THE UNIVERSITY OF NEW SOUTH WALES
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

Single Kernel Stack L4

Matthew Warton

Thesis submitted as a requirement for the degree
Bachelor of Engineering (Computer Engineering)

Submitted: November 2, 2005

Supervisor: Gernot Heiser
Assesor: Sergio Ruocco

Abstract

Operating systems use kernel stacks to support the execution of threads. A typical multi threaded operating system uses one kernel stack per thread. This stack per thread model consumes a significant amount of kernel memory.

An alternative model is the use of a single kernel stack that is shared between all threads. This reduces the opersating systems memory consumption.

This thesis implements the single stack kernel architecture in the L4 microkernel to evaluate the performance and memory tradeoffs. It is shown that significant memory savings can be achieved without degrading the kernels performance. Preliminary results show improvement in the kernels performance due to the single stack architecure, however more experiments are required to verify this result.

Contents

Chapter 1

Introduction

Operating system kernels are required to manage complex concurrency issues arising from the need to manage the execution of multiple programs and multiple pieces of hardware simultaneously. Historically, two programming models have been developed to manage concurrency; the event based model and the thread based model.

Traditionally, most operating systems have been based on the thread model. The thread model has the advantage of presenting concurrent systems as sequential flows of control. This allows concurrent systems to be programmed in a similar manner to simple sequential programs. Additionally, the thread model allows external hardware events to be processed as they occur, whereas the event model requires the current event to complete processing prior to processing the hardware event. These significant advantages have led major operating systems such as Linux, Windows NT and Macintosh OS X to use the thread model [6].

The thread model requires each thread to have its own stack, that is a continuous area of memory where a thread can store its temporary state. Although the amount of memory consumed by the stack of each thread is moderate, systems with a large number of threads may consume a large proportion of available memory due to the use of an individual stack for each thread. In contrast, the event model requires the use of a single stack thereby using less memory.

Memory is a precious commodity in many computer systems, especially in embedded systems. Operating Systems developed for embedded systems must utilise memory efficiently. As a result these embedded operating systems often use the event model to save memory through the use of a single stack. If a single stack could be used in a thread model, it would lead to a significant reduction of the memory usage by the operating system. This would allow the thread model to be used in systems with limited memory.

This theses aims to evaluate the impact of the use of a single stack in a thread based model. It examines the impact of both the performance and memory usage of such an operating system by implementing a single stack version of the L4 Pistachio micro-kernel. The L4 Pistachio micro-kernel uses the thread based model and was selected for its small size and highly optimised nature. It therefore serves to be a good candidate for evaluating the impact on the performance of the single kernel stack architecture on operating systems, particularly for micro-kernels.

The most interesting areas of impact are memory usage and performance. It is expected that using a single stack kernel will greatly reduce the amount of memory required per thread, however it is also expected to decrease the performance of the

micro-kernel. That is, the extra work necessary to use a single stack rather than multiple stacks is expected to increase the time required to perform any given workload. This performance decrease is contrary to previous research [4] that was carried out on the Mach micro-kernel. The optimisations made possible by the use of a single stack in the Mach micro-kernel have already been implemented in the multi stack Pistachio kernel. Thus the performance gains achieved in this research are not expected to be duplicated. This thesis investigates the performance impact of a single stack kernel with the goal of minimising any penalties inherent in the approach to a negligible proportion of total system execution time.

Chapter 2

Background

The examination of the differences between the single stack kernel and the multi stack kernel require an understanding of concepts such as virtual memory and caches. Additionally, much of the implementation of the single stack kernel is necessarily tied to the existing implementation of the L4 Pistachio kernel. This section therefore provides the neccesary background information on relevant architectural and micro-kernel concepts.

2.1 Kernel Mode

Computer architectures designed to support modern operating systems are capable of operating in at least two modes: kernel mode and user mode, alternatively known as privileged and unprivileged mode respectively. In kernel mode the operating system kernel forms the majority of the code executed whereas in user mode most of the executed code belong to user programs.

Kernel and user mode allows the concept of a trusted computing base to be implemented. The trusted computing base in a computer system is the part of the system that implements the security policies of the system, and hence falls outside of all security policies within the system. Generally the trusted computing base is comprised of the operating system and some hardware drivers. In the event that the trusted computing base contains incorrect or malicious code, the security of the entire system is compromised. Kernel and user mode form the basic mechanism by which the trusted computing base implements its security policies. The processor is switched from user to kernel mode by exceptions, and

Code that is executed in kernel mode has complete access to the system and is part of the trusted computing base. Kernel mode is necessary to perform certain actions such as communicating with hardware and manipulating restrictions placed on the user mode. Traditional or monolithic operating systems execute almost all of their code in kernel mode.

Code executed in user mode is subject to restrictions with respect to the actions it is able to perform. These restrictions are set by the code executing in kernel mode. The restrictions are designed to prevent code executing in user mode from interfering with the execution of any other part of the system. These restrictions include virtual memory which limits the memory accessed by code executed in user mode. Due to being subject to security restrictions, code running in user mode does not automatically become part of the trusted computing base.

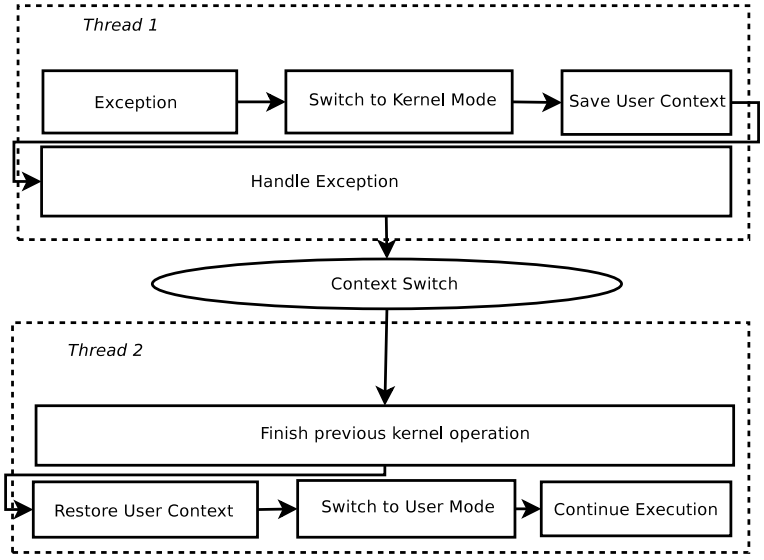

Figure 2.1: Exception Steps

An exception can interrupt the normal operation of the processor at any time. Possible causes of exceptions include hardware signals, calculation errors, and virtual memory page faults. An exception causes the processor to halt the processing of the current code and change to kernel mode. The processor then starts executing code at a special address, known as an exception vector. There is usually a different exception vector for each possible type of exception. The exception vectors can be set by the operating system, so that the operating system is notified of all exceptional circumstances in the system. The operating system is able to respond to exceptions as it sees fit.

Under most circumstances the user thread that was executing prior to the exception will need to be continued in the future. For this reason, the threads state must be saved in memory in such a way as the thread can be later restored to the same state. The code that performs the storing of the state is known as the kernel entry code. After the user's state is saved, the kernel entry code calls another function in the kernel to handle the actual exception.

The code that resumes the user thread once the kernel completes processing the exception is known as the kernel exit code. The kernel exit code is responsible for the restoration of the user thread to the same state it was prior to being paused. After restoring the thread's state the kernel exit code performs an exception return, switching the processor back in to user mode and jumping to the threads next instruction. The kernel entry and exit code are very tightly bound to the processor hardware, and are therefore similar in all operating systems for the same processor.

To perform input and output (IO), user program needs to communicate with the kernel. This communication takes place via a system call. A system call is an instruction that raises an exception therefore crossing the user-kernel boundary. The kernel can then examine the state of the user program and perform the requested action. The kernel can later resume the user program with the results of the action.

Changing between user and kernel mode takes time. This time includes the time taken by the processor to switch modes, and the time taken for the kernel entry and exit code to save and restore the user's state to and from memory. Because each mode

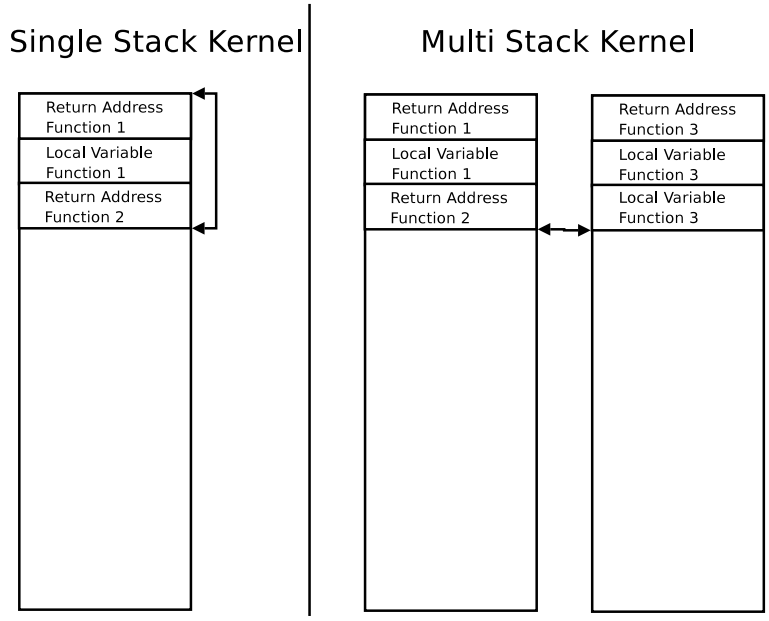

Figure 2.2: Kernel stacks in the single and multiple stack kernel

switch takes time without performing any work, system designers attempt to avoid system calls where possible. For instance the kernel may provide more data to the user than the user requests in order that the user's next request for data will not result in a system call. Optimisations such as these can amortise the performance cost of system calls.

2.2 Kernel Stacks

A stack is a fundamental data structure used in the execution of programs. The stack is used to store temporary data as a program is executed. This consists of data that is needed for the execution of a single function, known as local data, and the address used to return to execution once the function completes. The stack is represented by a register that points to the current bottom element of the stack. When a new function is called, the function subtracts the amount of space it needs on the stack from the stack pointer. The function is then free to use this stack space to support its execution. When the function returns it moves the stack pointer back to the place it occupied prior to the invocation of the function. This mechanism provides for the efficient allocation and deallocation of data on a per function basis.

Each thread in a system requires a stack in order to store the data associated with the sequential flow of its instructions. For parallel flows of instructions, one stack is needed for each thread. Context switching, or switching between threads requires the stack pointer to be switched to point to the current place in the new threads stack. Thus the new thread can resume from the point at which it was paused.

Errors can arise due to the incorrect use of the stack in a program's execution. The first type of error is stack overflow, which occurs when the program tries to use more stack space than the total size of the stack. Because the stack is of a finite size,

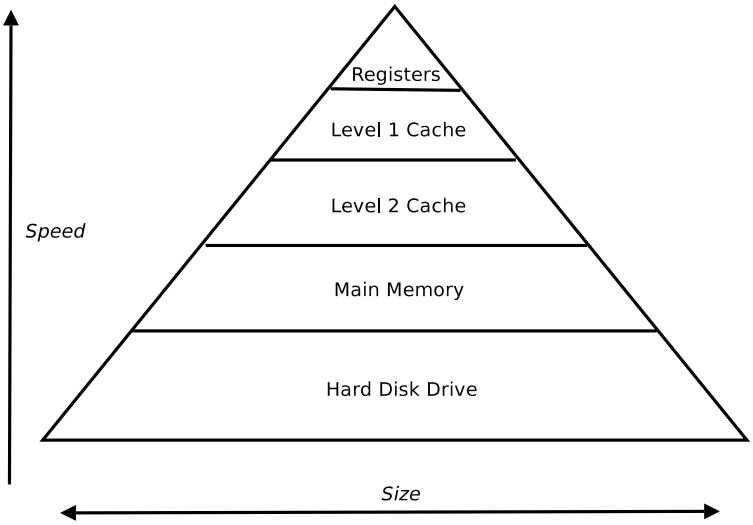

Figure 2.3: The memory heirarchy diagram

care must be taken not to allocate too much memory from the stack. This error is often detected in user threads by use of virtual memory techniques. If the error is not detected, important data may be overwritten as the stack pointer continues to descend into areas of memory that are used for other purposes. It is difficult to locate stack pointer errors in the kernel, as the kernel often does not use virtual memory techniques to detect such errors due to memory concerns.

Stack corruption errors is another source of possible error. These errors can occur when a program modifies it's stack pointer in a non-standard way, to point to some other area of memory. This may occur unintentionally, or intentionally with a malicious purpose. Again, important data may be overwritten due to stack pointer corruption.

When a user thread makes a system call to the kernel, it must not be able to bypass the security policies of the kernel. This means that the kernel must carefully validate every piece of data from the user thread in order to maintain security. The current stack pointer is one such piece of data. In the event that a user program has a corrupt stack pointer when a system call occurs, and the kernel uses the user's stack pointer, the kernel would then have an invalid stack. In order to avoid this scenario, the kernel needs to maintain a trusted stack and stack pointer for each thread and use this instead of the user's stack and stack pointer. The trusted stack is known as a kernel stack.

When referring to a single stack kernel it means precisely that it has only a single kernel stack. Similarly a multi stack kernel is a kernel that contains one kernel stack for each thread. Both kernels provide user stacks as required by the user program. A single stack kernel therefore reduces reduces the number of kernel stacks in a system.

2.3 Caches

Caches significantly increase the performance of a computer system. Programs written to take advantage of the cache can often achieve almost 10 times better performance than programs that fail to take advantage of the cache [16]. Therefore an understanding of cache architecture is highly important for the optimisation of any computer system.

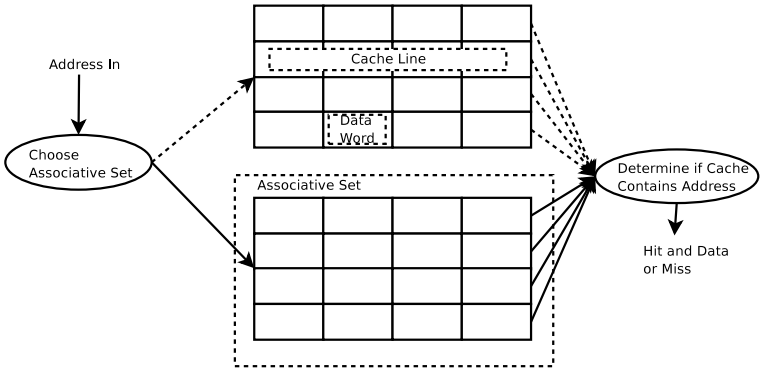

Figure 2.4: A 4 way set associative cache with 8 lines and a line size of 4 words

Caches form part of the memory hierarchy. The memory hierarchy exists due to the fact that an increase in the speed of memory is matched by and increase in its price. Since it is desirable to have computers with fast memory that are also inexpensive, a small amount of fast memory is used to hold copies of frequently used data from the larger, slower memories. The fastest memory in a computer system are the processor registers, all of which can be accessed in a single processor cycle. The next level of memory are the caches. There may be more than one level of caches, each being larger but slower than the previous level. The next level is RAM, also known as main memory. After this there may be several secondary storage devices such as magnetic disks or tape.

When a memory address is accessed, each level of the memory hierarchy is checked in turn to see if it contains a valid copy of the requisite data. If the level contains a copy of the data, it can supply the data to the higher levels of the hierarchy and processing can continue. If the level does not contain a copy of the requested memory, processing is stalled until the required data can be retrieved from the lower levels of the memory hierarchy. Retrieving the data from the lowest levels of the hierarchy can take millions of processor cycles. Therefore it is important to keep data that may be used in the near future as high in the memory hierarchy as possible.

If the requested item of data is not present in a cache, this is known as a cache miss, if it is present it is referred to as a cache hit. Retrieving the data from a cache only takes a few processor cycles as opposed to the retrieval of data from main memory which may take hundreds of cycles. In other words a cache miss is a very expensive operation. The ratio of cache hits to cache misses is called the cache hit ratio and is an important determinant of system performance. Modern computer systems typically have cache hit ratios of over 95% [16].

Cache design is based upon two principles: Spatial locality: If an item is referenced, items whose addresses are close by will tend to be referenced soon [16].

Temporal locality: If an item is referenced, it will tend to be referenced again soon [16].

To take advantage of spatial locality, caches fetch several adjacent pieces of data at a time. In order to take advantage of temporal locality, caches retain the most recently used data and discard old data.

A cache design is summarised by three parameters, these being line size, the number of lines, and associativity.

The line size is the number of bytes that make up a cache line. Upon a cache miss, an entire line is fetched, not just the byte being accessed by the processor. This design takes advantage of spatial locality, as well as the fact that accessing adjacent memory locations in main memory is faster if they are accessed at the same time. A larger cache line may decrease the number of cache misses if the program has strong spatial locality, however each cache miss takes longer as more data must be transferred. The cache line size is therefore a tradeoff between data transfer time and spatial locality.

The number of lines in the cache determines the total amount of data the cache can store. The more lines in the cache the greater advantage can be taken of spatial and temporal locality. Unfortunately cache memory is expensive and therefore limited in size. The number of lines in the cache is therefore a tradeoff between cost and performance.

The associativity of the cache is related to how data in the cache is addressed. In a direct mapped cache, every block in memory has a single cache line in which it may be stored. Because the cache is smaller than memory, multiple memory blocks map to the same cache line. If another of these memory blocks is used, the original will be evicted. This can lead to many conflict misses. Alternatively each block of memory can be made to map to a number of cache lines. This is known as n associativity. For example in a 2 way associative cache, each block can map to two cache lines, so two memory blocks that have the same mapping can be cached at the same time. In a fully associative cache, each block of memory can be mapped to any cache line. Caches are typically 1, 2, or 4 way associative as greater associativity increases the time a cache takes to respond to queries for data, thereby slowing processing. [16].

Cache misses can cause significant delays in processing. Cache misses can be classified into three categories, these being compulsory, capacity and conflict misses.

Compulsory cache misses occur the first time a piece of data is accessed. Compulsory misses are unavoidable as the only way in which the number of compulsory misses can be reduced is to use less memory.

Capacity cache misses occur because more data than the size of the cache has been accessed since the last time the initial data was used. That is the initial data has since been evicted from the cache. These misses can be reduced either by increasing the size of the cache or by reducing the amount of memory used by the software at any one time.

Conflict misses occur when multiple separate pieces of data map to the same location in the cache. Accessing one block of memory evicts another, creating a see-sawing effect that can slow down the system. In order reduce conflict misses either the associativity of the cache can be increased or the data can be moved in memory to map to different cache locations.

The difference between a capacity miss and a conflict miss is that a conflict miss can only occur in a non-fully associative cache. A conflict miss occurs when there is not enough associativity to keep the data in the cache, but if the associativity was increased the miss would not have occurred. A capacity miss is when the miss would have occurred even if the cache was fully associative.

The main implication of caches for software performance is that cache misses slow the software. All types of cache misses are related to the amount of memory used by software. Ideally the less memory used the faster software due to less cache misses. Each cache miss avoided saves another cache miss when the original data has to be loaded back in.

2.4 Virtual Memory

Virtual memory is one of the fundamental security mechanisms available in modern computing architectures. The purpose of virtual memory is to isolate user programs from physical memory. By providing a virtual address space, the user program is unable to access memory that it should not, whereas unrestricted access to physical memory allows the program to access all data in the system.

The operating system is responsible for mapping virtual addresses to physical addresses. This map is then used by the processor to determine the physical memory address to query. This mapping is performed on a block basis. Blocks of memory are known as pages if referring to virtual memory, or as frames if referring to physical memory. the current relationship between pages and frames is described by a data structure in memory known as the page table. A page table includes mappings for a complete virtual address space. Each virtual page in the address space can be either mapped to a particular physical frame or have no mapping.

The page table also specifies the operations that may be performed on each page. That is whether the page can be read from, written to, or code can be executed from this page by the user process. Many processors treat execute permission as read permission and therefore have only two types of pages, that is read only and writable pages. The ability to specify permissions for each page allows more flexible restrictions to be placed on user processes.

When a program attempts to access an address in memory, the system will look up the page corresponding to that address in the page table. If the page is mapped to a frame, and the attempted operation is allowed, the operation will be performed on the appropriate address in the physical frame. If there is no frame, or the operation is not permitted, the hardware will generate a page fault, and the operating system will be invoked. The operating system can then decide the appropriate action to take based on the conditions of the page fault.

Virtual memory is used for several purposes. It is used to detect errors in programs, that is if a program accesses a memory address that it should not, there is an error. Another use is to allow more than one user process to be executed. By using different page tables for each process, the processes are unable to interfere with each others memory. Finally, virtual memory can be used to make secondary storage such as hard disks become a lower level of the memory hierarchy. Pages that have not been used for some time can be copied to storage such as a hard disk. When the page is needed again, the program will page fault on the address. The operating system can then copy the page back into physical memory, establish the appropriate mapping, and resume execution of the program.

The advantages of virtual memory can incur a significant speed penalty due to the fact that the page table is stored in main memory. However, specialised hardware can be used to quickly translate virtual addresses into physical addresses. This hardware known as the translation look aside buffer, or TLB works as a cache of the page tables. The TLB caches the last few pages accessed, working on the principles of spatial and temporal locality. If a translation is found in the TLB, the virtual address is translated to the physical address immediately. If the page is not found, the page tables must be referenced to fill the TLB with the correct entry. The delay for this operation may range from a few cycles to several thousand cycles. Thus an important performance enhancement to any program is to access data on as few pages as possible.

Each TLB miss avoided will eliminate two TLB misses, one to load the new entry, and one to reload the old entry that was removed to store the new entry. With each TLB

miss costing multiple cycles any reduction in TLB misses may improve performance noticeably.

2.5 Interrupt Latency

Interrupts are exceptions caused by a signal from a hardware device indicating the occurrence of an event. Interrupts as the name suggests, interrupts the currently executing program and invokes the operating system to handle the event as for any other exception. Interrupts are a frequent occurrence in the normal operation of a computer system.

Interrupt latency is the time period between hardware generating an interrupt and the processor invoking the operating system to respond. This can be a problem with devices that have a high frequency of interrupts such as gigabit network cards, as another interrupt may be generated before the first is processed. In this case the two interrupts will be combined into one, leading to errors or inefficiencies in dealing with the hardware.

Interrupt latency is also important in real time systems. A real time system is a system in which time is an element of the correctness of the system. That is even if the correct result is calculated, it is of no consequence if it is not calculated by a certain time. Typical examples of real time systems are air traffic control systems and medical systems. Many embedded systems are also real time systems.

For a system to be use for real time problems, it must be able to provide guarantees on the worst case execution time for each of it's operations. These worst case times can then be used to calculate the worst case execution time of the program. This can then provide a guarantee of the correctness of the program. Interrupt latency is one of the guarantees that a real time system must provide. Many deadlines are set from the time a sensor device raises an interrupt therefore a long interrupt latency could cause the deadline to be missed.

Interrupt latency is primarily caused by software disabling interrupts. The operating system will disable interrupts if allowing the interrupt during an operation would place the system in an inconsistent state. User programs are not able to disable interrupts, only code executing in kernel mode is able to disable interrupts. It is important for systems to minimise the amount of time for which interrupts are disabled, if they contain hardware that generate a large number of interrupts, or are to be used to solve real time problems.

2.6 L4

L4 is a second generation micro-kernel Application Programming Interface. By second generation, it is meant that the L4 API [20] follows the classical design of a micro-kernel, but compliant kernels typically have much better performance than any of the first generation of micro-kernels. The L4 project was originally conceived by Jochen Liedtke, and is now developed by teams at the University of Karlsruhe, the University of New South Wales and the University of Dresden.

The L4 API [20] defines operations that an L4 compliant kernel must implement. The L4 API is written in a platform independent manner, and is supplemented by an Application Binary Interface for each specific computing platform. The current version of the L4 API is the L4 experimental version X2 API [20]. There are a number of

micro-kernels that implement the L4 API. The leading L4 micro-kernel in terms of performance and platform independence is the L4Ka::Pistachio kernel [19].

This section discusses the differences between L4 and other micro-kernels, and the key abstractions of the L4 API. It then discusses the internal implementation specific details of the Pistachio kernel that are relevant to this thesis.

2.6.1 Micro-kernels

The term *kernel* refers to the part of an operating system that executes in kernel mode. Traditional operating systems were written as *monolithic* operating systems, providing most operating services from kernel mode. This monolithic design typically leads to a large kernel as many services such as file systems are included in the kernel. The performance of monolithic systems is typically very good because the number of user-kernel mode changes and context switches is minimised by the inclusion of all operating services in the kernel.

In contrast, a micro-kernel attempts to minimise the size of the kernel. It achieves this by only including services that must be provided from kernel mode in the kernel. Other services such as device drivers, file systems and networking are provided by user processes. As a result of minimising the size of the kernel, micro-kernels benefit from a number of software engineering advantages. These include improved flexibility, extensibility, reliability and security [13].

2.6.2 Performance

The first generation of micro-kernels, including Amoeba [18], Chorus [17] and Mach [7], were notorious for their slow performance. Ultrix (a UNIX implementation) running as a user level server under Mach was shown to be up to 66% slower than Ultrix running alone [1]. Extensive investigation was conducted into the reasons behind the poor performance of micro-kernels.

It is immediately apparent that running operating system services at user level rather than inside the kernel causes additional user-kernel mode switches as well as context switches that are not needed in a monolithic kernel. However further investigation revealed that inter-process communication (IPC) was the principal reason for the poor performance of these systems [9, 13].

In fact micro-kernel system performance is essentially limited by IPC performance [11]. Hence reducing the cost of IPC is paramount to system performance. Research into the poor performance of Ultrix on Mach showed that 73% of the overhead was attributable to IPC related activities. Further it was demonstrated that there were 20% more cache misses when running Ultrix on top of Mach, due to competition between the user and kernel for cache space. IPC performance had to be improved.

The second generation of micro-kernels were designed around the goal of minimising IPC overhead. Micro-kernels such as L4 [20], Exokernel [5] and QNX [10] have to a large extent achieved this goal. For example, L4 has been shown to run Linux under the AIM multiuser workloads to within 5-8% of native Linux performance [9] whereas Linux under Mach suffered an average performance penalty of almost 50% [2].

2.6.3 L4 Abstractions

L4 provides very minimal services in the kernel to reduce size and improve performance. Specifically, L4 provides the abstractions of virtual memory address spaces

and threads within those address spaces. Additionally it provides a way for threads to communicate by sending messages to one another, known as interprocess communication (IPC). Other traditional operating system services such as file systems can be provided through the use of these abstractions by user level processes.

Threads

A *thread* is the basic abstraction of execution in L4. A thread executes a sequential stream of instructions, and can interact with the kernel through system calls. Every thread executes within an address space that defines the memory that the thread can access. Threads have other state associated with them, such as current status and a scheduling priority. Some of this state is accessed through system calls, and other items of this state reside in a data structure known as the *user thread control block* . The UTCB is an area of memory shared between the L4 kernel and the thread which contains state that is able to be modified by the user thread.

Threads are represented in user processes by thread identifiers. There are two types of thread identifiers, *global ids* and *local ids* . local thread ids are only valid within the threads own address space, whilst global thread ids are valid globally in the system. Local ids are not often used in systems under L4. Throughout this thesis the generic term thread id refers to a global thread identifier.

Address Spaces

An L4 *address space* is a mapping of virtual memory to physical memory. This mapping is built up recursively by the means of mapping operations. This is intentionally different to the virtual memory systems of most operating systems to enable virtual memory systems to be run as user processes.

A *mapping* operation maps memory from one address space to a different address in another. The memory can be either given to the new address space of shared after the mapping operation. There is a special address space, *sigma0* which is a one to one mapping of virtual to physical memory. By mapping memory from the sigma0 address space to other address spaces, address spaces with arbitrary layouts can be created. There is also an unmap operation, as memory is limited and may need to be reclaimed for another purpose.

L4 also includes special handling of page-faults to support this address space scheme. When a thread causes a page-fault in L4, the kernel interprets the page fault as a special IPC to the threads *pager* . The pager is another thread which is registered to receive the current threads page faults. The pager thread can reply with a mapping of memory to the thread that page-faulted, and the thread will be resumed.

Interprocess Communication

Interprocess Communication is the principle method of communication between threads in L4. IPC is a synchronous message based communication system. That is data is communicated as messages, and the messages are only delivered when both the source thread is ready to send, and the destination thread is ready to receive.

L4 IPC messages can contain up to 64 words of data. four types of data can be conveyed by L4 IPC. The receiver can specify the types of data they are willing to accept, to ensure security in the case of a malicious IPC partner.

Untyped data Untyped data is treated by the kernel as binary information. No special interpretation of this data is performed. Untyped data is the most common data transferred by IPC.

String items String items are typed data items that instruct the kernel to copy data from the source threads address space to the destination threads address space. Up to 4 kilobytes of data can be copied from buffers in the senders address space to buffers in the receivers address space per string item. In practice this form of IPC is rarely used.

Map items Map items are typed data items which instruct the kernel to map a section of virtual memory from the senders address space to the receivers address space. After the mapping operation the sender and receiver share access to the memory. Map items are most commonly used in response to page faults.

Grant items Grant items are typed data items that instruct the kernel to perform a grant operation between the sender and receiver. A grant operation is similar to the map operation, except the section of virtual memory is removed from the senders address space after the grant operation. Thus the virtual memory is not shared as in a map operation, but transferred between the sender and the receiver. The memory is only removed from the senders address space, not the address space that originally provided the memory to the senders address space. Grant items are not commonly used, but have applications in user level device drivers.

2.6.4 L4 internals

The single stack kernel needs to make modifications to the thread control blocks and kernel stacks in the pistachio kernel. In addition, care needs to be taken not to slow the performance of IPC in the kernel. This section discusses the existing implementations of these structures in the pistachio kernel.

Thread Control Blocks and Kernel Stacks

A *thread control block* is a data structure which contains information about a particular thread. In L4, there are two types of TCBs, the user TCB and the kernel TCB. The user TCB contains the thread state that the user is permitted to modify, while the kernel TCB contains the thread state that is not able to be modified by the user. This thesis is principally concerned with the kernel TCB, when TCB is used unqualified it refers to the kernel TCB.

In pistachio, the kernel stack and TCB are located in the same region of memory. The TCB starts at the lowest address in this region of memory, whilst the kernel stack starts at the top of this region of memory. By combining the two data structures in the one region of memory, the two structures can be located on the same virtual memory page. Thus a kernel operation involving a thread will only cause one TLB miss per thread.

TCB/kernel stacks are located in an array in virtual memory. This array is very large, and uses a good proportion of the virtual address space. However, as the array is in virtual memory, memory only needs to be used for those threads that actually exist, so actual memory consumption is reasonable. TCBs are stored in this manner because it allows easy indexing based on thread ids. The thread id contains an index into the virtual array of TCBs, and a thread id can therefore be turned into a TCB

address in very few instructions and with no references to memory. This is important in performance critical areas of the code, such as IPC.

TCBs and kernel stacks are the main data structure that is modified to convert pistachio into a single kernel stack.

IPC Fast-path

IPC limits the performance of micro-kernels such as the pistachio micro-kernel. Pistachio therefore treats IPC as the central focus of the kernel design. The principle method the pistachio kernel uses to speed up IPC operations is to treat the most common case, untyped only IPC, in a special section of highly optimised code known as the *IPC fast-path* . IPC operations that cannot be handled by the fast-path are handled by the fall back IPC slow-path which is able to perform all types of IPC, albeit less efficiently than the IPC fast-path.

To provide maximum performance the IPC fast-path is written in assembly code. Assembly code saves much of the overhead of compiled code, and allows register handling and instruction scheduling optimisation not possible in compiled code. It is important that the single stack kernel modifications are compatible with some version of the IPC fast-path, as otherwise the single stack kernel will not be able to match the performance of the multi stack kernel.

Chapter 3

Single Stack Kernel

The single stack kernel architecture is fundamentally different from the multi stack kernel architecture. This section explains the architectures in detail, and discusses the important differences and tradeoffs between the two architectures.

3.1 Stack Architecture

The single stack kernel uses a single kernel stack that is owned by the currently executing thread. When the currently executing thread changes in a context switch, ownership of the stack is transferred to the new thread. The new thread has no use for the information contained in the stack, so it treats the stack as a new blank stack. Consequently, in a single stack kernel a thread effectively discards it's stack, as it can never again access the information that was contained in the stack. In contrast, each multi stack kernel thread owns a separate kernel stack. Upon a context switch in a multi stack kernel stack ownership is not affected. The thread being switched to simply loads its own stack from before. Hence a thread in a multi stack kernel can retrieve information contained in it's stack even after a context switch has been performed.

Due to the fact that a thread in the multi stack kernel retains it's stack upon a context switch, it is possible to provide blocking context switches. A blocking context switch models a context switch as a function call. The function switches to the specified thread, and waits to be switched back to. When the thread is eventually switched back to the function will simply return, and all state and return address information will still be available. Blocking context switches provide a convenient programming interface for kernel programmers.

Single stack kernel threads lose any data stored in their stack upon a context switch. Therefore any local data and return addresses on the stack are lost during a context switch. This makes it impossible to provide blocking context switches. Context switches in a single stack kernel may still be modelled as a function call, however the function will never return. This behaviour is not usually expected from a function and can be a source of confusion when writing a single stack kernel. Also due to the loss of the state stored in the stack across a context switch, function return addresses and local state that will be needed after the context switch must be stored in an alternate location such as the TCB explicitly. This explicit state management is the bulk of the work that needs to be performed in converting a multi stack kernel to a single stack kernel.

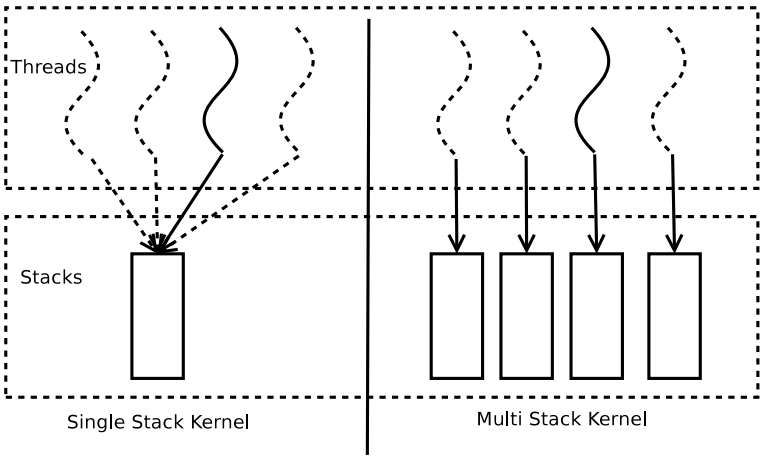

Figure 3.1: Single and Multi Stack Kernels

3.2 Alternative Models

There are a few major design choices that can affect the kernel's performance. These include whether to place the stacks in virtual or physical memory, and the possible use of an architecture where the number of stacks varies with need.

3.2.1 Virtual/Physical memory stacks

Whether to place kernel stacks in physical or virtual memory is an important design consideration. We define physical memory in the same way as Abi Nourai [15]. That is that architectures may use a super-page to simulate physical addressing. Because a single super-page is used there can be at most one TLB miss caused by the super-page, which will probably be incurred by other parts of the kernel in any normal operation.

On architectures that support physical addressing, placing the stack in physical memory relieves pressure on the TLB. If the stack is in physical memory, there can never be a TLB miss. Hence the kernel will not suffer a TLB miss penalty, and the user will not suffer from a TLB miss penalty to reload the evicted entry. Depending on the cost of TLB misses, placing the stack in physical memory and hence reducing the number of TLB misses can provide a significant performance advantage; particularly in calculations where the TLB is heavily used by the user program between kernel invocations.

Stacks in virtual memory may cause additional TLB misses, but they have the advantage of being able to provide a guard page. A guard page is a page of memory immediately below the stack that has no mapping to physical memory. Guard pages provide the ability to detect stack overflow errors. The first access from a stack overflow error will be on the guard page, because the guard page has no mapping a page fault will be raised. This page fault informs the kernel of it's own internal error, and appropriate action can be taken.

The L4 Pistachio multi stack kernel uses the same virtual memory page to hold the TCB and the kernel stack of a thread. In this way, each thread used will never cause more than one TLB miss. In the single stack kernel however, each thread control block used may still cause one TLB miss. In addition the use of the stack may cause an

additional miss if the stack is in virtual memory.

It was decided to use physical memory stacks in the implementation, primarily because the pistachio kernel has never used stack guard pages. The potential performance improvement is more important than the error detection for the current work.

3.2.2 Variable Stack Architecture

The two architectural models under consideration so far are the multi stack kernel and the single stack kernel. The multi stack kernel has one kernel stack per kernel thread, whilst the single kernel stack has a single kernel stack per processor. There is a third alternative which is very similar to the single stack kernel. This architecture that has dynamically allocates and deallocates stacks as needed, and hence has as many stacks as neccesary at any given time. Due to the varying number of stacks this architecture is referred to as the variable stack architecture.

Additional stacks may be neccesary for various reasons, such as handling in-kernel page-faults. In L4 an in-kernel page-fault may result from a page-fault in a user area during a string IPC operation. If a page-fault occurs, the kernel must send a page-fault message to the users pager. However, once the pager thread is invoked, the current threads state will be lost because there is only a single stack. With a variable stack architecture the page-faulting thread can retain it's stack, and a new stack can be allocated for other threads to use until the page-fault is resolved. When the page-faulting thread is switched back to the extra stack will be deallocated and the current thread will resume using its old stack, thus resuming exactly where it left off. This architecture is more flexible than the single stack kernel, but uses less memory than the multi stack kernel, as long as only infrequent operations require the use of an extra stack.

The variable stack model requires the kernel stack to be loaded through indirection. That is, because there is not a single stack, the address of the stack cannot be hard coded into the kernel. The current stack must be loaded from a pointer of some sort, implying an additional load operation per kernel entry that is not required in the single kernel stack. However, to expand the single kernel stack to a multiprocessor system, the stack must be loaded for each processor. This also implies a load operation per kernel entry, so the variable stack model has no extra cost over the single stack model in a multiprocessor system.

The variable stack model is very similar to the single stack model. If there is no special situation requiring an additional stack, the kernel operates with a single stack. Therefore it gains the memory benefits of a single kernel stack whilst maintaining more flexibility. For this reason the variable stack kernel architecture has been implemented in this thesis. However there are no situations in the implementation that use more than one stack, so the kernel is a single stack kernel with the ability to be easily expanded with extra stacks for particular operations as neccesary.

Since there are no operations requiring additional stacks, the code enabling variable stacks is easily disabled. For the purposes of performance testing there are three versions of the kernel, a multi stack kernel, a variable stack kernel and a single stack kernel. The single stack kernel is simply the variable stack kernel with the extra code in the context switch and fast-paths removed. That is the single stack kernel still addresses the kernel stack through indirection, so it is not as fast as is possible for a single stack kernel on a single processor system.

3.3 Trade offs

Many factors affect differentiate the performance of the single and multi stack kernel. Each factor treated separately below, as the relevance of these factors to performance differs with the underlying computer architecture.

3.3.1 Memory Usage

Since a stack is associated with a thread, traditionally there is one kernel stack per thread in the system. Since each kernel stack in Pistachio is more than 1k of memory, this can represent a significant toll on the memory of the system. In a system with hundreds of threads, Megabytes of memory can be used in providing the kernel stacks for these threads. To make matters worse, kernel stack memory cannot usually be paged to disk for security and performance reasons. There has been research into swapping this data to disk [8], but it is generally not implemented in L4 kernels, certainly not in L4 Pistachio.

A single stack kernel only uses memory for a single kernel stack. As such, a threads local data is discarded upon each context switch. This implies that each threads control block must be expanded to hold additional state. This additional state however does not use as much as the per thread kernel stack, and thus a great deal less memory is used per thread. This is why the single stack kernel uses less memory per thread than the multi stack kernel.

Thread control blocks are located in virtual memory by treating the thread id an index into a virtual array. Therefore to actually save memory, the thread id's of threads in the system must be essentially consecutive. If thread ID's are allocated in a sparse manner, such that no two thread control blocks are located on the same page, each TCB will have a full page allocated for it and thus use the same amount of memory as the multi-stack kernel. This problem can be solved in a kernel that addresses TCB's indirectly, such as the physically addressed L4 kernel [15]. If thread control blocks are addressed indirectly, the kernel can allocate them such that they are always consecutive, and hence do not waste space.

3.3.2 Kernel Entry and Exit

There are several differences in kernel entry and exit between a single and multi stack kernel.

The first difference is that the single stack kernel stores the user context in the TCB, whilst the multi stack kernel stores the user context on the stack. This difference only has minor performance implications however. The only difference is that the single kernel stack needs to derive the stack pointer from the TCB pointer or vice versa on every kernel entry and exit, whilst the multi-stack kernel can use the stack pointer exclusively. This means that there is an additional load operation for each single stack kernel entry and exit that is not present in the multi stack kernel.

The second difference is the code for handling an exception that occurs while the kernel is in kernel mode. In the multi stack kernel there is no special condition to be handled, the stack pointer is just decremented as for a normal user to kernel mode switch. However in the single stack kernel, there is a need for different handling of exceptions from user mode or kernel mode. If an exception is taken in kernel mode, the state cannot be stored in the TCB, as it will overwrite the user state that is stored there. The kernel state must be stored on the bottom of the stack. Thus the mode

of the exception must be determined so that the state can be stored in the appropriate location. This determination takes an extra few instructions that slow down the kernel entry. On kernel exit, the mode that is being returned to must be determined so that the appropriate state can be loaded. Again, this takes a few instructions that are not neccesary in the multi stack kernel.

The last difference between the kernel entry code for the single and multi stack kernel is only apparent in system calls. In a multi stack kernel, system call kernel entry is optimised by only saving part of the user context. The programming interface for most processors defines that certain registers are to be saved by the called function in the event that the function needs to use these registers. The C compiler performs this register saving automatically. A kernel which maintains a kernel stack per thread can therefore allow the compiler to save these registers implicitly. The registers will be saved on the stack, and later automatically restored by the compiler at the appropriate time. However a kernel which does not retain each threads stack must store these registers to the TCB, incurring additional overhead because there is no way to stop the compiler from also saving the contents of these registers. Thus the single stack kernel must store and load additional registers that the multi stack kernel does not. This slows down system calls.

In summary, a multi stack kernel will generally have faster system entry and exit costs than a single stack kernel. This is because of various small optimisations possible in a multi-stack kernel that are not possible in a single stack kernel. The difference is particularly pronounced in a system call kernel entry and exit.

3.3.3 Context Switch

There are four possible cases to consider due to stack differences during a context switch. The multi stack kernel need deal only with the stack retention case, whilst the single stack kernel need deal only with the stack hand over case. The variable stack kernel must deal with all four cases, indeed it is this ability which defines it as a variable stack kernel. Each of the cases is outlined below, along with a description of how the context switch is performed.

Stack Retention

When both threads involved in a context switch have their own stack, the current stack needs to be switched. This is always the case in the multi stack kernel, and occasionally the case in a variable stack kernel. To perform the context switch some context information from the source thread is saved to the source stack, the stacks pointer is loaded from the destination stack, and context information is loaded from the other stack. Finally the return address on the newly loaded stack is used to resume the destination threads execution.

Stack Hand over

The only case in the single stack kernel and the most common case in the variable stack kernel is stack hand over. In this case the source thread no longer requires it's stack, and the destination thread does not have a stack. The stack pointer is reset to the top of the stack to give the new thread the illusion of a new empty stack, and the destination threads processing is resumed by using the stored continuation. Stack hand over is a

very fast method of thread switching because there is little work to be done to support the context switch.

Stack Allocation and Deallocation

The last two cases only occur in the variable stack kernel. These cases are the slowest of the four cases because they involve the allocation and deallocation of memory. They are performed as an appropriate combination of the other two cases, along with the requisite memory allocation operation. These two cases are the cases that truly give the variable stack kernel its flexibility.

In summary, the single stack kernel is expected to have the fastest context switch time due to the smaller amount of work it is required to do. However, the difference is not a large proportion of time, and context switches are infrequent enough in the system that this small difference should not affect overall system performance. Additionally, context switch costs are dominated by hardware costs on most architectures, so it is not known whether this performance difference will actually be measurable.

3.3.4 IPC Fast-path

The IPC fast-path is acknowledged as the most performance critical code in the L4 micro-kernel [11]. Thus the impact of the single kernel stack on the fast-path is of central importance to the overall performance of the single stack kernel.

The basic difference between the single kernel stack kernel and the multi kernel stack kernel is in the way they handle the stack during both kernel entry/exit and the context switch in the IPC. The multi stack kernel must deal with two stacks, however the single stack kernel must deal with extra state involved in the single stack architecture.

During kernel entry and exit the differences are much the same as for normal kernel entry and exit. However, an IPC fast-path call is always invoked by the user, and does not need to save user context as a normal system call does. In addition, it does not need to save all of the user context because these registers in the processor are used to hold part of the message to be transferred. Therefore the principle difference in kernel entry/exit occurs because the single stack kernel needs to load the current stack from a pointer in the TCB. This involves a single extra load instruction. However, the single stack kernel may additionally be able to reduce the number of instructions slightly because it is easier to calculate both TCB addresses. On the ARM architecture, this saves 2 instructions per IPC fast-path.

During the context switch in the IPC fast-path, The single stack kernel has slightly more work to do. This is because it needs to update the stack pointers in the two TCBs and update the TCB pointer in the stack. These extra stores can slow down the fast-path, but should usually be performed with minimal latency by the processors write buffer. The processors write buffer allows a program to continue to be executed while memory stores have not yet completed. On most architectures these stores will be placed into the write buffer and therefore have a negligible impact on performance.

A kernel with a variable number of stacks needs an additional check in the fast-path. This check is to ensure that the destination thread does not already have an associated stack. If the destination thread does have an associated stack, the full context switch code must be used, not the abbreviated version in the fast path. Therefore if the destination thread does have an associated stack, the IPC is delivered through the slow path. This check involves one load and a few extra instructions. It may slow the IPC

fast-path a few percent of the total time taken. This check is not necessary in the single stack implementation, as no other thread will have an associated stack.

The main impact on the fast-path performance will be the additional load during kernel entry, and any cache effects. Other than these, the fast-path of the single and multiple kernel stack should perform very similarly.

3.3.5 Cache Usage

During a kernel operation that involves more than one thread, the single kernel stack will use less of the cache than the multi stack kernel. This is because the multi stack kernel needs to access memory in two stacks, whilst the single stack kernel reuses the same stack. Re using the same stack the cache entries for the stack memory are still valid, thus causing less cache misses due to the kernel stacks.

In addition, the multi stack kernel is more likely to cause conflict cache misses, as the different stacks will more than likely map to the same cache lines. This empties two of the associative places in the cache, whereas a single stack kernel will only empty one of the lines. This implies less conflict cache misses in the single stack kernel.

Using a single kernel stack enables the kernel to use less of the cache. This leads to better performance through fewer cache misses. In fact, each cache line not used by the kernel causes two less cache misses - one by the kernel, and the other by the user reloading the data that the kernel evicted. This improved cache locality should lead to a performance enhancement in the single stack kernel over the multi stack kernel.

3.3.6 TLB Usage

Depending on the computer architecture, TLB misses due to virtual memory accesses can be expensive. Thus consideration of the number of TLB misses due to kernel architecture is important. In the following discussion we assume that upon kernel entry the TLB does not contain any kernel entries.

The Pistachio multi stack kernel uses a virtual linear array in memory to represent it's threads. That is each thread id is used as an index into an array in virtual memory. Each element of this array consists of a threads control block and it's kernel stack. Thus accessing a threads control block or kernel stack will cause a TLB miss. In a typical kernel operation involving two threads such as an IPC there will be two TLB misses due to kernel stack/TCB activity.

In the single stack variant of the L4 kernel, a virtual linear array is again used to represent TCBs, however the stack is separate to the TCBs. Thus in any kernel operation each TCB accessed will cause one TLB miss. Additionally if the stack is in virtual memory it will also cause a TLB miss. However as the stack is in physical memory it will not cause a TLB miss.

The single stack TCBs are smaller than the multi stack combined TCB/kernel stacks, so more TCBs fit into one page of virtual memory. Thus if a kernel operation involves two or more TCBs that are stored on the same page in virtual memory, there will only be one TLB miss, as only one page is accessed. As an example, on the ARM architecture there are two multi stack threads per page of virtual memory, whereas there are 8 single stack threads per page of virtual memory. Thus there is a greater chance of multiple threads being on the same page in the single stack kernel. This chance is strongly influenced by the manner in which the system running on top of L4 allocates thread ids.

In summary, a single stack kernel which has a stack in physical memory will cause no more faults than the multi stack kernel in the worst case. In addition the single stack kernel has a greater chance of multiple threads being on the same page, and as such may incur less TLB misses than the multi stack kernel. Each TLB miss saved in the kernel will prevent another TLB miss in the user, so the single stack kernel may receive a small performance boost from causing less TLB misses, depending on the allocation of thread IDs.

A kernel which uses indirection to address TCBs physically such as [15] rather than as a virtual linear array does not suffer any TLB misses. This design, which is orthogonal to the design choice of single or multiple stack kernel, will eliminate the relevance of TLB to the different kernels comparative performance.

3.3.7 Interrupt Latency

A single stack kernel has longer interrupt latency than a multi stack kernel. This is because the single stack kernel must disable interrupts for all processing in the kernel. The multi stack kernel can leave interrupts enabled for most of the time it spends in the kernel.

When an exception such as an interrupt is taken in kernel mode, the previous kernel state is stored on the stack for both the single and multi stack kernels. An interrupt entails a context switch to another thread to handle the interrupt. Since a multi stack kernel retains its stack in the event of a context switch, it can later be resumed at the point where it was interrupted. However, a single stack kernel discards its stack in the event of a context switch, so the thread could not later be resumed. Thus a single stack kernel must disable interrupts whilst it is executing in kernel mode, as an interrupt would cause a context switch that would destroy part of the systems state.

The worst case interrupt latency for the multi stack kernel is thus the time that it takes to enter kernel mode, save the users/kernels state, and re-enable interrupts. However, the single stack kernel cannot re-enable interrupts, so it's worst case interrupt latency is the time taken to execute the longest kernel operation. Thus the single kernel stack kernel has significantly longer latency than the multi stack kernel.

It may be possible to enable interrupts in kernel mode in a variable stack kernel. This would involve allocation of a new stack in the event of a context switch where the stack contains information that cannot be discarded. This would potentially reduce interrupt latency to near the levels of a multi stack kernel, while retaining most of the memory benefits of the single stack kernel. This idea has not been extensively investigated in this thesis.

In the Pistachio kernel for ARM, interrupts are disabled while in kernel mode even in the multi kernel stack. This is because it simplifies many areas of the code, and to the present interrupt latency has not been an important measure of importance. Thus even though in theory the single stack kernel should have longer interrupt latency than the multi stack kernel, the difference is not expected to be pronounced in this particular implementation.

3.4 Related Work

This is not the first work investigating the single kernel stack concept. In fact there have been two previous implementations which are similar in concept to the single stack kernel.

3.5 Mach

Richard P. Draves converted Mach 3 to use what I refer to as a variable stack architecture in his PhD thesis in 1994 [3]. Draves was able to achieve significant memory savings, as well as significant performance improvements in the Mach kernel. However it was not clear which of the performance improvements were due to the single stack architecture, and which of the performance improvements were due to his optimisation of the original kernel code.

Mach is generally acknowledged to be a slow kernel [1], so it is interesting to investigate the effects of the single stack architecture on the faster L4 micro-kernel. Finally, L4 has a much smaller cache footprint than Mach, which the single kernel stack is expected to further reduce. It will be interesting to see if this further reduction has a similar impact to the cache footprint reduction which motivated the design of L4 [13].

3.6 Fluke

Ford et al. [6] implemented the Fluke kernel API to make every kernel operation *atomic* . They define an atomic operation as an operation that is fully interruptible and restartable. This kernel API allowed them to implement the kernel as both event based and thread based with changes to only a small section of the code. They then proceeded to compare the performance of the different kernel types. This work is similar to the single kernel stack, but not the same. They achieved the different models (which use a different number of kernel stacks) by removing all blocking from within the kernel. Thus this work differs from the single stack kernel which retains blocking within the kernel, but achieves the reduction in memory use proceeding from using a single stack. The approach used in Fluke could not be used in L4, as the API is previously specified and cannot be arbitrarily modified to include only atomic operations.

Chapter 4

Implementation Issues

This section discusses the details of the implementation. The architectural change from a multiple kernel stack to a single kernel stack involves changes to many areas of the kernel. The major abstractions and methods used are outlined in this chapter.

4.1 Abstractions

Several abstractions were developed to ease development of the single stack kernel. These include continuations, finish functions and continuation accepting functions. The original single stack implementation also used the continuation stack abstraction, but this was not used in the final single stack implementation.

4.1.1 Continuations

A *continuation* is an abstraction of where execution should be continued when a certain condition is met. The most obvious condition is that the thread is resumed in a context switch, in which case a continuation will be used to restart execution of the thread. However continuations are not limited to being used in thread switches, and are in fact most often used in recreating chain of functions that have been called before a context switch.

A continuation is an address containing the first instruction to execute when this continuation is activated. When a continuation is activated, the stack pointer is reset to the top of the stack, and no further information is supplied to the code that will be executed. However, because the code is able to retrieve a pointer to the current TCB, it is able to determine all of the information neccesary for its execution. Hence a continuation is a function pointer to a function that accepts no arguments and is never expected to return.

The macro `ACTIVATE_CONTINUATION` allows continuations to be invoked. This macro is very efficient, on ARM it consists of only 3 instructions and does not reference memory.

4.1.2 Finish Functions

A *finish function* is an abstraction for a function that will be called after a context switch to finish the work of the current function. These functions are often named by

prepending `finish_` to the name of the function they are intended to complete. The functions normally pointed to by continuations are finish functions.

A finish function has a regular structure. All finish functions take no arguments and have no return value. The typical first action of a finish function is to load a pointer to the current TCB and reload any state needed to finish the functions work. After the functions work is complete, the finish function will invoke another continuation that was stored for this purpose by the function that registered the finish function. Finish functions are a design pattern that occurs often in the single stack kernel.

4.1.3 Continuation Stack

The continuation stack is a stack data structure (not a threads execution stack) whose data elements are continuations. I used a continuation stack in my original implementation to store the functions that would need to be executed upon the threads return from a context switch. The last in first out nature of the continuation stack is similar to the way return addresses are stored on the threads execution stack. This made it convenient to handle nested function calls that could lead to a context switch.

Unfortunately the continuation stack obscured the flow of control in the kernel. In many places in the kernel it is difficult to determine when a value should be pushed onto or popped from the continuation stack. In addition the continuation stack was not a very efficient data structure. Thus the continuation stack was dropped from the single stack kernel in the final implementation.

4.1.4 Continuation accepting functions

To replace the continuation stack, functions that accept continuations were used. These functions have a special type signature that must be used for all continuation accepting functions. A continuation accepting function may not return a value, and the last argument to the function must be a continuation. These functions are defined to return by calling the continuation given, or performing a normal return if a null continuation argument is supplied.

Continuation accepting functions allow the compiler to check that continuations are correctly handled between functions. If a function is called without the extra argument, the compiler will report an error. This ensures that the programmer cannot forget to modify functions that depend upon the function that is being modified. This was the chief source of errors with the continuation stack model. Using continuation accepting functions the programmer only need worry about the handling of continuations within each function he writes. This eases the programming of the single stack kernel.

Continuation accepting functions also enable incremental testing. Functions in the kernel can be modified one at a time to use continuations. After each function modification, the kernel is in an executable state. Therefore a test can be run on the function modified, and any errors can be easily pinpointed in the code. Incremental development and testing of the single stack kernel makes bugs in the kernel much easier to find.

It should be noted that the compiler checking and incremental testing advantages are only useful when converting existing multi stack kernel code to single stack kernel code. When writing entirely new code in the single stack kernel these advantages are not important. However since the actively maintained version of Pistachio is a multi stack kernel, most new code to be incorporated in the kernel will need to be converted to use a single kernel stack and continuations.

4.2 Control Modification

The bulk of the work in converting the multi stack kernel into a single stack kernel was modifying the functions in the kernel to use continuations rather than blocking context switches. The functions that needed to be modified were easily identified through compiler errors, because each function was converted into a continuation accepting function in turn. Each function that needed to be modified was modified and tested in turn.

The biggest difference between the multiple stack kernel and the single stack kernel is that state is not able to be stored on the stack. To this end all state required by a function across a context switch needs to be stored in the TCB. There is usually not more than a few words of state per function that needs to be preserved across context switches.

The modification of the control flow for a function involves creating a new function to finish the work after the context switch, storing any needed state in the TCB, and restoring this state in the finish function. In addition it is neccesary to change the function type to add a continuation argument and change the return type to void. The code in the finish function often needs to be modified in structure due to context switches embedded in `if` statements and `while` loops. This modification can take considerable time per function.

The above modification procedure applies to 90% of functions, the other functions must be treated on an individual basis. Several functions that have complicated control flow patterns were broken into several smaller functions. It is neccesary to break them into multiple functions around each context switch, and perform state saving and restoration about this split. Unfortunately the control often flows in complicated ways which make this deconstruction hard to perform. The IPC slow-path was the most complicated function to convert to use a single stack due to the large number of conditions under which a context switch may be taken, as well as the need to maintain good performance. One technique used in the IPC slow-path is to restart the function in certain conditions rather than to splitting the function around a context switch. This comes at a performance penalty, but the reduction in complexity is well worth it for the infrequently used IPC redirection operation.

System calls also posed a problem for conversion to a single stack kernel. Certain system calls need to take a continuation argument because they may cause a context switch, yet their arguments cannot be modified due to the way that the system calls are invoked from assembly code. This problem was solved by placing the continuation in the return address register and using the appropriate gcc macro to retrieve this value. This has the advantage of being automatically architecture independent in the system call code.

4.3 Porting Architectures

As the Pistachio kernel runs on multiple architectures, it is important that this work be as easily portable as the kernel itself. To this end the changes were kept architecture independent where possible. Any architecture dependent code was places in the pistachio architecture dependant directories of the kernel, to avoid introducing conditional compilation into the architecture independent sections of the kernel. The majority of the control flow changes are architecture independent, whilst the kernel entry and exit, context switch and fast-path code are architecture dependant. Thus there is only a

small amount of work to port the code to a new architecture. I believe the work could be ported to each architecture supported by pistachio within a couple of weeks by someone experienced on that architecture.

Chapter 5

Analysis

This chapter details the tests that were performed to investigate the performance of the single stack kernel. It is important to test the kernel performance thoroughly in both average and worst case scenarios. Without both of these cases, the performance data could not be used as a solid basis to draw conclusions about the single stack kernel.

5.1 Methodology

The goal of the thesis is to measure and understand the tradeoffs in performance and memory between the single stack and multi stack kernel architectures. To this end the pistachio kernel has been modified only as much as neccesary to implement the single stack kernel. Because all code not related to the stack architecture is left unmodified, there is a strong basis for comparing the two kernels side by side. All optimisations have been implemented in both kernels or neither of the kernels to minimise bias and error in the results. This gives the results that can reasonably be compared. There may be further optimisations possible to increase the performance of the single stack kernel, such as merging the thread state and continuation fields in the TCB, however these optimisations have not been implemented as they would change too much of the kernel code to enable a fair comparison.

In determining the comparative performance of the overall system it is important to simulate a typical system workload. This can give an average case performance for the single stack vs. multi stack kernel. It is also important to find the worst case performance difference. The kernel stack architecture only affects kernel operations, not normal user calculations, so the worst case performance is achieved by code that performs no calculations other than invoking kernel operations. To measure the worst case performance penalties a series of micro-benchmarks have been developed to test common kernel operations.

The micro-benchmarks were developed based on the expected performance differences outlined in section 3. Each benchmark has been included for a specific purpose outlined below.

5.2 Test Environment

All benchmarks were carried out on a littlechips LN2410SBC single board computer [14]. This single board computer contains a Samsung S3C2410 arm processor clocked

at 200 MHz, a 32 kilobyte, 64 way associative cache, and 64 megabytes of ram. The board has additional peripherals such as a VGA touch screen, however these peripherals are not used in any of the performance tests. The littlechips board is capable of running Windows CE and Linux, and as such provides a useful test system similar to many advanced embedded systems which also run these operating systems.

The littlechips board's cache is highly associative, so not many caching benefits are expected in the code due to the single kernel stacks reduction of conflict misses. Additionally, because the ARM architectures TLB is refilled by hardware, TLB miss costs are not high, so any reduction in TLB miss rates by the single stack kernel is not expected to yield major performance improvements. Therefore the single stack kernel is expected to perform similarly to or worse than the multi stack kernel.

5.3 Micro-benchmarks

All of the performance micro-benchmarks are based around a similar design. They read the current system time, perform a kernel operation in a tight loop a large number of times, and then read the system time again. There is a small amount of overhead in reading the system time, as well as page faults that may occur the first time through the loop. However, the operation is performed at least 50000 times in each benchmark, so the error introduced by these overheads is very small. In any case the error in these overheads is common to all three kernels, so in a performance comparison it does not matter. In addition two memory benchmarks are conducted to observe the effect of the single stack kernel on the kernels memory usage.

5.3.1 Base Memory

The first test conducted is to use the systems memory tracing architecture to measure the amount of memory the kernel uses in a normal quiescent state immediately after boot-up. Two figures here are interesting for comparing kernel architectures: How much memory is used for kernel stacks/, and the proportion of total memory usage that this comprises. These figure are measured for each of the kernels, with the single and variable stack kernels expected to have lower memory usage than the multi stack kernel. The memory saving in this benchmark is expected to be small, as there are initially few threads in the system.

5.3.2 High Thread Count Memory

This benchmark gives the memory usage when there are a large number of threads active in the system. By creating one hundred threads, and then measuring the memory usage of the kernel, the true memory impact of the single kernel stack can be observed. The same figures are reported as for the base memory micro-benchmark. Obviously the single and variable stack kernels are expected to use a great deal less memory than the multi stack kernel in this situation.

5.3.3 Ping-Pong

IPC is the most important kernel operation of a micro-kernel [13]. Many of the IPC operations in L4 are handled by the fast-path, hence it is important to understand the

effect of the kernel stack architecture on the IPC fast-path. This is measured through the ping-pong micro-benchmark.

The *ping-pong* micro-benchmark consists of two threads communicating back and forth via IPC. The threads do no work other than that required for communication, so the time reported is essentially the time required for a basic IPC operation. Transferring memory registers incurs memory copying costs, which are the same all three kernels. Hence to measure the worst case performance, no message registers are transferred. This minimises the IPC time, and hence exposes any overhead as a greater proportion of the total IPC time.

The single stack kernel is expected to be slightly slower than the multi stack kernel in the ping-pong benchmark. However, as discussed in chapter 3, there are very few differences between the single stack kernel and the multi stack kernel in the IPC fast-path. Any observed increase in IPC time is expected to be due to the increased number of memory locations that the single stack kernel must access on the fast-path.

The variable kernel stack is expected to be slower again than the single stack kernel. This is because the variable stack kernel is required to check that IPC's destination thread does not have an associated kernel stack. This check is expected to slow execution of the IPC fast-path by a further few percent.

5.3.4 Null Syscall

A *null syscall* is a system call that does no work in the kernel but returns immediately to the user. The cost of a null system call is the inherent overhead of a system call. This overhead is incurred in all system calls, in addition to the cost of the work performed by the system call. System calls are made frequently, so the system call overhead is an important factor in system performance. The null system call can be approximated in L4 a thread switch system call, where the supplied argument is the currently executing thread. The kernel returns immediately to the user rather than attempting a context switch where the source and destination thread are the same thread.

It is expected that the single stack kernel will take longer to perform the null system call than the multi stack kernel. This is because the single stack kernel must save additional user registers compared to the multi stack kernel, as discussed in chapter 3. Additionally, the single stack kernel must load the kernel stack from a pointer in the TCB. It is expected that the single kernel stack will take marginally longer to perform a null system call than the multi stack kernel due to these extra operations.

5.3.5 Context Switch

A context switch changes the currently executing thread. It involves saving and loading register state and switching page tables. The currently executing thread can voluntarily invoke a context switch to allow another thread to make additional progress. A voluntary context switch is invoked by the thread switch system call, where the argument supplied is the thread to activate. The context switch micro-benchmark measures the time two threads take to switch back and forth a large number of times. This time measure operation yields an approximation of the context switch time. The EAS context switch benchmark is the same as the context switch benchmark, except that the threads are located in different address spaces. This benchmark yields the context switch time between address spaces, expected to be higher than the context switch within an address space.

It is expected that the context switch take approximately the same amount of time as the work performed is very similar in all three kernels.

5.3.6 Exception IPC

L4 allows *user level exception handling* by transforming exceptions into a simulated IPC message. Since system calls are an exception, user level exception handling enables the simulation of operating systems that rely on the use of the system call exception in user space. Since one of the current uses of L4 is to support the Wombat Linux server, implemented using user level exception handling, the performance of exception IPCs is important. The Wombat Linux server uses exception IPCs to convey system calls from Linux user processes to the user level Linux kernel. As such exception IPCs are very common in the Wombat Linux server. The exception IPC benchmark reports the cost of these exception IPCs when delivered within the same address space, whilst the EAS exception IPC benchmark reports the cost of exception IPCs delivered across address spaces. The cross address space exception IPC costs are expected to be higher then the intra address space exception IPCs by the same margin as the context switch and EAS context switch operations.

Exception IPCs are usually delivered by the exception IPC fast-path on ARM. This code is very similar to the IPC fast-path code, so the performance of the single stack kernel is again expected to be slightly worse than that of the multi stack kernel, although this is not expected to be a significant problem. Once again, the variable stack kernel must perform an additional check for an associated stack in the destination thread, so the variable stack kernel is expected to have the worst performance of the three kernels.

5.4 AIM7 Macro-benchmark

The AIM7 benchmark is a measures system performance by simulating workload on a multiuser system. The AIM7 benchmark was modified slightly so that it could run on Wombat. The modifications included disabling the network operation simulations because Wombat does not support the GetHost function, and disabling the file system operation simulations, because Wombat runs from a ram disk, and the ram disk is not large enough to support the benchmarks. The AIM7 benchmark is believed to be representative of a typical system workload, so it should be a good measure for the average case performance of the three kernels.

The precise benchmark used was 2 clients with the normal workload file, with the disk and network tests removed. All three kernels are expected to perform similarly, as the slightly longer kernel operations of the single stack kernel should be compensated for by a smaller cache and TLB footprint.

Chapter 6

Performance Results

The results of the benchmarks are supplied in this chapter in a concise format to allow easy reference. The implications of the results will be discussed in the next chapter.

Single number results have been given for the micro benchmarks. These benchmarks behaved in a deterministic manner, probably due to their small size and highly repetitive nature. As such there is no need for a statistical analysis of the results of the micro-benchmarks. See the previous chapter for a description of each of the tests.

Micro-benchmark	Single Stack	Multi Stack	Variable Stack
Base Memory (kb)	16/278	32/290	16/278
Memory For 100 threads (kb)	68/358	232/518	68/358
Ping Pong (μS)	2.08	2.06	2.12
Null System Call (μS)	1.828	1.612	1.828
Context Switch (μS)	2.266	2.273	2.36
EAS Context Switch (μS)	103	98	103
Exception IPC (μS)	2.861	2.843	2.889
EAS Exception IPC (μS)	103.4	99.6	103.5

The AIM7 benchmark scores varied slightly on each run of the benchmark set. Therefore the average and standard deviation have been given for 5 runs of the test. The particular AIM7 workload used was the standard workload with the disk and network tasks removed. This workload was run in two user tasks.

Quantity	Single Stack	Multi Stack	Variable Stack
Average Time	157.69	197.22	157.78
Standard Deviation	0.015	0.344	0.019

Chapter 7

Discussion of Performance

The results of the AIM7 benchmark are very unexpected. The three kernels were expected to perform similarly, or the single stack kernel was expected to perform worse due to the extra work it must perform. However, the single stack and variable stack kernels outperformed the multi stack kernel by an astonishing 20

Meanwhile, the micro-benchmarks showed the expected performance differences between the kernels. That is, the single kernel stack uses less memory than the multi stack kernel, but is slightly slower, while the variable stack kernel uses the same memory as the single stack kernel and is slightly slower again. The micro-benchmarks are treated below first to give a solid foundation for discussion of the surprising AIM7 result.

7.1 Memory Micro-benchmarks

The motivation for using a single kernel stack is to save memory. A static analysis of the code shows that the TCB size in the single stack kernel is 424 bytes, expanded from 196 bytes in the multi stack kernel. Thus there is an extra 228 bytes of memory needed to maintain the single kernels state without the stack. However, because the multi stack kernel combines its TCB and kernel stack, each thread uses 2 kilobytes of memory. The single kernel stack uses only the 424 bytes of memory per thread for its TCB. Due to the method by which TCBs are addressed by the fast-path, it is neccesary to align TCBs on 512 byte boundaries. The fast-path calculates the address of any TCB by shifting the bits in the thread identifier to obtain the address of the TCB. This calculation requires the TCB size to be a power of two. Therefore the effective TCB size in the single stack kernel is 512 bytes.

The single stack kernel therefore achieves a saving of 75% of the memory used per thread for TCBs and kernel stacks. This is a significant theoretical saving, but needs to be measured in practice. As shown in the Base memory benchmark, this saving accounts for an immediate saving of 16 kilobytes of memory due to TCBs in the kernel after initial boot-up. since 4 kilobytes is used for the shared kernel stack, this represents a total saving of 12 kilobytes of memory. This is an immediate saving of four percent of the total memory in use by the kernel.

After creating 100 threads, the memory saving is much greater as expected. In this case 164 kilobytes of memory is saved, in relation to the TCBs, with 160 kilobytes of memory saved in total. The total memory saving of over 30% is worthwhile for em-

bedded systems. It should be noted that this figure is essentially a best case figure, as it is unlikely any embedded system requires this many threads. In addition these threads have all been created in the same address space, so the memory used by the page tables and mapping database has only been increased by 4 kilobytes for one hundred threads. Thread creation in a normal system could be expected to increase other kernel memory usage such as page tables by more than this amount. This reduces the percentage memory saving, but not the overall memory saving. Hence, the single kernel stack represents a method for significant kernel memory use reduction for embedded systems.

7.2 Performance Micro-benchmarks

The other important goal of this thesis was to achieve comparable single stack kernel performance to multi stack kernel performance. Micro-benchmarks were used to measure the worst case performance of common kernel operations. The results were largely as expected, that is the single and variable stack kernels are slightly slower than the multi stack kernel. Each benchmark is discussed in turn below.

The ping-pong micro-benchmark establishes the systems performance in the IPC operation. As previously noted, IPC performance is the most important determinant of micro-kernel performance. Therefore the results of this test are critical. It is immediately apparent that the multi stack kernel performs IPC faster than the single and variable stack kernel, as expected by the analysis in chapter 3. However, the performance difference between the single and multi stack kernel is only 0.02 microseconds. This is only 10 processor cycles of the 200 MHz ARM processor, approximately 1% of the time required for an IPC operation. This performance penalty is acceptable in my view, if IPC performance is truly comparable to system performance, then there will only be a 1% slowdown in the overall system. However, since IPC is not the only operation performed in the system, this cost should be amortised to less than 1% of overall system performance. In my view the performance of IPC in the single stack kernel is not a hindrance to the use of the single stack kernel.

The null system call benchmark measures the overhead of a system call. The results show that the single kernel stack takes longer to make a system call than the multi stack kernel. This difference was expected as discussed in chapter 3. The 0.21 microsecond difference equates to approximately 105 processor cycles. This performance difference between the single and multi stack kernel is slightly higher than anticipated, but is a cost of the single stack kernel that cannot be avoided. The difference is caused by the necessity of storing the extra registers when making a system call, and reload them when returning to the user. These figures show the advantage of the multi stack kernel optimisation for system calls to give a speed gain of almost 10%. This performance penalty is not expected to excessively impact the single stack kernels overall system performance, as the extra 0.21 microseconds is expected to be a small percentage of the execution time of any system call that performs work in the kernel.

The two context switch benchmarks again bear out the expectations set forth in chapter 3. The single stack kernels context switch is actually marginally faster, even before subtracting the extra overhead involved in a single stack kernel system call. Subtracting system call costs reveals a speedup of approximately 0.21 microseconds for the single stack kernel as opposed to the multi stack kernel. This represents a significant saving on the cost of a context switch in a single stack kernel. This saving is not apparent in a cross address space context switch. In fact there is a large performance

penalty for the single stack kernel, requiring 2500 additional processor cycles for a context switch than the multi stack kernel. Because there is a complete cache and TLB flush when changing address spaces on ARM, the EAS context switch benchmark runs essentially uncached, the reason for its high cost. This makes the additional instructions used in single stack kernel system call entry and exit very expensive, as they are no longer cached in the instruction cache. This is believed to be the main determinant of this performance discrepancy.

The Exception IPC benchmarks are important for the performance of the Wombat Linux server, one of the main current uses for the L4 micro-kernel. The difference between the single stack and multi stack kernel is approximately the same as for the IPC Fast-path, as expected. Again the difference is almost insignificant, so it is not expected to greatly impact on overall system performance. Once again, the EAS exception costs are incredibly high due to the need to flush the cache and TLB. Again the single stack kernel suffers in the EAS benchmark due to the additional instructions it must load into the cache that are not used in the multi stack kernel.

It can be seen from the micro-benchmarks that the performance of the single stack kernel is quite close to the multi stack kernel. In no instance is there more than a 10% performance difference, and these are the operations expected to be most expensive on a single stack kernel. Therefore in overall system performance we should see no more than a 10% performance degradation, and ideally much less as the costs are not incurred constantly as they are in the micro-benchmarks.

7.3 AIM7 Macro-benchmark

To measure the complete system performance, the AIM7 benchmark was used. This was designed to show the difference in performance of the system in an actual usage scenario. The test was expected to show a slight performance penalty for the single and variable stack kernels due to the results from the micro-benchmarks. However the benchmark returned a very surprising 20

As with any experimental result, this needs to be treated with scepticism until it can be satisfactorily explained. Because the benchmark is not very stable and crashes on some runs, I initially doubted the results. Additionally, I had received advice that the timer in th Wombat Linux server may not be reliable.

I reran the benchmarks and timed the results with a wall clock to ensure that Wombat timers were not the reason behind the performance discrepancy. The wall clock agreed with the wombat timers to within a second in every case. This was not the reason for the performance discrepancy.

If the result is accurate, it must be due to reductions in the cache and TLB footprint of the single stack kernel. I did not expect that this reduction would make such a massive difference in performance, even though this is one of the reasons L4 outperforms Mach [12]. To determine the validity of this result a simulation of the cache impact of the kernels must be performed. There was not enough time to complete this simulation in the course of this thesis, due to external events.

7.4 Future Work

Further research into the single stack kernel is required. Specifically, it needs to be determined how the single stack kernel affects interrupt latency. Time did not permit the

modification of the multi stack kernel to make it interruptible, so any testing of interrupt latency in this thesis would not have produced reliable results. It is an interesting area of investigation, how does the single stack kernel affect interrupt latency, and can the variable stack kernel improve the interrupt latency performance.

Additionally, it is necessary to run more macro benchmarks on the system. Because the AIM7 benchmark showed a surprising result, another benchmark needs to be performed to try to refine the picture of performance under different workloads. Time constraints once again prevented this.

There is additional optimisation of the single stack kernel that can be performed. The space can be optimised by combining the state in the TCB into appropriate union data structures so that variables that are never used at the same time are written to the same place in memory. It is not known whether this optimisation will be able to reduce the TCB to 256 bytes, although it would represent a further significant memory saving if this were possible. An idea for a performance optimisation is to pass the current TCB as an argument to functions called from continuations. This is because these functions first operation is invariably to load the current TCB, and this could be passed more efficiently in a register than accessed from memory again. The impact on the performance of the kernel from this optimisation is not expected to be large, as the TCB value is heavily used, so it is usually cached.

Finally, this work needs to be expanded to other architectures, as well as multiprocessor machines. The tradeoffs involved with a single kernel stack have not previously been investigated on multi processor systems, and hence the performance effect is unknown. It is not expected to significantly affect the performance of these systems. Porting the kernel to other architectures will provide a more interesting basis for examining the architectural features that influence the performance of the single stack kernel. For instance, it is expected that the possible TLB miss savings will greatly enhance comparative performance on the MIPS architecture, where TLB misses are very expensive.

Chapter 8

Conclusions

To conclude, the single stack kernel has largley met expectations. The original motivation was to reduce kernel memory usage without significantly affecting overall system performance. It has met both of these goals.

The single stack kernel uses 75% less memory per thread than the multi stack kernel. This represents a total kernel memory saving of up to 30%.

The single stack kernel at least matches the performance of the multi stack kernel, as shown by the microbenchmarks, and in fact improves it by a great deal if the AIM7 benchmark is taken at face value.

Therefore it is concluded that the single stack kernel architecture provides an attractive alternative to the traditional threaded and event based kernel architectures. It is expected to be most useful in embedded systems where memory is at a premium. It allows embedded systems to use more threads without worrying about kernel memory limitations. If the performance difference shown in the AIM7 benchmark can be attributed to the single stack kernel architecture, It is also relevant to any system in which L4 is used.

Further investigation into these suprising results is needed, both on the ARM architecture and other architectures that L4 supports. The future looks bright for the single kernel stack architecture in L4.

Bibliography

[1] J. Bradley Chen and Brian N. Bershad. The impact of operating system structure on memory system performance. In *Proceedings of the 14th ACM Symposium on OS Principles*, pages 120–133, Asheville, NC, USA, December 1993.

[2] Francois Barbou des Places, Nick Stephen, and Franklin D. Reynolds. Linux on the OSF Mach3 microkernel. Technical report, 1996. http://www.gr.osf.org/~stephen/fsf96.ps.

[3] Richard P. Draves. *Control Transfer in Operating System Kernels*. PhD thesis, Carnegie Mellon University, May 1994.

[4] R.P. Draves, Brian N. Bershad, R.F. Rashid, and R.W. Dean. Using continuations to implement thread management and communication in operating systems. In *Proceedings of the 13th ACM Symposium on OS Principles*, Asilomar, CA, USA, October 1991.

[5] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on OS Principles*, pages 251–266, Copper Mountain, CO, USA, December 1995.

[6] Brian Ford, Mike Hibler, Jay Lepreau, Roland McGrath, and Patrick Tullmann. Interface and execution models in the Fluke kernel. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, pages 101–115, New Orleans, LA, USA, February 1999. USENIX.

[7] David Golub, Randall Dean, Allesandro Forin, and Richard Rashid. Unix as an application program. In *Proceedings of the 1990 Summer USENIX Technical Conference*, June 1990.

[8] Andreas Haeberlen and Kevin Elphinstone. User-level management of kernel memory. In *Proceedings of the 8th Asia-Pacific Computer Systems Architecture Conference*, volume 2823 of *Lecture Notes in Computer Science*, Aizu-Wakamatsu City, Japan, September 2003. Springer Verlag.

[9] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of μ-kernel-based systems. In *Proceedings of the 16th ACM Symposium on OS Principles*, pages 66–77, St. Malo, France, October 1997.

[10] Dan Hildebrand. An architectural overview of QNX. In *Proceedings of the USENIX Workshop on Microkernels and other Kernel Architectures*, pages 113–126, Seattle, WA, USA, 1992.

[11] Jochen Liedtke. Improving IPC by kernel design. In *Proceedings of the 14th ACM Symposium on OS Principles*, pages 175–88, Asheville, NC, USA, December 1993.

[12] Jochen Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on OS Principles*, pages 237–250, Copper Mountain, CO, USA, December 1995.

[13] Jochen Liedtke. Towards real microkernels. *Communications of the ACM*, 39(9):70–77, September 1996.

[14] littlechips.com. LN2410SBC/TFT. http://www.littlechips.com/LN2410SBC_TFT.htm.

[15] Abi Nourai. A physically-addressed L4 kernel. BE thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, March 2005.

[16] David A. Patterson and John L. Hennessy. *Computer Organization: Hardware/Software Interface*, chapter 7. Morgan Kauffman, 2 edition, 2003.

[17] M. Rozier, V. Abrossimov, F. Armand, L. Boule, M. Gien, M. Guillemont, F. Herrman, C. Kaiser, S. Langlois, P. Leonard, , and W. Neuhauser. Overview of the Chorus distributed operating system. pages 39–69, Seattle, WA, USA, 1992.

[18] Andrew S. Tanenbaum and Sape J. Mullender. An overview of the Amoeba distributed operating system. *Operating Systems Review*, 15(3):51–64, 1981.

[19] L4Ka Team. L4ka::pistachio kernel. http://l4ka.org/projects/pistachio/.

[20] L4Ka Team. L4 eXperimental kernel reference manual version X.2. Technical report, University of Karlsruhe, 2001. http://l4ka.org/projects/version4/l4-x2.pdf.