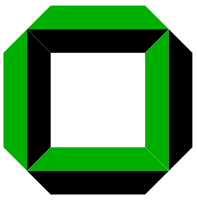


Universität Karlsruhe (TH)
Institut für
Betriebs- und Dialogsysteme
Lehrstuhl Systemarchitektur

Batch Scheduling for the L4Ka::Pistachio Microkernel

Andreas Mähler

Studienarbeit

Verantwortlicher Betreuer: Prof. Dr. Frank Bellosa
Betreuende Mitarbeiter: Dipl.-Inf. Jan Stöß
Dipl.-Inf. Raphael Neider

18. Januar 2008

Hiermit erkläre ich, die vorliegende Arbeit selbständig verfaßt und keine anderen als die angegebenen Literaturhilfsmittel verwendet zu haben.

I hereby declare that this thesis is a work of my own, and that only cited sources have been used.

Karlsruhe, den 18. Januar 2008

_____

Andreas Mähler

Abstract

The L4 microkernel interface defines a dedicated system call to control thread scheduling parameters: `Schedule`. The current version of `Schedule` takes just one thread argument, which implies that only one thread can be modified per invocation. However, many applications (e.g. virtual machines or time-sharing systems) internally group threads together and manage them accordingly. Whenever large sets of threads (i.e. their scheduling parameters) must be manipulated, the necessary information cannot be passed to the kernel at once. This thesis proposes modifications to L4's scheduling interface, which allows a user-level scheduler to have the kernel set scheduling parameters for multiple threads at a single blow.

In multiprocessor systems, manipulation of threads that reside on a CPU other than the scheduler thread's (especially Processor migration, which is the most complex scheduling task) is performed by an in-kernel mechanism. In the current version of Pistachio, the mechanism is only designed for single threads (which is all it needs to do, because the interface can only deliver a single thread ID to the kernel level). The altered interface also makes it possible to introduce a new mechanism that is optimised to perform „batch scheduling" and therefore induces less overhead.

The implementations of the proposed designs for the interface and the mechanism will be evaluated using a microbenchmark that sets certain scheduling parameters for a growing set of destination threads.

Contents

1 Introduction

As research has shown [1], hypervisors (or virtual machine monitors) [2] and microkernels – although having different design motivations – possess a significant affinity, which is the reason why microkernel systems can be easily adopted/used to cater virtualisation tasks.

L4Ka::Pistachio is a second generation microkernel that is developed at the University of Karlsruhe and implements the L4 X.2 API [3]. This API defines a minimalistic set of abstractions that cover the underlying hardware resources of a common computer system [4]. Due to its minimality, which directly derives from the microkernel design paradigm, Pistachio is also used for many applications that differ from its original purpose: to be utilised as a foundation for a traditional multi-server operating system [5, 6]. Today, it is mainly used as a hypervisor which multiplexes the system resources to a set of guest (operating-)systems [7, 8]. A number of popular operating systems have already been adapted to run on top of L4. Examples include L4Linux [9] and L4/Darwin [10], which are *paravirtualised* [11], and more recent Linux-Kernels (as of writing the newest one is Version 2.6.9), which are *pre-virtualised* [7].

In order to efficiently fulfill its job, the virtualisation architecture usually makes heavy use of L4 Threads. Examples are virtual CPUs which can be mapped to kernel level threads by an in-place VMM [7, Section 2.1] or a guest system which completely relies on the thread mechanisms and the scheduling of L4. Especially when Pistachio is used to host many virtual environments (as it is the case for server consolidation), there are many *sets* of threads that it has to manage – one set for each virtual environment. To perform management tasks (i.e. migration, swapping or the manipulation of priority levels and compute time) for these sets, all threads of a set must be manipulated in the same way. If these manipulations occur frequently, high efficiency is crucial. (An example is a server consolidation scenario where the migration of whole guest operating systems among the CPUs in an SMP system is performed according to a certain load balancing policy.)

1.1 The Problem with Single Thread Scheduling

With today's implementations of the L4 X.2 API, the only way to set the scheduling parameters (to perform the management tasks mentioned above) for a set of destination threads is to iterate over its elements in user mode and invoke the `Schedule` system call for every single thread. Switches to kernel mode, however, are very time-consuming on many hardware architectures. This is particularly true for the processors of the IA32 family, on which this thesis focuses. Even by using its optimised `SYSENTER` and `SYSEXIT` commands, which bypass many processor internal operations (that support legacy software), L4::Pistachio takes 150 to 200 cycles to enter/exit kernel mode on the Intel Pentium 4 processors [12]. As this is too much overhead to achieve proper scalability, our thesis presents a different solution.

1.2 Approach: Batch Scheduling

All necessary information that is needed to schedule a whole set of threads is already available in user space when the `Schedule` system call is going to be made. Nevertheless, this information cannot be passed to the kernel at once because of the current L4 ABI/APIs shortcomings.

The main objective of the thesis is to define a generalization of the `Schedule` call that takes a *set* of threads as destination parameter and therefore carries out the batch processing in kernel mode. This reduces the kernel entry overhead for many threads to a single enter/exit at best. We will present modifications for the L4 binary interface (ABI) and the L4 programming interface (API) that enable "batch scheduling" for the kernel. Source-level compatibility with the current interface and the ability to implement the altered user-level interface on top of an unmodified kernel are desireable for an evaluation that utilises existing applications.

The main problem is how the parameters are passed to the kernel: In L4, all parameters for a system call are generally loaded into the processor's general purpose registers before the processor is switched into kernel mode. These registers, however, are rare on certain architectures. Therefore, as it is already the case for IPC calls, a static per-thread memory object that is always mapped to physical memory is utilised to emulate an adequate number of registers.

Thread migration, one of the scheduling operations, is performed by an in-kernel mechanism. Because of L4's current shortcomings, it is only designed to migrate single threads (in Pistachio) and must therefore be invoked for every thread in the set. To tap the full potential of the altered interface (which makes all necessary information that is required to process a set of threads available at once), we will propose a new mechanism: Instead of triggering the remote CPU for every cross-processor (XCPU) scheduling operation, scheduling requests are accumulated in a mailbox that is assigned to the remote CPU. This CPU is finally triggered when *all* scheduling requests have been submitted.

The implementations of the design proposals (interface and mechanism) will be evaluated using a microbenchmark that sets certain scheduling parameters for a growing set of destination (schedulee) threads.

1.3 Related work

To our best knowledge, there is no documentation of similar approaches yet.

1.4 Outline

The remainder of this thesis is structured as follows: Chapter 2 gives the necessary background information on how system calls and thread scheduling are defined in L4 and implemented by Pistachio. Afterwards, the modifications that we propose for the L4 X.2 ABI/API in order to support „batch scheduling" are presented in chapter 3. Chapter 4 describes certain implementation issues and analyses the effects. Chapter 5 gives a conclusion and suggests future work.

2 On L4 and Pistachio

This section presents details about L4 (the microkernel specification) and the internals of the L4Ka::Piatachio kernel that are required to understand the design proposal and implementation descriptions of the following chapters. The architecture dependent explanations always refer to IA32.

2.1 Interfaces

L4 specifies three interface levels:

- The **Convenience Programming Interface (CPI)** presents the highest level of abstraction that L4 specifies. It consists of several specialised wrapper functions and constant values that are meant to increase readability and maintainability for the programmer (thus increasing convenience) and is intended to be used by the programmer.
- The **Generic Programming Interface (GPI)** maps the kernels binary interface to the particular programming language. It only defines the essential counterparts of the function calls and data types that are required to interact with the kernel. Therefore, L4 requires the GPI to be fully implemented; the CPI is optional.
- The **Application Binary Interface (ABI)** defines the lowest level of interaction between microkernel and user code. While the other interfaces are platform independent, the ABI defines how the GPI is mapped to the specific hardware instructions of the target architecture.

2.2 User Data Objects

L4 defines data objects that are visible to user code, which uses them to invoke system calls (among other things).

2.2.1 Kernel Interface Page

The Kernel Interface Page (KIP) is a memory page that is always mapped into every address space. It is maintained by the kernel and cannot be modified by user code. Besides providing information about the kernel and the underlying hardware, it also contains stub code blocks for the system calls. With the exception of some constraints like alignment or kernel memory segments, the address of the KIP can freely be chosen for any new address space. Therefore, there exists an architecture dependent mechanism, which determines this address for the user code.

2.2.2 User Level Thread Control Block

In L4, the data that is associated with a thread divides into two parts: User accessible and kernel-only data. All TCB elements that do not influence threads in another protection domain (address space) on modification are stored in a data object in user memory and can also be modified by user code. This is done to increase performance, as it saves kernel mode switches to set and retrieve thread properties (like the thread ID). Because the UTCB is also read and written by the kernel, it must be located at an address that is known by the kernel. Therefore, the second L4 specific object in every address space is the so-called **UTCB area** which is an array of UTCBs. As it is the case for the KIP, the UTCB area's address (and size) is specified at address space creation, and the UTCB area is also guaranteed to be mapped into the address space at all times.

2.2.3 Virtual Registers

A virtual register (VR) is a static per-thread object that is implemented by the kernel. Depending on the underlying architecture, it is either mapped to a hardware register or to a memory location, for which it is guaranteed that no pagefault is triggered on access. Due to IA32s shortage of general purpose registers, the L4 IA32-ABI maps nearly all VRs to memory locations in the UTCB.

2.3 System Calls

2.3.1 Control Flow

L4 System calls are performed by calling a system call address in the KIP. The offsets (relative to the KIP's base address) for the system call entry points are also obtained from this data object (In practice, obtaining the proper start address for the intended system call is encapsulated in a library that provides the Generic Programming Interface – see Section 3.2). At every entry point, there is a code stub which either assists the corresponding kernel code to reduce compute time in kernel mode (Pistachio's `Schedule` system call, for example, accepts a parameter in two different formats, but the kernel function does not, so this parameter is transformed by the stub) or even performs the whole operation without a mode switch (as it is the case for Pistachio's `ExchangeRegisters`, `Lipc`, or `SystemClock`). If there is a kernel mode entry, the according instruction (IA32: `SYSENTER` for `Ipc`, `INT` otherwise) is also located in the stub. Pistachio only uses one software interrupt number for system calls; control is dispatched to the appropriate function according to the user instruction pointer that was saved to the kernel stack.

2.3.2 Parameters

The parameters and return values for L4 system calls are normally transferred through the processor's general purpose registers. On IA32, the register bank is automatically saved to the threads kernel stack when a software interrupt occurs. This "exception frame object" is later used to call the system call function with the proper parameters; if the function has return values, they are also stored there. Errors are reported to the user code by returning a system call

dependent error value. Details on the error cause are stored in a dedicated variable in the user threads UTCB.

Although the general purpose registers are sufficient for almost every system call, this is not always the case. Especially `Ipc` needs up to 64 words to store a message – not counting the other parameters. This is the reason why the UTCB contains a set of virtual registers, called the Message Registers (MRs). The `Ipc` operation transfers the MR's contents of the sender thread to the receiver thread's MRs. Although this was the MR's original purpose, they are now also used to pass parameters for the `Unmap` and the `MemoryControl` system call, making the name "Message Registers" somewhat obsolete.

2.4 Scheduling

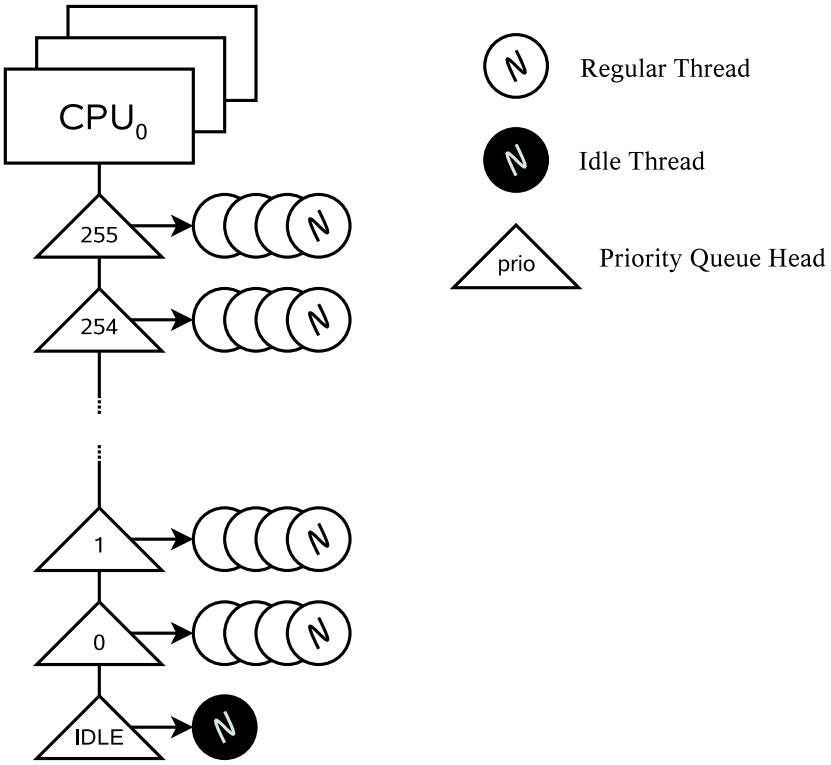
2.4.1 Overview

L4 Threads are scheduled to the system's CPUs according to a fixed in-kernel scheduling policy. (An in-kernel policy is contradictory to the design goals of minimality and flexibility that were set for L4, and the approach was only taken for performance reasons. As a consequence, the scheduler was designed to offer high flexibility. An approach to finally remove the last in-kernel policy has been presented by Stoess et.al. [13].) The decisions that are made by Pistachio's scheduler are influenced by a set of per-thread parameters (e.g. timeslice control, the threads priority or the CPU on which the thread is running in an SMP system). These parameters can be modified by a dedicated "scheduler thread" (or any other thread residing in the same protection domain) that is assigned to each thread upon its creation.

The scheduler defines 256 priority levels for the threads. These priorities are hard: A thread is never scheduled to the CPU when another thread with a higher priority level on the same CPU is ready to run as well. Threads with the same priority level are scheduled in round robin fashion.

The CPU's idle thread is the only thread which resides on the lowest priority; it is scheduled when a running thread blocks or runs out of compute time. When activated, it calls the CPU's scheduler object to find the next runnable thread. If there is no such thread on the CPU, the idle thread temporarily halts the CPU. The amount of compute time that is assigned to a thread is manipulated through a time-control parameter. This parameter defines the total quantum and the timeslice length. When a thread is running (or has donated its compute time to another thread), its so-called current quantum is decreased. Upon quantum exhaustion, the thread is preempted, another thread is selected to run, and the current quantum is reset to the timeslice length. The total quantum is also decreased whenever the thread is running, but in contrast to the total quantum, a message is sent to the scheduler thread whenever it is exhausted. It is up to the scheduler thread to fit its schedulee with a new total quantum to make it ready to run again. As it is very common to provide threads with a timeslice that lasts forever, the total quantum is set to infinity in most cases.

Figure 2.1: Conceptual View of the Pistachio Scheduler



2.4.2 System Call Signature

```
Word Schedule( ThreadId dest,
               Word     TimeControl,
               Word     ProcessorControl,
               Word     Prio,
               Word     PreemptionControl,
               Word*    old_TimeControl )
```

2.4.3 Pistachio Thread Migration

In SMP systems, Pistachio uses a message passing system for cross processor communication. It guarantees mutual exclusion on per-CPU data objects. Whenever a CPU must alter another CPU's data, it sends a message to the other CPU which invokes the according operation locally.

One static mailbox object per CPU is held in memory. These mailboxes are basically ring buffers that contain message entries of a fixed size (aligned to the underlying architectures cache

lines). The message consists of a function address and up to eight parameters. A message transfer is performed by adding a new message to the receiving CPU's mailbox and triggering it by invoking an inter processor interrupt (IPI). The mailbox of the receiving CPU is processed on every IPI and on every timer interrupt by calling the function pointers with the corresponding parameters.

Whenever `Schedule` is called with a valid `processor_control` parameter, a migration function is called on the destination thread's TCB. The function basically unchains the TCB from the local processor's ready queue and sets its CPU field to the destination CPU's ID. If the thread is not blocked, it must be added to the destination CPU's ready queue by sending a message to the destination CPU.

3 Interface Design

In this chapter, we present the proposed modifications to the L4 ABI/API to provide a new `Schedule` system call that accepts a set of thread IDs (instead of just one) as destination for its operation. Our goal is to achieve as much source-level compatibility (because most of the software which is developed for L4 is distributed as source – binary compatibility is not necessary yet) with existing program code as possible. It should also be possible to implement the proposed interface for an unmodified kernel so that different applications can be evaluated easily.

This design proposal will be explained following a top-down scheme. The design of the convenience programming interface (CPI) is followed first by the generic programming interface (GPI) and then by the application binary interface (ABI) for IA-32.

3.1 Convenience Programming Interface

Status Quo

At present, the CPI only defines four functions that are derived from the `Schedule` system call, which is part of the GPI. The CPI receives the necessary scheduling parameters, the destination thread's ID and a pointer to the memory location where an output value is to be written. Because these are only very few word-typed parameters, there is no necessity for a generic `Schedule` function in the CPI, which hides non-trivial parameter-passing.

Analysis

A generic `Schedule` function which hides low-level kernel- and hardware details must be defined. This function is the new foundation for the specialised CPI functions. There are several design parameters that must be taken into consideration:

- **Parameter Passing:** Thread IDs (like other data) can be passed to a library function in different ways. Committal via the stack (as function parameters) is very intuitive for the programmer, but not all programming languages offer infinite parameter lists. Therefore, it is not an option, as the L4 specification is designed with language independence in mind. A shared memory object (stored at a pre-defined location in memory that is checked by the function code) is another solution, but it also has major drawbacks: The user code (e.g. the L4 library) must provide one object per thread, and the user must be careful not to overwrite the object. We therefore chose a third solution: The user passes a reference to a data object as a parameter to the function. This data object represents a set of threads and is composed by the user; such a practice is also very flexible because the user is able to use the same format for thread group management and pass it to the function as-is.

- **Data Object:** We have chosen the simplest form, which is an array of thread IDs. There are two ways for the function to determine the array size: Either the number of TIDs is passed as an extra parameter (or can be found in the data object itself at a fixed position) or the array is terminated by a special data sequence (i.e. a character string is normally terminated by 0). We have chosen the first alternative for consistency reasons – it is already widely used by other function calls, like `Unmap`.

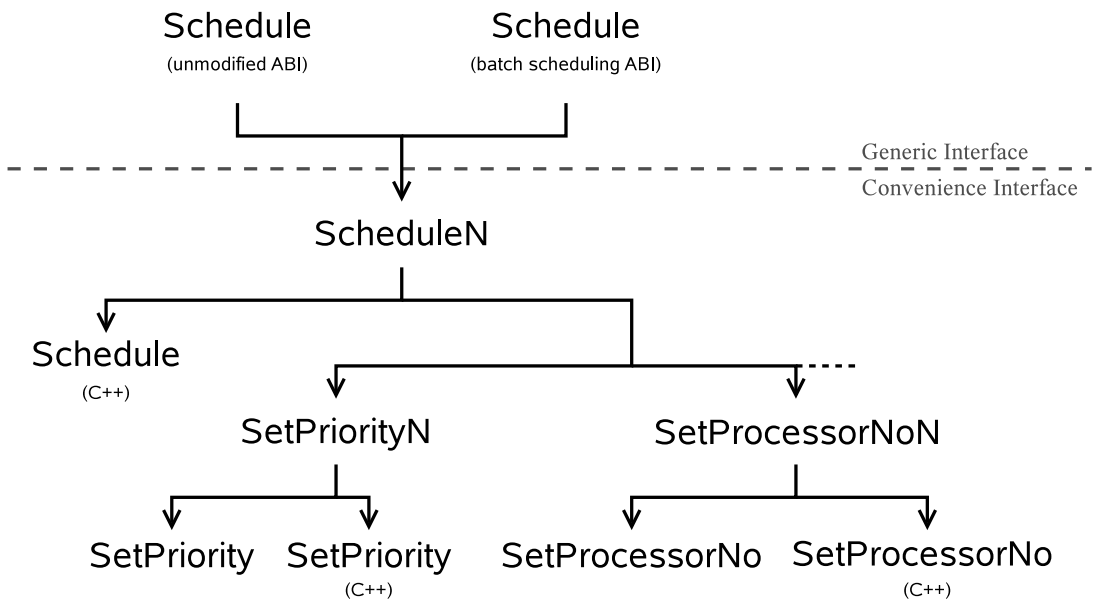
- **Return Values:** Two result words are returned for each thread schedule operation. The simplest solution is to let the user provide a memory address where the variables can be saved. We decided to use two separate arrays (instead of one array that contains composed objects) with two pointers that are passed to the function. This enables the user to express disinterest in the return values (by passing a pre-defined invalid address). If return values are not required, optimisations are possible.

- **Error Handling:** The scheduling operation can fail for every thread. This is no problem for a one-thread schedule, but the semantics of the generic `Schedule` function must be refined. What happens if the scheduling operation fails for one thread? The operation can be aborted, and all previous threads are guaranteed to be in their original state (transaction semantics). Although this behavior is very convenient for the programmer, it cannot be implemented if the kernel only accepts a finite number of threads (like the unmodified kernel, for which the new function should also be implemented). The function's performance is also decreased. An alternative is that the function schedules all threads, and the user is notified of a failure by the per-thread result values. We chose not to implement this version, as L4 specifies that error details are transmitted through a single TCR, which implies that this can only be done for one thread. The third alternative is to change the semantics of the parameter that specifies the array size from "in" to "in/out". This enables the function to proceed until the first error, where the operation is aborted. The output value of the num parameter specifies the array index of the faulting thread's ID; then, the user can handle the error situation and proceed with the scheduling, if necessary.

Proposed Design

As in the case of `Unmap`, we introduce a function that hides the parameter passing to the kernel. Figure 3.1 illustrates how all specialised functions that are part of the CPI are now derived from this new generic `ScheduleN` function instead from the GPI function. The function itself can be implemented upon both GPIs. To retain the source level compatibility, all specialised functions are also defined for single threads. If it is supported by the programming language, the function names are overloaded accordingly (this is indicated by the (C++) sign). Overloading also enables full source level compatibility, as the original GPI `Schedule` function can then be emulated.

Figure 3.1: Proposed function hierarchy



The following function is intended to be used by the programmer to set all scheduling parameters for a set of destination threads [1]:

```
Word ScheduleN( ThreadId* dest,
                Word*     n,
                Word      TimeControl,
                Word      ProcessorControl,
                Word      Prio,
                Word      PreemptionControl,
                Word*     result,
                Word*     old_TimeControl )
```

In order to support the manipulation of a set of threads, the function must receive a pointer to a data object which represents it. This data structure is an array of L4 thread IDs. When the function is called, the threads that correspond to the first `n` thread IDs in the array are manipulated according to the other input parameters.

The function returns an aggregated result value (which had to be altered from the L4 X.2 version in order to support the aggregation) as described below. If per-thread return values are required, the `result` and `old_TimeControl` parameters must be set to pointers. The vectors of result- and timeControl-values are then saved to the specified memory locations. Both

[1] For a complete reference of the new scheduling CPI see Appendix A

parameters must be set to `ScheduleNoDetails` otherwise.

result:

$\sim_{(24/56)}$	$cwspride$

The $cwspride$ bits, if set, indicate the occurrence of at least one thread whose operation resulted as: e=Error, d=Dead, i=Inactive, r=Running, p=Pending send, s=Sending, w=Waiting to receive, c=Receiving.

3.2 Generic Programming Interface

Status Quo

In the GPI, the `Schedule` system call takes the destination thread ID as a normal parameter and stores it according to the target architectures ABI specification before calling the stub code. Obviously, this technique is not applicable for a set of threads.

Analysis

- **Parameter passing:** A naive solution would be to make the `Schedule` system call accept a pointer to the array of thread IDs instead of just one thread ID. This, however, would contradict the design of L4: Pagefaults must not happen in kernel mode (the kernels access to the array could cause a page fault as it is not guaranteed that the page which contains the array is mapped). Therefore, we chose a different approach: The Message Registers are used to pass the parameters to the kernel.
- **Return values:** The same drawbacks that apply to parameters also apply to return values. To maintain consistency among the system calls, the proposed `Schedule` call also uses the 64 MRs to return per-thread status values to the caller. In contrast to `Unmap` and `MemoryControl`, though, there are *two* return values - not just one. Our approach to this problem is the constraint that the return values are only available for the first 32 threads. This means that only blocks of at most 32 thread IDs per schedule call must be passed to the kernel if all return values are of importance.

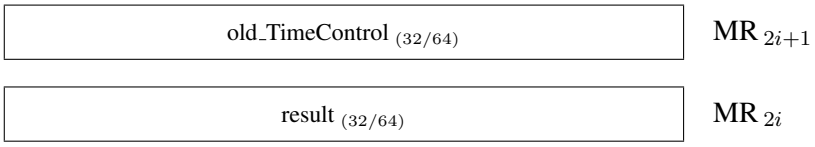
Proposed Design

All functions that are part of the convenience programming interface are derived from the following function:

```
Word Schedule( Word* n,
               Word  TimeControl,
               Word  ProcessorControl,
               Word  Prio,
               Word  PreemptionControl )
```

Instead of passing an array of thread IDs to the function, the destination IDs must be loaded into the message registers. The parameter `n` now specifies that the thread IDs from MR_0 to MR_n are supposed to be manipulated. This now implies $0 < n \leq 63$ (in contrast to the CPI where the parameter n can have any value). The call has to be repeated if there are still threads left that are to be manipulated.

`Schedule` delivers two result words for each of the first 32 threads. They are also saved in the message registers and can be accessed in the following form:



The values correspond to the ith thread where $0 < i \leq 31$ while the return values for $32 < i \leq 63$ are omitted.

3.3 IA32 Binary Interface

As described in section 2.3, the parameters that are passed to the GPI function must be loaded into general purpose registers (hence, they are not loaded into the message registers). The register assignment was also kept in consistency with the other system calls. Notable is the fact that a pointer to the current thread's UTCB must be loaded into the `EDX` register. This is done to save the kernel the effort of determining the UTCB address.

n	EAX	− **Schedule** →	EAX	*n*
prio	ECX		ECX	*result*
time control	EDX		EDX	∼
MR_0	ESI	call *Schedule*	ESI	MR_0
UTCB	EDI		EDI	≡
processor control	EBX		EBX	∼
preemption control	EBP		EBP	∼
−	ESP		ESP	≡

4 Evaluation

4.1 Implementation

This section contains a description of the user- and kernel-level modifications that we performed on Pistachio.

4.1.1 User Level

This section describes the glue between the CPI that was presented in section 3.1 and the GPI from section 3.2. The glue is basically the implementation of the CPI's `ScheduleN` function, because all other CPI functions are derived from it. This function does not change the semantics of the underlying `Schedule` system call, which is defined in the GPI, but it hides kernel-specific details from the user: Thread IDs must be loaded into a finite number of message registers. Because the number of MRs is finite and the array of thread IDs that is passed to the CPI function is almost[1] infinite, the GPI function must eventually be called repeatedly with blocks of thread IDs. The size of these blocks depends on whether the user is interested in the return values `result` and `old_TimeControl` that are generated by the operation for each thread. There are two paths in the implementation. The values of the `result` and `old_TimeControl` parameters that are passed to the CPI function determine which path is taken:

- The so-called **generic path** is executed when the parameters contain pointers to memory locations where the arrays containing the return values are to be saved. Because the kernel only returns these values for the first 32 threads, the implementation only passes a maximum of 32 threadIDs to the kernel at once in order to receive the values for each thread, making it possible that they are written to the result arrays.
- By setting the memory addresses to `ScheduleNoDetails`, the user expresses disinterest in the return values. Then the **fast path** can be executed; in this path, a maximum of 64 threadIDs can be passed to the kernel at once. The return values are ignored.

The changes that we made to the GPI and the ABI are straightforward and can easily be derived from the related specifications in the sections 3.2 and 3.3.

4.1.2 Kernel Level

We have adapted the internal support functions and macros (KIP stub code, in/out parameter handling) that cater the system call function. The function itself was changed to perform "batch scheduling". We will also present the design of an optimised XCPU scheduling mechanism.

[1] It is only restricted by the address space and the range of the parameter `n`

Generic

Three loops have been introduced:

1. The first loop retrieves the TCB pointers from the thread IDs and checks for error conditions. In case of error, the loop is aborted, and the upper bound of the following loops is set to the last sane thread item. This means that all following items, including the erroneous one, are not processed. After the invocation, the MRs do not need to be read again and are therefore free to be used to store the return values.

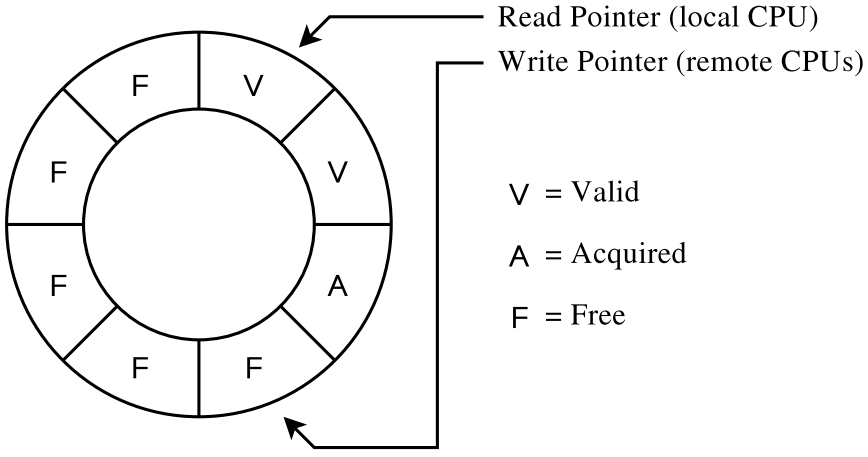
2. The actual processing happens in the second loop. According to the parameters, the proper functions for scheduling and migration are called on each TCB.

3. The third loop processes at most the first 32 threads, as it saves the return tuples (result, old_TimeControl) to the MRs.

Optimised

Pistachio's current XCPU-mechanism is a bottleneck, as it can only deliver single messages, and therefore an IPI must be triggered for each remote scheduling operation (e.g. migration of a ready thread). An alternative design of the in-kernel mechanism can address this issue. We have introduced a static per-CPU mailbox object which stores scheduling requests. If a remote CPU must alter a CPU's scheduling data structures (e.g. the ready queues), the according message is appended to the CPU's mailbox. The only remaining XCPU-message is a trigger message that is sent to the destination CPU once all pending scheduling requests are saved. (Therefore, the overall number of XCPU messages is reduced significantly.)

Figure 4.1: Ring Buffer of the Scheduling Mailbox



The scheduling mailbox itself is constructed as a ring buffer. Whenever a remote CPU sends a scheduling request to the mailbox, it marks the first free entry as „acquired" and increments the

write pointer. (The atomicity of this operation is secured by a spinlock. Acquiring and writing is split up to minimise the amount of time in which the lock is held by a CPU.) When the data is written to the entry, the remote CPU marks it as „valid". As soon as all scheduling requests are posted, the destination CPU it triggered. It then starts processing the scheduling requests (and incrementing the read pointer / resetting the entries to „free") until the first non-„valid" entry is reached.

4.2 Microbenchmark

To illustrate the effect of the improved system call, we have implemented a special benchmark. It is a user program that creates one scheduler thread and a number N of schedulee threads. The schedulees run in an infinite loop on a lower priority level than the scheduler (in order to prevent corrupted measurements caused by scheduler preemption). By using the kernel-specific implementation of the new CPI, the scheduler then applies new scheduling parameters to the schedulees. The elapsed time is also measured by the scheduler using the Pentium4's `RDTSC` instruction. All time values below indicate the number of elapsed cycles from the invocation of the CPI function to the return. However, this does not necessarily imply that the operation has been completed on all CPUs, as SMP scheduling operations (like thread migration or changing the scheduling parameters of a thread that resides on a different processor) are performed asynchronously.

The test runs were ranging from $n = 2$ threads to $n = 100$ threads (with a step size of two threads). The test operation (without return values) was performed 100 times for each n, and the listed/plotted time is the mean value. For each test series, the test runs with the modified kernels ("Batch Scheduling ABI" and "Optimised") are benchmarked against the unmodified kernel.

We used a Pentium 4 2.4 GHz with HyperThreading and 1024 MiB of RAM as a test machine.

We have evaluated three different test configurations:

1. The first configuration sets the **`prio`** parameter for threads on the **local** CPU. `prio` represents all parameters except `processor_control`. These parameters are more or less directly written to the schedulee's TCB and do not need a complex logic to be applied (at least when the destination thread resides on the same CPU as the scheduler). Plain TCB manipulation is fast; therefore, it does not require a large amount of compute time compared to the kernel entry/exit overhead. The test series basically indicates the potential of the ABI. Figure 4.2 shows the results.

2. The second test series also sets the **`prio`** parameter, but the destination threads reside on a **remote** CPU. This means that Pistachio needs to use the XCPU mechanism to modify the destination TCBs. The results are illustrated by figure 4.3.

3. The **`processor_control`** parameter, which is set in the third test series, performs a thread migration. This is a more complex task. Figure 4.4 compares the unmodified kernel with the new ABI and the optimised implementation that was explained above.

4.3 Analysis

As it is illustrated by figure 4.2, the performance of the pure system call mechanism (without expensive migration) lies between 40% (for small N) and 80% (for $N \geq 40$). However, this performance gain is relativised if the processing in the kernel is time-consuming – in our case, profiling has shown that especially pistachios XCPU mechanism is a bottleneck. The effect can be observed in figure 4.3 and in figure 4.4. The performance gain for a kernel with the new "Batch Scheduling ABI" alone is only around 70% for the thread migration scenario. For priority changes, the gain is even overcompensated. At the time of writing, we have no explanation for this behavior. However, the optimisation from section 4.1.2, which drastically saves XCPU messages, reveals that pistachios XCPU messaging system has a dramatical performance impact. With only very few messages per batch schedule, any operation is nearly as fast as a local priority change. The result is a performance increase of up to 95% for migration operations.

Figure 4.2: Priority Manipulation Costs (Local CPU)

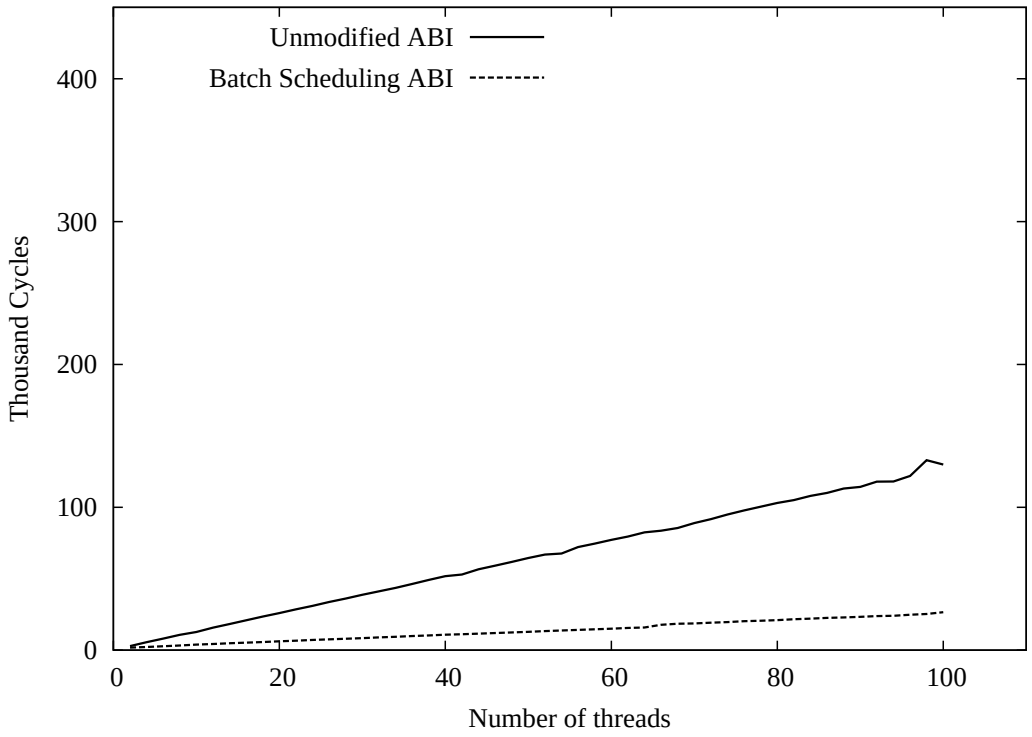


Figure 4.3: Priority Manipulation Costs (Remote CPU)

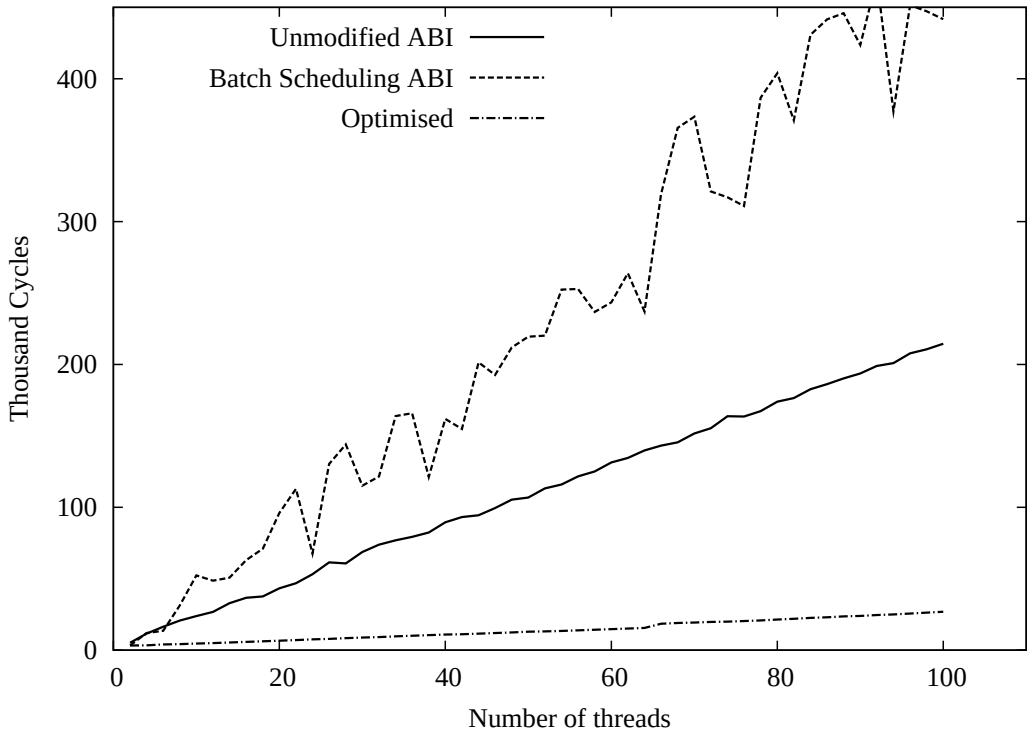
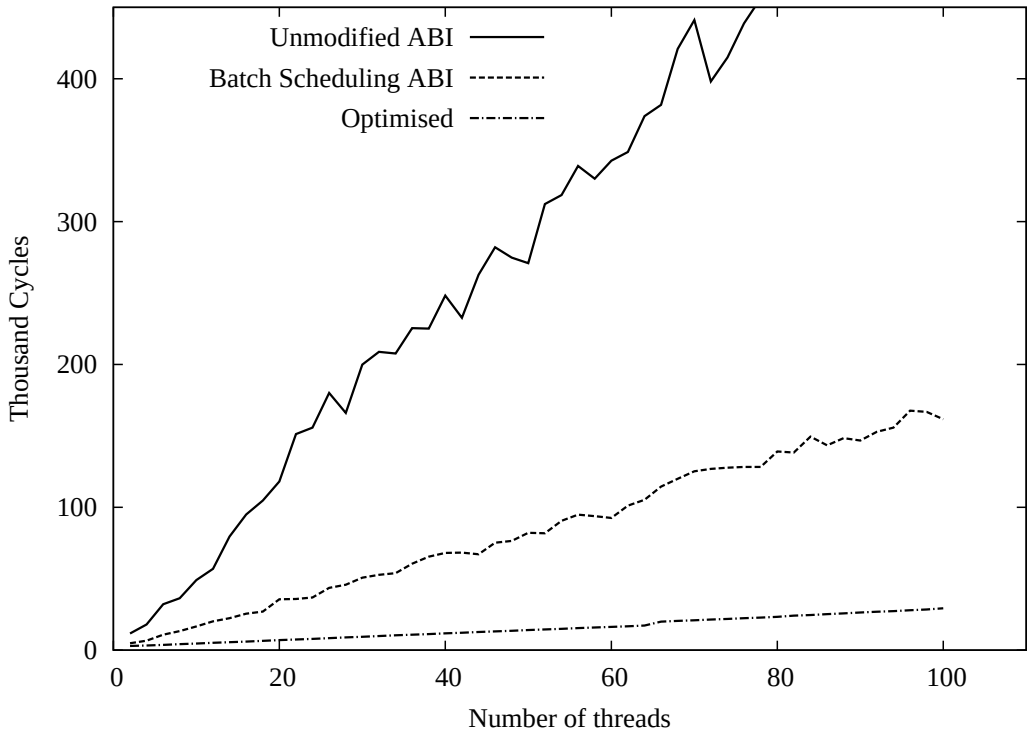


Figure 4.4: Processor Migration Costs



5 Conclusion

Up to now, management of large L4 thread sets – as they appear with applications like virtual machines – is connected with a large overhead because the L4 interface only allows the scheduling parameters to be modified for one thread at a time (which implies that a kernel mode switch must be performed for every thread).

In this thesis, we have developed and presented a modification to the `Schedule` system call and the derived parts of the L4 microkernel specification. The proposed function receives an array of L4 thread IDs (which represents a thread set) as destination parameter for the scheduling operation. The corresponding library implementation invokes the system call via a modified kernel binary interface.

This new interface made it possible to alter Pistachio's in-kernel mechanisms for thread manipulation in SMP systems in order to perform „batch scheduling" – i.e. to reduce communication and synchronisation overhead by processing a set of destination threads.

We have evaluated the proposed design for the interface and the mechanism using a microbenchmark which performs certain thread manipulations on a growing set of destination threads.

5.1 Lessons Learned

„Batch scheduling" – if done right – brings a huge performance boost for the management of large thread sets. Although the performance gain that is caused by the saved kernel mode switches is remarkable (40% - 80%), the full potential is only unchained when the extra information that is given to the kernel is used for further optimisations inside the kernel (up to 95% performance gain).

5.2 Future Work

5.2.1 Practical evaluation

Although the microbenchmark reveals the promising potential of "Batch Scheduling", the concept has not been tested in a practical environment yet. To investigate the effects, the concept should be employed in a real virtual machine environment. Load balancing of many virtual machines is an option.

5.2.2 Alternative interface design

The interface that was proposed in chapter 3 is just one possible generalisation of the `Schedule` system call. It is based on the assumption that the application requires the same scheduling

parameters for all destination threads. Given a finite number of parameter words that can be passed to the kernel – i.e. (virtual) registers – this is a tradeoff for speed (more threads can be manipulated per system call) against flexibility (the same scheduling parameters are applied to all threads in the set). An alternative approach is to simply multiply `Schedule`'s parameter tuples; it then becomes possible to freely manipulate a number of threads (which is smaller, though). This design decision is highly application specific and must be evaluated accordingly.

5.2.3 Virtual Register Bank

The Message Registers were originally intended to only serve the `Ipc` system call, but they are already "misused" by two other system calls besides the proposed `Schedule` system call. This is highly unintentional and can result in nasty bugs, as the programmer is unaware that a CPI call (i.e. `Unmap`) will corrupt a message which was previously loaded into the MRs for a pending IPC operation. We therefore propose a new, more general term: the **Virtual Register Bank**. The API documentation should also give information about which CPI functions alter these virtual registers.

5.2.4 XCPU Mailbox Overflow

Although the modification described in Section 4.1.2 relieved the XCPU bottleneck, there is still demand for optimisation. The problem of an overflowing XCPU mailbox has not yet been addressed in Pistachio – resulting in a kernel panic in many situations. We were only able to run the benchmarks because we implemented busy waiting (while processing the own mailbox to avoid deadlocks) for the case that the destination mailbox is full, and we argue that this circumstance has a considerable performance impact.

Although the problem was exposed by the application of batch scheduling, there are also situations in which the original Pistachio kernel can be crashed. An example is a large number of XCPU IPC - i.e. pagefault messages that can be caused by a large amount of threads being scheduled on a different processor than their pager.

A Convenience Programming Interface Extensions

#include <l4/schedule.h>

*Word** ***ScheduleNoDetails***

Word ***Schedule*** (*ThreadId *dest, Word *n, Word TimeControl, Word ProcessorControl, Word Prio, Word PreemptionControl, Word * result = ScheduleNoDetails, Word * old_TimeControl = ScheduleNoDetails*) [*ScheduleN*]

Generic Implementation - Refer to section 3.1.

Word ***SetPriority*** (*ThreadId *tid, Word *n, Word Prio, Word *res*) [*SetPriorityN*]

{ return ScheduleN(tid, n, -1, -1, Prio, -1, res) }

Word ***SetProcessorNo*** (*ThreadId *tid, Word *n, Word p, Word *res*) [*SetProcessorNoN*]

{ return ScheduleN(tid, n, -1, p, -1, -1, res) }

Word ***SetTimeslice*** (*ThreadId *tid, Word *n, Time ts, Time tq, Word *res*) [*SetTimesliceN*]

{ return ScheduleN(tid, n, $ts * 2^{16} + tq$, -1, -1, -1, res) }

Word ***SetPreemptionDelay*** (*ThreadId *tid, Word *n, Word sensitivePrio, Word maxDelay, Word * res*) [*SetPreemptionDelayN*]

{ return ScheduleN(tid, n, -1, -1, -1, $sensitivePrio * 2^{16} + maxDelay$, res) }

Literature

[1] Michael Hohmuth, Michael Peter, Hermann Härtig, and Jonathan S. Shapiro. Reducing tcb size by using untrusted components: small kernels versus virtual-machine monitors. In *Proceedings of the 11th ACM SIGOPS European workshop: beyond the PC*, page 22, Leuven, Belgium, 2004. ACM Press.

[2] Robert P. Goldberg. Survey of virtual machine research. *IEEE Computer Magazine*, 7(6):34–45, June 1974.

[3] The L4Ka Team. *L4 Kernel Reference Manual (Version X.2)*. Universität Karlsruhe, 2006. Available from http://l4ka.org/.

[4] Jochen Liedtke. On μ-kernel construction. In *Proceedings of the 15th Symposium on Operating System Principles*, pages 237–250, Copper Mountain, CO, December 3–6 1995.

[5] Jochen Liedtke. Improving IPC by kernel design. In *Proceedings of the 14th Symposium on Operating System Principles*, pages 175–188, Asheville, NC, December 5–8 1993.

[6] Alain Gefflaut, Trent Jaeger, Yoonho Park, Jochen Liedtke, Kevin Elphinstone, Volkmar Uhlig, Jonathon E. Tidswell, Luke Deller, and Lars Reuther. The SawMill multiserver approach. In *Proceedings of the 9th ACM SIGOPS European Workshop*, Kolding Denmark, September17–20 2000.

[7] Joshua LeVasseur, Volkmar Uhlig, Matthew Chapman, Peter Chubb, Ben Leslie, and Gernot Heiser. Pre-virtualization: Slashing the cost of virtualization. Technical Report 2005-30, Fakultät für Informatik, Universität Karlsruhe (TH), November 2005.

[8] Sebastian Biemueller and Uwe Dannowski. L4-based real virtual machines - an api proposal. In *Proceedings of the First International Workshop on MicroKernels for Embedded Systems*, pages 36–42, Sydney, Australia, January 16 2007.

[9] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, and Sebastian Schönberg. The performance of μ-kernel based systems. In *Proceedings of the 16th Symposium on Operating System Principles*, pages 66–77, Saint Malo, France, October 5–8 1997.

[10] Geoffrey Lee and Charles Gray. L4/darwin: evolving unix. 2006. Available from http://www.ertos.nicta.com.au/publications/.

[11] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and performance in the denali isolation kernel. *SIGOPS Oper. Syst. Rev.*, 36(SI):195–209, 2002.

[12] Joshua LeVasseur. Lecture slides: Microkernel construction, 2006. Available from http://i30www.ira.uka.de/teaching/.

[13] Jan Stoess. Towards effective user-controlled scheduling for microkernel-based systems. *Operating Systems Review*, 41(3), July 2007.