock)` and `condition_signal(cond_var)`.
• To set the limit of virtual processors (kernel threads) that the application can use. By default, there is no limit, so the kernel maintains a kernel thread for each cthread: `cthread_set_kernel_limit(N)`.

3. Scheduling proposals to the thread package

Kernel threads have been proved to be too expensive to support relatively fine-grain parallelism. User-level thread packages uses procedure calls to provide thread management operations, avoiding the overhead of kernel traps.

In the microkernels arena, the kernel usually relies on message passing mechanisms to block and unblock kernel threads. Some proposals suggest the replacement of this mechanism with a kernel call with less overhead [4].

A handicap for the application execution is the poor information that the current CThreads library returns: there is no knowledge whether a call has succeeded or not and the reason why. And the library often takes flow control decisions transparently to the user. We believe the thread package calls may return more information of what has really been done.

Our purpose is to offer to application programmers a set of new and simple primitives to get more control over the user-level thread scheduling. These new primitives have to work only at user level. The design of our new execution environment has taken care of keeping the semantics the user expects to find at the new tools we have provided. If she/he wants a context switch at user level, and it is not possible, the control is returned to the calling thread to let her/him decide where to continue. We show that this style of solution improves the application execution time.

We have made some modifications to allow CThreads to do user level context switches whenever possible, and consequently to reduce the number of Mach kernel calls. The old implementation relayed on forcing the kernel scheduler to choose between all runnable threads on the target processor. The application is now responsible for choosing the more convenient thread to continue on the same kernel thread. Therefore, the multiplexation of cthreads on top a controlled amount of kernel threads also reduces the large resource allocation needed in the kernel space to manage the task concurrence (by default a kernel thread is instantiated for each user thread).

Although some user-level thread packages ([5], [6]) incorporate scheduling policies, they are mainly designed to work on uniprocessors, thus assuming the presence of only one physical processor [7], or they have other types of restrictions, that reduce their performance, compared with the kernel services.

Moreover, within an application, to limit the number of context switches to the times a thread has to block is the best way to save library overhead time. The application will finish when all its threads terminate, so it seems not to be necessary to add to the user level all the semantics that carries an operating system kernel because the kernel scheduling has to be useful in a general purpose environment, but the user level scheduling deals only with threads of one application.

Other packages use some system services such as the alarm signal in some UNIX implementations, to get a clock interrupt at user level [5]. In this latter case, the application will fail if the programmer attempts to use the alarm signal.

In this work we will show how some of the kernel mechanisms and scheduling policies have been ported to user level to improve application execution times. The library interface attempts to be very general to fit the requirements of efficiency and portability to other systems.

User level scheduling

Microkernels have introduced new scheduling mechanisms. For example, now users can suggest a context switch to the kernel (hint). Then, the kernel applies its scheduling policy and selects another
thread (perhaps in another task) to execute. Users can also directly give their own processor to another kernel thread (handoff) [8]. In this case, the kernel do not apply its scheduling policy and transfers the physical processor to that thread, if it is not blocked.

As an experience, we have implemented a schedule hint and handoff mechanism into CThreads. The new primitive: `cthread_handoff(thread)` tries to give the current virtual processor -the kernel thread- to the target cthread, without caring for any other scheduling policy. There is a new call `cthread_hint()` which searches another runnable cthread according to the current scheduling policy; if there are no candidates, it simply returns control to the caller. This contrasts with the behaviour of `cthread_yield()`, which usually calls the kernel, relying on the kernel scheduling decisions.

In order to introduce user-level priorities in the CThreads library, we have added three new primitives: `cthread_set_prio()`, `cthread_get_prio()` and `cthread_init_prio()`. Applications can set the priority of their cthreads using the `cthread_set_prio(thread, prio)` library call. All cthreads with the maximum priority are scheduled in FIFO order. Cthreads with less priority are not scheduled if there are enough higher priority cthreads to fill all application virtual processors. After a `cthread_fork(-function, arg)` call, new cthreads begin execution at the default priority level.

Originally, the queue manipulation routines of the CThreads library were designed to maintain a FIFO order. We have modified such routines to insert items in queues in the right place according to their priority. Within a priority level, these routines have the traditional FIFO behaviour.

Library priorities are static, that is, the library does not attempt to change them. The priority system purpose is to have some levels of execution. In this way, applications can assign higher priorities to such cthreads that execute the more important work. Priorities can be used to control the execution order of the application cthreads, modifying them and suggesting a context switch (with `cthread_hint()`, for example).

We have provided a user-level preemption mechanism into the CThreads package, in order to be able to preempt the virtual processor and give it to another one. This mechanism, presented in section 4, allows us to supply a scheduler thread at user level that implements preemptable policies as round robin.

**Optimizations to the synchronization mechanisms**

It is possible to use the priority system in an attempt to spent less time in getting a spin lock and to reduce the time a cthread is in a critical section. This can be done lowering the priority of cthreads getting a spin lock and raising the priority of cthreads executing in a critical section.

In the standard library, if a critical section is protected with a spin lock, cthreads that find the section locked will be testing the lock during a period of time. The `spin_lock()` routine can decrease the cthread priority to the minimum value and can suggest a user context switch (cthread_hint()). It is very important to note that, in this case, the application must assure that the cthread in the spin lock is going to receive a virtual processor when the critical section will be freed. This is possible recording the cthreads waiting in a spin lock. When the cthread in the critical section leaves it (with `spin_unlock()`), the application can also decide to give its virtual processor to a waiting cthread, raising its priority to the original value (with `cthread_set_prio()` and `cthread_handoff()`).

Also working with spin locks, it is possible to modify the `spin_lock()` routine to test for the lock during a fixed number of iterations only, and, if the lock is not freed while waiting, suggest a user level context switch with `cthread_hint()`.

On the other hand, when a cthread acquires a critical section with `mutex_lock()`, it is possible to raise its user priority to the maximum value. In this way, the cthread will execute at maximum priority until it releases the lock (using `mutex_unlock()`). As the library is not preemptive, the time in the critical section will be reduced only in case the cthread blocks while is in the section, because when it wakes up, it will receive a virtual processor at the first scheduling time.

Moreover, it is possible to modify the `mutex_unlock()` routine to suggest a `cthread_handoff()` to the first thread waiting in the mutex queue. When a cthread exits from a critical section, it gives its virtual processor to another thread that wants to enter the critical section.

These optimizations are available through compilation options in the modified CThreads library, so for every application it is possible to select the library that fits the application require-
Some of these implementations are already tested in some systems at kernel level (smart scheduler in Symunix [9]).

4. Environment and performance evaluation

We have taken a multithreaded server as the archetype of applications we want to give support to, that is, those tasks whose goal is to give some service to operating system users [10].

Besides, we have implemented a CPU server, a facility to allocate processors to tasks. It is needed if we want to support a variety of programming models running on a multiprocessor. Fine grain parallel applications need to know how many processors are currently available and allocate them accordingly.

A shared memory event collection mechanism developed at GMD (JEWEL) has let us to take measures of the server process running on Mach. We have compared the standard CThreads library with the modified library.

The current implementation of the benchmark consists on some number of producer threads and some number of consumer threads. As an example, producer threads generate matrices and consumer threads calculate the matrix product. The coordination of duties use the new scheduling primitives.

Mutex and condition variables are used to ensure mutual exclusion to get shared data, following this scheme:

```c
type_t data;
mutex_t data_lock;
condition_t data_cond;
```

To get access to a data item:
```c
mutex_lock (data_lock);
while (!freeData ());
mutex_unlock (data_lock);
```

To release a data item:
```c
mutex_lock (data_lock);
release_data ();
mutex_unlock (data_lock);
```

The producer and consumer synchronization points follow this pattern:

### Producer

```c
WHILE (TRUE) {
    // get data item
    // generate data

    // Get mutex to select a consumer
    mutex_lock(consumers_lock);
    while (consumers_busy())
        condition_wait(consumers_available, consumers_lock);
    c = fetch_consumer();
    c->free = FALSE;
    // Release mutex
    mutex_unlock(consumers_lock);

    // Get mutex to become available
    mutex_lock(consumers_lock);
    myself->free = TRUE;
    // Wakeup any waiting producer
    condition_signal(consumers_available);
    // Release mutex
    mutex_unlock(consumers_lock);

    // Block while no work to do
    mutex_lock(myself->waiting_for_work);

    // Optionally, give virtual processor
    // to consumer thread
    // (see options below)
}
```

### Consumer

```c
WHILE (TRUE) {
    // Get mutex to become available
    mutex_lock(consumers_lock);
    myself->free = TRUE;
    // Wakeup any waiting producer
    condition_signal(consumers_available);
    // Release mutex
    mutex_unlock(consumers_lock);

    // consume data
    // release data item
}
```

Figure 1: Producer and consumer synchronization

In some cases, when we produce some data and we know which thread is going to use it, we can give control to this thread using the primitives added to the CThreads library: `cthread_handoff()` and `cthread_hint()`. Using them, if it is possible, the library performs a user level context switch.
instead of leaving that decision to the kernel. For example, when a producer thread has generated
some data to be processed, it gets a free consumer thread and gives control to it (Figure 1).

As the producer knows which consumer thread is waiting for the generated data, we will
calculate four possibilities to give the virtual processor to the consumer (see Figure 1). Mutex_un-
lock() puts a consumer thread in the ready user level queue. Then, the producer can follow one of
these constructions:

- With the standard CThreads library:
  - NOP: the first option is do nothing. Usually, users will get this option. In this case, sched-
    uling will occur later, when the kernel selects the consumer or when the producer thread blocks.
  - YIELD: the second option is to execute a cthread_yield(), which attempts a user level con-
    text switch. If it fails, calls the kernel to let it choose another thread.

- With the modified library:
  - HANDOFF: the first option is to call cthread_handoff(), that attempts to give the producer
    virtual processor to the consumer. If it fails, it returns control to the producer.
  - HANDOFF+HINT: the last option first also tries a cthread_handoff() and, if it fails,
    attempts a cthread_hint(), which gives the virtual processor to any user level ready thread. If this also
    fails, it returns control to the original thread.

The library routine cthread_handoff() misses and cannot perform a context switch when the
target cthread is already attached to a virtual processor (kernel thread). This happens when the num-
ber of kernel threads is not adequately limited. If the programmer goal is to have the maximum num-
ber of user level context switches, he has to determine the number of kernel threads that matches the
inherent parallelism of his application.

The cthread_hint() function fails when there are no user-level ready cthreads.

In order to isolate the application from external interferences, with the assistance of the CPU
server, a machine partition with dedicated processors is allocated during the execution of the applica-
tion. Kernel context switches are then confined within the application virtual processors.

We have run the example of the producer-consumer problem, with four threads of each class,
varying the physical parallelism (processors) and the number of virtual processors (kernel threads),
for each of the four policies trying to hand over the execution context at user level explained above.
The machine is a DEC433MP with four i486 processors running Mach 3.0 MK4.1 (OSF/1 1.1)
microkernel.

Figures 2 to 4 show the benchmark execution time depending on the number of kernel
threads and physical processors. Although four processors is a narrow experience, we though the
size of the chosen problem is adequate and the consequences are scalable with larger architectures
and larger parallel applications. Figure 5 presents the number of handoff misses in those experi-
ments.

Comments to the experiences

Uniprocessor scenario

When the application runs on a dedicated physical processor with one to four virtual proces-
sors, its behaviour is nearly constant (Figure 2). Each producer thread attempts to give their virtual
processor to the selected consumer, and it almost always hits (Figure 5). The kernel is unaware of
this changes at user level. In this case, the HANDOFF execution time is better than the NOP or
YIELD options (about 10 - 20%).

On the other hand, when the number of kernel threads exceeds the number of producers (4),
there are some consumer threads with their own virtual processor. If a producer attempts to give its
kernel thread to them, it misses (Figure 5). Moreover, kernel scheduling overhead increases and
affects the intended user level scheduling. In fact, looking at the specific execution time results, val-
ues differ a lot between different executions. Unpaired attempts of give control produces unpredict-
able effects. The YIELD option produces more kernel context switches and has unpredictable paths.

With the NOP option, the producer continues until blocks. The other policies are worse at this
point because they spend user time at handoff misses.

The scheduling discipline is very sensitive on a uniprocessor. When the number of processors
increases, the policies are much less important. This scenario demonstrates that even with no phy-

cisal parallelism, the constraint of the number of kernel threads is a good practice.

**Two processors scenario**

In this situation we have two dedicated processors. The application has enough threads (8) to
distribute correctly between the kernel threads. That is the reason why a great number of *cthread_handoff()* hit and hand over the virtual processor from the producer to the consumer.

The producer wakes up the consumer (*mutex_unlock()*) and, before the kernel can dispatch a
kernel thread to run it on, the producer give to it its virtual processor.

This is the best case for all the options; we have found the right granularity to this experiment
that match the maximum hits of the handoff mechanism (Figure 5).

**Four processors scenario**

When the physical parallelism increases, consumers complete easier and quickly. Producers
have to wait less time to get a free consumer. Producers do not stop for resource shortage (less buff-
ers are needed).

But, in the other hand, the kernel is free to assign a kernel thread to a consumer when a pro-
ducer wakes it up. So, although the producer knows that the consumer is available and tries to give
its virtual processor to it, it founds that 16% times (Figure 4, 4 kernel threads) is already running
because, between the *mutex_unlock()* and handoff the kernel has already dispatched it.

When the application has more than four kernel threads, some consumers have a virtual pro-
cessor associated. They seem to be running at user level, then handoff misses but the kernel may
have not selected them for running. In this case, the application suffers more context switches than
expected. The application cannot achieve its maximum speed.
Although been an heuristic remark, the benchmark runs better with two physical processors because it has reached the right machine power for this application. Add more processors, and more movement will appear in the kernel side, which conflicts against the user scheduling. User time increases a few due to user level mutex synchronization -more often when there are more processors-, but kernel time increases a lot (up to 150%).

5. User-level preemption and context switch

We notice that user-level packages usually do not allow preemption at user level or, if implemented, it is based on the UNIX signals mechanism. That ties the library to the UNIX environment and also prevents the application of using this mechanism for its own purpose.

We have enhanced our library to support timer expiration upcalls, in a manner similar to Scheduler Activations [11] and First-Class Threads [12] do. A new kernel call sets a one shot or at fixed intervals notification to the user which has to provide a routine and a stack to manage the software interrupt. The kernel brings the state (registers) of the preempted thread as a parameter to the upcall routine. Then, the library is in charge of doing a user-level context switch to the appropriate cthread, allowing round-robin, handoff or whatever scheduling and timeout handling at application level. To prevent deadlocks, we inhibit new interrupts during the upcall handling, quickly save the preempted state, and then handoff to a user-level scheduler thread that, as a average cthread, chooses which to continue.

Table 1 presents the execution times spent in the upcall service routine and, for the user scheduler, to select a thread and perform a context switch. Those times include the management of the CThreads ready queue and the context switch.

<table>
<thead>
<tr>
<th>Action</th>
<th>Time (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Upcall timer handler</td>
<td>36.22</td>
</tr>
<tr>
<td>User-level scheduler</td>
<td>51.06</td>
</tr>
</tbody>
</table>

Table 1: Execution times for the upcall timer handler and select a new thread

The application, besides the ability to decide its own internal scheduling, can also benefit from a cut down in scheduling overhead if it is done in less time at user level. Table 2 shows that there is an order of magnitude difference in cost between user level and kernel level thread context switch.

<table>
<thead>
<tr>
<th>Context switch routine</th>
<th>Time (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>cthread_handoff()</td>
<td>16.29</td>
</tr>
<tr>
<td>cthread_hint()</td>
<td>16.26</td>
</tr>
<tr>
<td>cthread_yield()</td>
<td>13.90</td>
</tr>
<tr>
<td>thread_switch() [kernel call]</td>
<td>134.86</td>
</tr>
</tbody>
</table>

Table 2: Execution times for user-level (handoff, hint, yield) and kernel-level context switch.

The primitives we have added to the package spend a little more time because they select new cthreads by priority, instead of cthread_yield() that works FIFO. The thread_switch() measured performs a kernel context switch between two kernel threads of the same task.

The best the application can do is to limit the number of kernel threads to the amount of physical processors it has allocated, set a timer preemption event, and carry out its own flow scheduling.
6. Conclusions and lessons learned

Simpler kernel scheduling mechanisms, if associated with appropriated user level scheduling policies, permit us to obtain better performance. This is due to the fact that the application knows better than the system the behaviour of its own threads, and because the user mechanisms are cheaper.

In the overall executions, when the application creates more user level threads than kernel threads, the modified library executes more efficiently than the standard CThreads package. The programmer knowledge, or implemented implicitly into the library functionalities, is able to foresee and force a successful continuation. The interference of the kernel is reduced, due to the hits of user-level context switch at synchronization points.

If the number of virtual processors exceeds the adequate to the application granularity (the number of instructions between two synchronization points in the sequential execution of a control flow), although this number is till now a heuristic, the fair kernel scheduling do not suit the application needs.

In the other hand, when parallel algorithms can adapt its execution flows to the number of processors assigned, the kernel should not interfere making other scheduling decisions and should inform the application about any changes in resources allocation.

In the parallel applications support field, the microkernel might manage the allocation of processors to tasks but might not schedule threads at a quantum expiration or priority recalculation rate. Kernel should notify the user of system events that may affect the job. This allows the thread package to respond to the event in the most appropriate manner for the application. We are working in this trend, following the scheduler activations mechanism proposal [11].

7. References

[1] “UNIX scheduling for large systems”  

Marisa Gil, Toni Cortes, Angel Toribio and Nacho Navarro  
OSF Research Workshop, 22-24 June 1993, Grenoble (France)

E.C. Cooper and Richard P. Draves  

Randall W. Dean  
School of Computer Science, CMU, March 1993

Thomas W. Doeppner Jr.  
Brown University, Technical Report CS-87-11, June 1987

IEEE, P1003.4a/D6, February 1992

Frank Mueller  
USENIX, Winter'93, San Diego, CA

David L. Black  
USENIX Winter'91, Dallas TX
Jan Edler, Jim Lipkis, and Édith Schonberg,
NYU Ultracomputer Research Laboratory
Workshop on UNIX and Supercomputers, USENIX, September, 1988

Open Software Foundation and Carnegie Mellon University
Keith Loepere Editor, January 1992

Thommas E. Anderson, Brian N. Bershad, Edward D. Lazowska and Henry M. Levy

[12] “First-Class User-Level Threads”
Brian D. Marsh et al.